Utrecht University
Bachelor Kunstmatige Intelligentie

A forward chaining theorem prover for the extended Lambek calculus

Author:
Emmelie Slotboom
5654394

Supervisor:
Prof. dr. Michael Moortgat
Second evaluator:
Dr. Benjamin Rin

15 ECTS
November 19, 2019

Abstract

Natural Language Processing is one of the main branches of Artificial Intelligence, in which Lambek calculus is used to make deductive proofs about properties of language structures, which are presented as symbols. It can be done automatically with the implementation of an algorithm. This thesis constructs such an algorithm and presents a usable Python program that follows this algorithm to form proofs with the deductive rules of Lambek calculus. The construction of the algorithm was based on an extended form of Lambek calculus, in which the range of provable theorems more closely represents natural language than the system without any extensions. Forward chaining was used, so the tree structure of a linguistic input is formed automatically as well. It is not yet verified whether this algorithm can form every possible proof, so a formal analysis and proof concerning this is recommended as further research.

Contents

Introduction **1**

1 Typelogical grammars **3**
1.1 Lambek Calculus . . . 3
1.1.1 Syntactic types and type computations . . . 4
1.1.2 Sequent calculus . . . 5
1.2 Extended Lambek calculus . . . 7
1.2.1 Control modalities . . . 7
1.2.2 Controlled structural rules . . . 8
1.2.3 Examples . . . 8
1.3 Focused display calculus . . . 9
1.3.1 Focusing . . . 9
1.3.2 Display calculus . . . 9
1.4 Semantics . . . 11

2 The parsing procedure **15**
2.1 Parsing procedures . . . 15
2.1.1 Backward chaining . . . 16
2.1.2 Forward chaining . . . 16
2.2 Implementation . . . 17
2.2.1 Partial proof trees . . . 17
2.2.2 Breadth First Search . . . 19

Conclusion & further research **22**

References **24**

A Examples **25**

B Theorem Prover **41**

Introduction

The aim of Artificial Intelligence is to imitate human behavior. Something that is typically human is the ability to use language, so Natural Language Processing (NLP) is one of the main branches of Artificial Intelligence. Within NLP there is the symbolic approach, in which symbols are used to represent linguistic information, and the non-symbolic machine learning approach. In this thesis the symbolic approach is used, so NLP is presented as a reasoning problem. This combines the field of linguistics with logic: another important branch of Artificial Intelligence.

A deductive system that lends itself well for this purpose is Lambek calculus, in which a string of words is translated to types and given a tree structure and goal type. By using the deductive rules of the calculus it is decided whether the words together are of the goal type. With additional linguistic information, the semantic interpretation of the words can be calculated as well. The construction of proofs in Lambek calculus is a decision procedure, so an algorithm can be constructed that automatically forms every possible proof for a given string of words and goal type.

This thesis consists of two components: the textual part, in which the theoretical background and parsing procedure are described, and the code, with which the algorithm can be run. In the first chapter of the textual part Lambek calculus is introduced and extended with control modalities and structural rules, so proofs can be made for a wide enough range of language structures. The proof system is also converted to its focused display variant with the purpose of making the proof procedure more restrictive, yet simpler to compute. In the last section the translation to a semantic interpretation is given. The second chapter focuses on the parsing procedure, in which a method is created that allows an unstructured string of words to be structured and proven simultaneously by using both forward and backward chaining. The program will apply the method of chapter 2 to create every proof for a given theorem. It also gives a semantic interpretation and generates a LaTeX file that formats the proof or proofs that belong to the given sequent.

The construction of forward chaining theorem provers for typelogical grammars has been done before (Jumelet, 2017, Moot, 2016). However, an extended Lambek

calculus prover had not been written in Python yet and a theorem prover is often not the most straightforward to use. This thesis will provide a prover with code that is easy to understand and use. It was done in Python, because of its NLP applications. This way it will be easy to integrate the prover with existing NLP tools, like the NLTK toolkit. Python is also a well-known programming language and code written in it is often very clear to follow.

Chapter 1

Typelogical grammars

This chapter is an overview of the theoretical background of the typelogical systems that will be used in this thesis. It starts with Lambek's original syntactic calculus and its sequent presentation, which is the basis for decidable proof search. Expressive limitations of the original calculus will be addressed, and the extended Lambek systems are introduced, which will be the subject of the implementation efforts. In the extended Lambek systems unary operations are added, with which it is possible to formulate restricted methods to reorder and restructure grammatical material. They can be used to block unwanted cases of overgeneration as well.

Sequent calculus by itself is not very efficient for proof search. To obtain a presentation of the type logic with good computational properties, sequent calculus will be reformulated in display format. A focusing method will be introduced as well to eliminate spurious ambiguity, which means that it will no longer be possible for a theorem to have multiple proofs that only differ in irrelevant rule orderings.

At the end of the chapter it is shown how a syntactic derivation can be associated with a term of the linear lambda calculus. These terms can be seen as programs with which the meaning of a phrase can be computed, given the meanings of the words that it is composed of.

1.1 Lambek Calculus

Lambek calculus was introduced in 1958 and revisited in 1961 by Joachim Lambek. He explains the goals for his writings as follows:

> "The aim of this paper is to obtain an effective rule (or algorithm) for distinguishing sentences from nonsentences, which works not only for the formal languages of interest to the mathematical logician, but also for natural languages such as English, or at least for fragments of such languages." –Lambek, 1958

1.1.1 Syntactic types and type computations

In the Lambek calculus, the familiar categories (noun, verb, adjective, adverb, and so on) are replaced by types. Starting from a set of atomic types p (where s will be used for sentences, n for simple nouns and np for noun phrases) the full type language is presented by grammar 1.1.

$$A, B \quad ::= \quad p \quad | \quad A/B \quad | \quad B\backslash A \quad | \quad A \otimes B \tag{1.1}$$

A product type $A \otimes B$ describes a phrase that consists of a phrase of type A followed by a phrase of type B. The types A/B (read A over B) and $B\backslash A$ (read B under A) are used for incomplete phrases: an expression of type A/B combines with an expression of type B to its right to form an expression of type A. Similarly, an expression of type $B\backslash A$ combined with an expression of type B to its left produces an expression of type A.

With this interpretation the operations $/, \otimes, \backslash$ are a residuated triple. From this the rules in 1.2, 1.3 and 1.4 are obtained, in which 1.2 is identity, 1.3 transitivity and 1.4 residuation.

$$A \to A \tag{1.2}$$

$$\text{if } A \to B \text{ and } B \to C \text{ then } A \to C \tag{1.3}$$

$$A \to C/B \quad \text{iff} \quad A \otimes B \to C \quad \text{iff} \quad B \to A\backslash C \tag{1.4}$$

Together, they form the non-associative calculus **NL** (Lambek, 1961). Adding the associativity laws from 1.5 results in the associative calculus **L** (Lambek, 1958).

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

The difference between **L** and **NL** is that in the former types are assigned to strings, whereas in the latter types are assigned to bracketed strings, or, more formally, phrase structure trees.

From the residuation laws, identity and transitivity, type combination schemata like the so-called application rules in 1.6 can be derived. In these rules the explicit product operation is omitted.

$$B(B\backslash A) \to A \qquad (A/B)B \to A \tag{1.6}$$

Because this works recursively longer sentences like “John never works” or “John works and Jane rests” can be formed. To be able to assign types to these sentences in **NL** they need to be given a tree structure, so it is clear how the types

are grouped. Tree structures are shown with parentheses as in examples 1.7 and 1.8.

$$\begin{array}{cccccc} \text{John} & & (\text{never} & & \text{works}) & \\ np & (((np\backslash s)/(np\backslash s)) & & (np\backslash s)) & \rightarrow & s \end{array} \tag{1.7}$$

$$\begin{array}{cccccccc} (\text{John} & \text{works}) & & (\text{and} & & (\text{Jane} & \text{rests})) & \\ (np & (np\backslash s)) & (((s\backslash s)/s) & & & (np & (np\backslash s)) & \rightarrow & s \end{array} \tag{1.8}$$

Note that there may be multiple ways to form sentence structures and types. For example if the sentence "John works and Jane rests" was structured as "((John works) and) (Jane rests)" the word "and" would have needed to be of the type $s\backslash(s/s)$.

The type combination schemata in 1.6 reduce the type complexity, but there are also type transitions that increase complexity. An instance of this is 1.9, in which the lifting rules are presented.

$$A \rightarrow B/(A\backslash B) \qquad A \rightarrow (B/A)\backslash B \tag{1.9}$$

Because of transitivity and complexity increasing type transitions, Lambek's goal of obtaining an effective procedure for distinguishing sentences from non-sentences is a non-trivial task. In the next section it will be shown how Lambek reformulated the residuation based presentation to an equivalent sequent calculus, for which a decision procedure can be given.

1.1.2 Sequent calculus

A sequent is a statement $X \Rightarrow A$ where X is called the antecedent, which should be a non-empty structure and A the consequent: a type formula (Moortgat, 2014). A structure X or Y is formed as in 1.10, where A is a type formula. The structural connective $\cdot \otimes \cdot$ is a structure building operation corresponding to the connective $\otimes$, as the rules will show. The notation $X[Y]$ will be used for a structure X that contains a distinguished substructure Y.

$$X, Y \quad ::= \quad A \quad | \quad X \cdot \otimes \cdot Y \tag{1.10}$$

The rules for the deductive systems **NL** and **L** are presented in Figure 1.1. The L and R rules are are inference rules that introduce a connective on the left (L) or on the right (R) side of the arrow. The cut rule is a staple in many deduction systems. It is admissible in sequent calculus as well, but no additional language structures can be derived with it, and thus it will not be used in this thesis. Lambek only included the global associativity rules in his 1958 writings

and not the ones from 1961, because a problem with the global associativity rules being included as they are is that they make the system overgenerate. This means that some ungrammatical sentences like "The mother of and John thinks that Bill left" can be derived (Moot, 2015). To avoid this, for now the non-associative rule system **NL** will be used.

$$\frac{}{A \Rightarrow A}\ \mathsf{Ax} \qquad \frac{X \Rightarrow A \quad Y[A] \Rightarrow B}{Y[X] \Rightarrow B}\ \mathsf{cut}$$

$$\frac{Y \Rightarrow B \quad X[A] \Rightarrow C}{X[Y \cdot \otimes \cdot (B \backslash A)] \Rightarrow C}\ \backslash L \qquad \frac{B \cdot \otimes \cdot X \Rightarrow A}{X \Rightarrow B \backslash A}\ \backslash R$$

$$\frac{X[A] \Rightarrow C \quad Y \Rightarrow B}{X[(A/B) \cdot \otimes \cdot Y] \Rightarrow C}\ /L \qquad \frac{X \cdot \otimes \cdot B \Rightarrow A}{X \Rightarrow A/B}\ /R$$

$$\frac{X[A \cdot \otimes \cdot B] \Rightarrow C}{X[A \otimes B] \Rightarrow C}\ \otimes L \qquad \frac{X \Rightarrow A \quad Y \Rightarrow B}{X \cdot \otimes \cdot Y \Rightarrow A \otimes B}\ \otimes R$$

$$\frac{W[(X \cdot \otimes \cdot Y) \cdot \otimes \cdot Z] \Rightarrow A}{W[X \cdot \otimes \cdot (Y \cdot \otimes \cdot Z)] \Rightarrow A}\ \mathsf{as}$$

Figure 1.1: Rules of the sequent calculus **L**. The rules labeled as can be omitted to obtain the non-associative variant **NL**.

An example derivation for the sentence "John never works" can be found in Figure A.1. Every atomic formula is provided with an index so it can be easily followed throughout the derivation tree.

Applying rules to a sequent with the system **NL** is a decision procedure, because the amount of cut-free derivations that can be attempted for a sequent is finite and a sequent is deducible if and only if at least one of the attempted derivations is successful (Lambek, 1958).

Even though sequent calculus is a great starting point for a deductive system for natural language, it is not very reliable when searching for derivations for grammatical language structures only. In the following sections the system will be modified so it is more usable for natural language structures.

1.2 Extended Lambek calculus

A shortcoming of sequent calculus with global associativity is that it is only able to generate context-free languages (Pentus, 1997), and natural languages go beyond that (Shieber, 1985). However, including global associativity causes the deductive system to overgenerate. In this section the calculus will be expanded to allow properties like associativity to be controlled.

1.2.1 Control modalities

The system presented so far uses atomic (s, n, np, ...) and binary (A/B, $B\backslash A$, $A \otimes B$) formulas. In this section new rules will be included that use unary connectives. This is done to influence the proof system in two ways: to allow a controlled form of properties like associativity and commutativity to be included in the system, but also to disallow otherwise provable derivations for ungrammatical language structures. For both of these features a new set of connectives will be introduced: $\diamondsuit$ and $\square$ to enable more derivations, and $\blacklozenge$ and $\blacksquare$ to block ungrammatical ones.

The connectives $\diamondsuit$ and $\square$ are added and a type formula A or B with atomic type p is redefined as in definition 1.11.

$$\begin{array}{rcl} A, B & ::= & p \mid A/B \mid B\backslash A \mid A \otimes B \mid \\ & & \diamondsuit A \mid \square A \mid \blacklozenge A \mid \blacksquare A \end{array} \tag{1.11}$$

The redefinition of a structure can be found in definition 1.12, in which $\cdot \diamondsuit \cdot$ and $\cdot \blacklozenge \cdot$ are introduced as structural counterparts to $\diamondsuit$ and $\blacklozenge$ respectively.

$$X, Y ::= A \mid X \cdot \otimes \cdot Y \mid \cdot \diamondsuit \cdot X \mid \cdot \blacklozenge \cdot X \tag{1.12}$$

The rules presented in Figure 1.2 are added, which use the new type formulas and structures (Moortgat, 1996).

$$\frac{X \Rightarrow A}{\cdot \$ \cdot X \Rightarrow \$A}\ \$R \qquad \frac{X[\cdot \$ \cdot A] \Rightarrow B}{X[\$A] \Rightarrow B}\ \$L$$

$$\frac{\cdot \$ \cdot X \Rightarrow A}{X \Rightarrow \#A}\ \#R \qquad \frac{X[A] \Rightarrow B}{X[\cdot \$ \cdot \#A] \Rightarrow B}\ \#L$$

Figure 1.2: Rules with the diamond and box operators, where \$ and # are either $\diamondsuit$ and $\square$ or $\blacklozenge$ and $\blacksquare$ respectively

1.2.2 Controlled structural rules

The modalities allow reintroduction of associativity, because it can now be controlled so it does not overgenerate. A similar property, commutativity, will be introduced as well. Two sets of rules are now included: rules 1.13 are to be used in the Dutch language, rules 1.14 in the English language.

$$\frac{V[((\cdot \diamondsuit \cdot W) \cdot \otimes \cdot X) \cdot \otimes \cdot Y] \Rightarrow Z}{V[(\cdot \diamondsuit \cdot W) \cdot \otimes \cdot (X \cdot \otimes \cdot Y)] \Rightarrow Z}\ \mathsf{nl}$$
$$\frac{V[X \cdot \otimes \cdot ((\cdot \diamondsuit \cdot W) \cdot \otimes \cdot Y)] \Rightarrow Z}{V[(\cdot \diamondsuit \cdot W) \cdot \otimes \cdot (X \cdot \otimes \cdot Y)] \Rightarrow Z}\ \mathsf{nl} \tag{1.13}$$

$$\frac{V[W \cdot \otimes \cdot (X \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y))] \Rightarrow Z}{V[(W \cdot \otimes \cdot X) \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y)] \Rightarrow Z}\ \mathsf{en}$$
$$\frac{V[(W \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y)) \cdot \otimes \cdot X] \Rightarrow Z}{V[(W \cdot \otimes \cdot X) \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y)] \Rightarrow Z}\ \mathsf{en} \tag{1.14}$$

1.2.3 Examples

An example where unary connectives are needed, but not associativity or commutativity is with the word "without" as in the sentence "Alice left without closing the window", which is initially translated to pre-sequent 1.15. This type assignment produces the desired derivation in Figure A.3, but it also allows a derivation for the ungrammatical "window that Alice left without closing", as can be seen in Figure A.4. To block this, the the type of "without" will be changed as in pre-sequent 1.16, with which the derivation in A.5 can be formed.

Alice	left	without	closing	the	window		
np	$np\backslash s$	$((np\backslash s)\backslash(np\backslash s))/gp$	gp/np	np/n	n	$\rightsquigarrow$	s

(1.15)

$$np, np\backslash s, (\cdot \blacklozenge \cdot \blacksquare((np\backslash s)\backslash(np\backslash s)))/gp, gp/np, np/n, n \rightsquigarrow s \tag{1.16}$$

An example where additional structural rules are used can be found in Figure A.6.

As was illustrated with these examples, the deductive system has been made more precise in what can and cannot be derived. Before more theorems are proven, the system will be improved once more by limiting the amount of derivations to one per meaning, which will be done in the next section.

1.3 Focused display calculus

With the expansion of the proof system there are two more alterations left to apply to it. First the system will be restricted to disallow excessive derivations. After this the system will be converted to its display variant.

1.3.1 Focusing

In a proof system that forms derivations for a language structure it is convenient if every derivation corresponds to one meaning and vice versa. Ambiguous structures would thus be the only ones that have multiple derivations. This is not yet the case, as can be seen with the sentence "Alice bakes the cake" which has the two derivations in Figure A.2a, but only one meaning. To solve this focused proof search is introduced.

In focused proof search rules can only be applied if the relevant formulas are of the correct polarity. Formulas of the form $A \otimes B$ and $\Diamond A$ are of a positive polarity, formulas of the form A/B, $B\backslash A$ and $\Box A$ are negative. Atomic formulas are assigned an arbitrary polarity. Structures can never be focused. A neutral sequent is a sequent without a focused formula. A focused formula A is presented as $\boxed{A}$ (Bastenhof, 2011, Moortgat and Moot, 2011). An example derivation will be given using the display representation of sequent calculus, as shown in the next section.

1.3.2 Display calculus

The system presented so far will be used to prove theorems with. However, as it uses substructures it may sometimes be difficult for a machine to decide which formulas are substructures and which are not. To avoid this problem the sequent calculus and its extensions will be translated to an equivalent display calculus.

In such a display calculus substructures can be isolated because of the display property. The display property implies that a substructure in a sequent can be rewritten (or displayed) as the entire antecedent or consequent, but not both (Ciabattoni, Ramanayake, and Wansing, 2014). Every other structure is moved to the other side of the sequent.

Sequent calculus and its extensions can be systematically transformed to a display calculus, as described by Ciabattoni et al. (2014). Because any calculus has inference rules a template can be followed to change a rule to its display variant. This change involves the introduction of new connectives: a structural counterpart for every formula connective. Some of these connectives were already used in the grammar for a structure, which can be found again in 1.17, where A is a type formula.

$$X, Y \;::=\; A \;\mid\; X \cdot \otimes \cdot Y \;\mid\; \cdot \diamondsuit \cdot X \;\mid\; \cdot \blacklozenge \cdot X \tag{1.17}$$

Because structures with these connectives always appear as antecedents in the rules, they will be referred to as input structures. The other structural connectives are introduced as output structures P in definition 1.18, and will only appear as consequents. In this definition, A is a type formula again, and Y an input structure.

$$P \;::=\; A \;\mid\; P \cdot / \cdot Y \;\mid\; Y \cdot \backslash \cdot P \;\mid\; \cdot \Box \cdot P \;\mid\; \cdot \blacksquare \cdot P \tag{1.18}$$

Besides the axiom and cut rules, the central idea behind the construction of a display calculus is once again residuation. This takes shape in the rules labeled rp and rc in Figure 1.4. The remaining rules in this figure are rewrite rules: one for every connective. An overview of all of the display rules with focusing can be found in Figures 1.3, 1.4 and 1.5 (Moortgat and Moot, 2011). Because the concept of substructures is not used anymore, the controlled structural rules are now presented as in 1.19 and 1.20. Figure A.2 shows how the sentence "Alice bakes the cake" is derived in both the previous and current proof system variants. The derivations for example 1.21 can be found in Figure A.6. Throughout the thesis np and n will be given a positive polarity and s a negative polarity unless stated otherwise.

The current system will be used for the parsing and proving procedure. It was extended with modalities and structural rules, so a decent range of derivations can be produced. Focused proof search was applied to the system; as a result redundant proofs can not be derived anymore. The system was also presented as a display calculus to ease its use in the parsing procedure. The meaning of language structure was mentioned before as well, but only intuitively. In the next section it will be explained more extensively.

$$\frac{}{A \Rightarrow \boxed{A}}\,\mathsf{Ax} \qquad \frac{X \Rightarrow \boxed{A}}{X \Rightarrow A}\,{\rightharpoonup} \qquad \frac{A \Rightarrow Y}{\boxed{A} \Rightarrow Y}\,{\leftharpoondown}$$

$$\frac{}{\boxed{A} \Rightarrow A}\,\mathsf{CoAx} \qquad \frac{\boxed{A} \Rightarrow Y}{A \Rightarrow Y}\,{\leftharpoonup} \qquad \frac{X \Rightarrow A}{X \Rightarrow \boxed{A}}\,{\rightharpoondown}$$

Figure 1.3: (Co)axioms and (de)focusing rules for display sequent calculus. To apply a rule in the first row the focused formula must be positive, in the second row it must be negative.

$$\frac{\dfrac{X \Rightarrow Z \cdot / \cdot Y}{X \cdot \otimes \cdot Y \Rightarrow Z}\,\mathsf{rp}}{Y \Rightarrow X \cdot \backslash \cdot Z}\,\mathsf{rp} \qquad \frac{\cdot \$ \cdot X \Rightarrow Y}{X \Rightarrow \cdot \# \cdot Y}\,\mathsf{rc}$$

$$\frac{X \Rightarrow \cdot \# \cdot A}{X \Rightarrow \# A}\,\#R \qquad \frac{\cdot \$ \cdot A \Rightarrow Y}{\$A \Rightarrow Y}\,\$L$$

$$\frac{X \Rightarrow A \cdot / \cdot B}{X \Rightarrow A/B}\,/R \qquad \frac{A \cdot \otimes \cdot B \Rightarrow Y}{A \otimes B \Rightarrow Y}\,\otimes L \qquad \frac{X \Rightarrow B \cdot \backslash \cdot A}{X \Rightarrow B \backslash A}\,\backslash R$$

Figure 1.4: Structural and rewrite rules for display sequent calculus, where \$ and # are either $\diamondsuit$ and $\square$ or $\blacklozenge$ and $\blacksquare$ respectively

$$\begin{array}{l} \dfrac{((\cdot \diamondsuit \cdot W) \cdot \otimes \cdot X) \cdot \otimes \cdot Y \Rightarrow Z}{(\cdot \diamondsuit \cdot W) \cdot \otimes \cdot (X \cdot \otimes \cdot Y) \Rightarrow Z}\,\mathsf{nl} \\[2ex] \dfrac{X \cdot \otimes \cdot ((\cdot \diamondsuit \cdot W) \cdot \otimes \cdot Y) \Rightarrow Z}{(\cdot \diamondsuit \cdot W) \cdot \otimes \cdot (X \cdot \otimes \cdot Y) \Rightarrow Z}\,\mathsf{nl} \end{array} \tag{1.19}$$

$$\begin{array}{l} \dfrac{W \cdot \otimes \cdot (X \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y)) \Rightarrow Z}{(W \cdot \otimes \cdot X) \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y) \Rightarrow Z}\,\mathsf{en} \\[2ex] \dfrac{(W \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y)) \cdot \otimes \cdot X \Rightarrow Z}{(W \cdot \otimes \cdot X) \cdot \otimes \cdot (\cdot \diamondsuit \cdot Y) \Rightarrow Z}\,\mathsf{en} \end{array} \tag{1.20}$$

$$\begin{array}{l} \text{lakei (die (Alice plaagt))} \\ \text{lackey (who (Alice teases))} \\ \text{lackey whom Alice teases} \\ \text{lackey who teases Alice} \end{array} \tag{1.21}$$

1.4 Semantics

The final linguistic property that can be extracted from a derivation is a meaning of the language structure. The fact that a sentence like “Everybody admires someone” has more than one meaning can be shown with the proof system. This

$$\frac{X \Rightarrow \boxed{A}}{\cdot \$ \cdot X \Rightarrow \boxed{\$A}}\ \$R \qquad \frac{\boxed{A} \Rightarrow Y}{\boxed{\#A} \Rightarrow \cdot \# \cdot Y}\ \#L$$

$$\frac{\boxed{A} \Rightarrow X \quad Y \Rightarrow \boxed{B}}{\boxed{A/B} \Rightarrow X \cdot / \cdot Y}\ /L \qquad \frac{Y \Rightarrow \boxed{B} \quad \boxed{A} \Rightarrow X}{\boxed{B\backslash A} \Rightarrow Y \cdot \backslash \cdot X}\ \backslash L$$

$$\frac{X \Rightarrow \boxed{A} \quad Y \Rightarrow \boxed{B}}{X \cdot \otimes \cdot Y \Rightarrow \boxed{A \otimes B}}\ \otimes R$$

Figure 1.5: Rules with focused formulas for display sequent calculus, where $\$ \in \{\Diamond, \blacklozenge\}$ and $\# \in \{\Box, \blacksquare\}$

is done by translating a derivation, which shows how the proof term is built up rule by rule. One such term represents one meaning. If the separate proof terms of each of the words are given, they can be inserted into the derivation term. The term can then be shortened using β-reduction.

Figure 1.6 visualizes the theoretical procedures behind the translation. Figures 1.7, 1.8 and 1.9 show the rules with their corresponding proof term manipulations (Bastenhof, 2011, Moortgat and Moot, 2011). In Figure A.7 the first derivation from Figure A.6 is shown, where the sequents are substituted by their proof term and residuation rules are omitted. The second derivation has a very similar proof term, as the only difference is that the parts $\lceil \mathsf{Alice} \rceil$ and x_3 are switched in the tuple $\langle \lceil \mathsf{Alice} \rceil, \langle x_3, \alpha_4 \rangle \rangle$, which results in $\langle x_3, \langle \lceil \mathsf{Alice} \rceil, \alpha_4 \rangle \rangle$ instead.

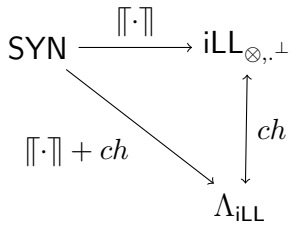

Figure 1.6: The translations between a syntactical derivation SYN, the semantic derivation $\mathsf{iLL}_{\otimes,.^{\perp}}$ and the proof term Λ_{iLL}. The notation $[\![\cdot]\!]$ represent a translation in continutation passing style. The letters *ch* stand for the Curry-Howard correspondence.

$$\frac{}{\underset{x}{x : A \Rightarrow \boxed{A}}}\ \mathsf{Ax} \qquad \frac{\overset{M}{X \Rightarrow \boxed{A}}}{\underset{(\alpha\ M)}{X \Rightarrow \alpha : A}}\ \rightharpoonup \qquad \frac{\overset{M}{x : A \Rightarrow Y}}{\underset{\lambda x.M}{\boxed{A} \Rightarrow Y}}\ \leftharpoonup$$

$$\frac{}{\underset{\alpha}{\boxed{A} \Rightarrow \alpha : A}}\ \mathsf{CoAx} \qquad \frac{\overset{M}{\boxed{A} \Rightarrow Y}}{\underset{(x\ M)}{x : A \Rightarrow Y}}\ \leftharpoonup \qquad \frac{\overset{M}{X \Rightarrow \alpha : A}}{\underset{\lambda\alpha.M}{X \Rightarrow \boxed{A}}}\ \rightharpoonup$$

Figure 1.7: (Co)axioms and (de)focusing rules for display sequent calculus with their interpretation. To apply a rule in the first row the focused formula must be positive, in the second row it must be negative.

$$\frac{\dfrac{X \Rightarrow Z \cdot / \cdot Y}{\overline{\overline{X \cdot \otimes \cdot Y \Rightarrow Z}}}\ \mathsf{rp}}{\underline{\underline{Y \Rightarrow X \cdot \backslash \cdot Z}}}\ \mathsf{rp} \qquad \frac{\cdot \$ \cdot X \Rightarrow Y}{\underline{\underline{X \Rightarrow \cdot \# \cdot Y}}}\ \mathsf{rc}$$

$$\frac{\overset{M}{X \Rightarrow \cdot \# \cdot \alpha : A}}{\underset{\texttt{case}\ \beta\ \texttt{of}\ \langle\alpha\rangle.M}{X \Rightarrow \beta : \# A}}\ \#R \qquad \frac{\overset{M}{\cdot \$ \cdot x : A \Rightarrow Y}}{\underset{\texttt{case}\ y\ \texttt{of}\ \langle x\rangle.M}{y : \$A \Rightarrow Y}}\ \$L$$

$$\frac{\overset{M}{X \Rightarrow \alpha : A \cdot / \cdot x : B}}{\underset{\texttt{case}\ \beta\ \texttt{of}\ \langle\alpha, x\rangle.M}{X \Rightarrow \beta : A/B}}\ /R \qquad \frac{\overset{M}{X \Rightarrow x : B \cdot \backslash \cdot \alpha : A}}{\underset{\texttt{case}\ \beta\ \texttt{of}\ \langle x, \alpha\rangle.M}{X \Rightarrow \beta : B \backslash A}}\ \backslash R$$

$$\frac{\overset{M}{x : A \cdot \otimes \cdot y : B \Rightarrow Y}}{\underset{\texttt{case}\ z\ \texttt{of}\ \langle x, y\rangle.M}{z : A \otimes B \Rightarrow Y}}\ \otimes L$$

Figure 1.8: Structural and rewrite rules for display sequent calculus with their interpretation, where \$ and # are either $\Diamond$ and $\Box$ or $\blacklozenge$ and $\blacksquare$ respectively

$$\frac{\overset{M}{X \Rightarrow \boxed{A}}}{\underset{\langle M \rangle}{\cdot \$ \cdot X \Rightarrow \boxed{\$A}}}\ \$R \qquad \frac{\overset{M}{\boxed{A} \Rightarrow Y}}{\underset{\langle M \rangle}{\boxed{\#A} \Rightarrow \cdot \# \cdot Y}}\ \#L$$

$$\frac{\overset{M}{\boxed{A} \Rightarrow Z} \quad \overset{N}{Y \Rightarrow \boxed{B}}}{\underset{\langle M, N \rangle}{\boxed{A/B} \Rightarrow Z \cdot / \cdot Y}}\ /L \qquad \frac{\overset{M}{Y \Rightarrow \boxed{B}} \quad \overset{N}{\boxed{A} \Rightarrow Z}}{\underset{\langle M, N \rangle}{\boxed{B \backslash A} \Rightarrow Y \cdot \backslash \cdot Z}}\ \backslash L$$

$$\frac{\overset{M}{X \Rightarrow \boxed{A}} \quad \overset{N}{Y \Rightarrow \boxed{B}}}{\underset{\langle M, N \rangle}{X \cdot \otimes \cdot Y \Rightarrow \boxed{A \otimes B}}}\ \otimes R$$

Figure 1.9: Rules with focused formulas for display sequent calculus with their interpretation, where $\$ \in \{\Diamond, \blacklozenge\}$ and $\$ \in \{\Box, \blacksquare\}$

Chapter 2

The parsing procedure

Instead of deciding the tree structure(s) of an expression by hand, it is preferable that the prover forms it by itself. This means that the correct method for parsing expressions must be chosen. In this chapter two parsing procedures will be explored: backward and forward chaining. After this the usable characteristics of both parsing methods will be taken to form an algorithm for the theorem prover. The steps of the algorithm will be described and examples will be given.

2.1 Parsing procedures

A description of parsing is as follows:

> "Parsing can be viewed as a deductive process that seeks to prove claims about the grammatical status of a string from assumptions describing the grammatical properties of the string's elements and the linear order between them." –Shieber, Schabes, and Pereira, 1994

Applied to the aims of this thesis, this means that parsing can be used to form the deductions and tree structure for a string of words and its goal type as in 2.1, where every w is a word and A_{n+1} a type formula.

$$w_1\ w_2\ ...\ w_n \rightsquigarrow A_{n+1} \tag{2.1}$$

The words are translated to type formulas, after which the pre-sequent 2.2 is formed.

$$A_1, A_2, ..., A_n \rightsquigarrow A_{n+1} \tag{2.2}$$

Pre-sequent 2.2 is the input for the parsing procedure. There are multiple ways to parse such a pre-sequent, so it is important to choose the correct ones to use

with the deductive system. In this section the strengths and weaknesses of two popular parsing procedures will be discussed.

2.1.1 Backward chaining

Also known as the top-down or goal-driven procedure, backward chaining in deductive systems like extended Lambek calculus starts at the root of the derivation tree and seeks goals in which every leaf is an axiom or a co-axiom. A problem arises when the tree structure of the expression is not given and global associativity is not included in the system. In this situation following a pure backward chaining algorithm would involve attempting a proof for every possible tree structure for the expression. This is very inefficient, because the amount of tree structures in a sequence of n words is the nth Catalan number[1]. Thus a simple expression with only five type formulas like "John works and Jane rests" has 42 different tree structures and with an expression that consists of six formulas this number increases to 132.

Even though a pure backward chaining algorithm is not an option, it may be useful when parsing (sub)structures of length one or two.

2.1.2 Forward chaining

The procedure of forward chaining, also referred to as bottom-up or data-driven, takes the opposite approach to backward chaining. In a deductive system it starts at the leafs of a derivation tree and seek goals in which the conclusion of the tree is the structured goal sequent. It is not necessary that a tree structure is given, which is exactly what is needed. However, pure forward chaining would in this case be undesirable. This will be illustrated with the example sentence "John works and Jane rests", which translates to pre-sequent 2.3.

$$np_0, np_1 \backslash s_2, (s_3 \backslash s_4)/s_5, np_6, np_7 \backslash s_8 \rightsquigarrow s_9 \tag{2.3}$$

A derivation tree is formed from its leafs, so the first step is to create the axioms. To avoid forming impossible axioms, the input and output properties will be utilized first. Based on these properties, it can already be determined which formulas will be an antecedent of an axiom and which ones will be a consequent; this is done as follows. Let every type formula in the antecedent of a sequent be an input and the type formula in the consequent an output. Which atoms are inputs and which are outputs can be deduced from Table 2.1. In example 2.3 the type

[1]This is the case in a system without control modalities. If they are included the amount of possible tree structures is even higher.

formulas np_0, s_2, s_4, np_6 and s_8 are input atoms and np_1, s_3, s_5, np_7 and s_9 are output atoms.

Input ($\downarrow$)	$p^{\downarrow}$	$A^{\downarrow}/B^{\uparrow}$	$B^{\uparrow}\backslash A^{\downarrow}$	$A^{\downarrow} \otimes B^{\downarrow}$	$\Diamond A^{\downarrow}$	$\Box A^{\downarrow}$
Output ($\uparrow$)	$p^{\uparrow}$	$A^{\uparrow}/B^{\downarrow}$	$B^{\downarrow}\backslash A^{\uparrow}$	$A^{\uparrow} \otimes B^{\uparrow}$	$\Diamond A^{\uparrow}$	$\Box A^{\uparrow}$

Table 2.1: Positions in formulas

Because input atoms can only form an axiom with output atoms and vice versa, and the antecedent and consequent of an axiom must be of the same type, there are two combinations for axioms with np and six for co-axioms with s possible, resulting in twelve combinations of axioms. Increasing the amount of atomic formulas in the theorem will make the algorithm significantly worse at finding a solution quickly.

In the next sections these problems will be minimized by incorporating backward chaining into the forward chaining algorithm.

2.2 Implementation

To optimize the process of constructing a proof for a given theorem, an algorithm will be described that uses forward and backward chaining by going through the following steps. Partial derivations are made for the type formulas in an input pre-sequent that takes the form of 2.2. Together they will be combined into the final derivation(s) by applying structural rules and connecting their premises and conclusions. This will be done with the use of Breadth First Search.

Throughout the chapter the algorithm will be explained using the example "lakei die Alice plaagt", which is translated to pre-sequent 2.4.

$$n_0, (n_1\backslash n_2)/((\Diamond\Box np_3)\backslash s_4), np_5, np_6\backslash(np_7\backslash s_8) \rightsquigarrow n_9 \tag{2.4}$$

2.2.1 Partial proof trees

In this section backward chaining will be used to form parts of the desired derivation(s) for a pre-sequent. Therefore these partial proofs are formed from the conclusion towards the premises.

A smaller proof for every lexical item of the antecedent and for the consequent of the sequent can be constructed and will be part of any of its goal proofs (Joshi and Kulick, 1997). Such a smaller proof is made with a sequent as the conclusion where the type formula is one side (the antecedent or consequent) and an incomplete structure on the other. Such an incomplete structure will be referred to as unknown, which is described in Definition 2.2.1. A smaller proof will be called a

partial proof tree; see Definition 2.2.2. PPTs in this thesis are similar to the PPTs described by Joshi and Kulick (1997).

Definition 2.2.1. *An unknown atom is an atom $p_?$, where p is an atomic type. An unknown structure is a structure where not every substructure is shown, but instead denoted with a variable.*

Definition 2.2.2. *A partial proof tree (PPT) is a sub tree of a derivation in extended Lambek calculus, where some structures may be unknown.*

At least one PPT is constructed for every type formula A in pre-sequent 2.5. This results in set 2.6, where every function $\texttt{ppts}(A_i)$ results in a set of PPTs that is unfolded from its corresponding type formula A_i.

$$A_1, A_2, ..., A_n \rightsquigarrow A_{n+1} \tag{2.5}$$

$$\texttt{ppts}(A_1) \cup \texttt{ppts}(A_2) \cup ... \cup \texttt{ppts}(A_n) \cup \texttt{ppts}(A_{n+1}) \tag{2.6}$$

The root (or conclusion) of a PPT that is unfolded from a type formula A is a sequent where A is changed to the form $A \Rightarrow X$ if A is an input formula and $X \Rightarrow A$ if A is an output formula, and X is an unknown structure. A PPT t is unfolded by attempting the following extensions:

1. t is extended with a non-structural rule application for every leaf in which the antecedent or consequent is a type formula A. The rule that is applied is the one in which the type formula in the conclusion of the rule matches the position (input or output), connective and focusing of A. Any new structural connectives are added if present in the rule.

2. A new PPT is constructed for every non-atomic structure $A \,\$\, B$ or $\$\, A$ in the leaf(s), where A and B are type formulas and $\$$ a structural connective.

Any new or changed PPT is unfolded, so the algorithm keeps extending and creating PPTs until no more extensions or additions are possible. The PPTs corresponding to example sequent 2.4 can be found in Figure A.8, where a non-atomic unknown structure is presented as $X_{i \in \mathbb{N}}$.

When all PPTs are constructed for a pre-sequent, they will be connected using structural rules in order to find the final derivation(s). This will be discussed in the next section.

2.2.2 Breadth First Search

Because backward chaining was used as much as possible with the PPTs, the attention will be shifted to the forward chaining procedure. For this method it is necessary to start from the axioms, so the algorithm will operate on a PPT that contains only axioms as leafs. This PPT will be called the main proof S. If there are multiple possibilities for S, any of those can be chosen arbitrarily. The process of combining S with other PPTs is described using the concept of compatibility, which is introduced in Definitions 2.2.3 and 2.2.4.

Definition 2.2.3. *Two structures X and Y are compatible if and only if*

- *X or Y is unknown, not an atom $p_?$ and does not have any connectives, or*
- *X is an atom p_m and Y is an atom p_n where $m = n$ or $m =?$ or $n =?$, or*
- *X is of the form $\$\,W$ and Y is of the form $\#\,Z$ where $\$$ and $\#$ are unary connectives, and $\$ = \#$, and W is compatible with Z, or*
- *X is of the form $U\,\$\,V$ and Y is of the form $W\,\#\,Z$ where $\$$ and $\#$ are binary connectives, and $\$ = \#$, and U is compatible with W and V is compatible with Z.*

Definition 2.2.4. *Two sequents $X \Rightarrow Y$ and $W \Rightarrow Z$ are compatible if and only if*

- *one of the sequents is neutral and the other is either neutral or an axiom, and*
- *X is compatible with W, and*
- *Y is compatible with Z.*

The main proof S can be transformed in the following ways:

1. an axiom $p_? \Rightarrow p_m$ or $p_m \Rightarrow p_?$ in S is completed by combining S with a PPT that consists only of a sequent $p_n \Rightarrow X$ or $X \Rightarrow p_n$, given that the axiom and sequent are compatible, or
2. a structural rule is applied to the root of S, or
3. S is combined with another PPT R, given that the root of S is compatible with a leaf of R.

The transformations can be recorded in a search tree, where every transformation is an edge and the set of PPTs a node. To ensure that every solution is found the transformations are applied Breadth First, which means that every node at a given depth is expanded before any deeper nodes. To avoid unnecessary branches in the search tree, any node that is a duplicate of another node that was explored before will not be expanded, and neither will one that contains a PPT in which the same sequent appears two or more times.

Because only the main proof S can be transformed, it is important that it is a PPT where all of its leafs are axioms. This way it is not possible that S could have been combined with another PPT R if R was transformed first. This means that a new main proof is chosen whenever S is transformed to contain at least one leaf that is not an axiom[2]. A node is a solution when it only contains S and its root is the structured goal sequent. The algorithm is done when the search tree can not be expanded anymore.

The search tree corresponding to the example sequent consists of hundreds of nodes, so the complete derivation tree will not be shown. To give an example for how the BFS algorithm works, the first few steps will be explained. As can be seen in Figure A.8, there are seven PPTs for the example sequent 2.4, and PPTs (c) and (g) contain only axioms as leafs. Before starting the BFS procedure, one of those is chosen as the main proof. It does not matter which one; in this explanation (g) is arbitrarily selected as the main proof. Now the search tree in Figure A.9 can be built, which has the seven PPTs in Figure A.8 as the root. This node has three descendants: (g) can be combined with (b) in two ways, resulting in the PPTs (bg1) and (bg2), and a structural rule can be applied to (g), creating the PPT (g1). Now all of the descendants in the next depth are expanded. The nodes with PPTs (bg1) and (bg2) both have one extension through structural rule application, which results in the PPTs (bg1.1) and (bg2.1) respectively. Five transformations are possible for (g1): three structural rules can be applied and (g1) can be combined with (b) via two axioms again. However, the node has only two descendants, because one structural rule application would have resulted in a PPT that contains sequent 2.7 twice, and if (g1) was combined with (b) both of the resulting nodes would have been duplicates of the nodes that contain the PPTs (bg1.1) or (bg2.1).

$$np_6 \backslash (np_7 \backslash np_8) \Rightarrow np_? \cdot \backslash \cdot (np_? \cdot \backslash \cdot s_?) \tag{2.7}$$

The order of node expansion is: the node with the main proof (g), (bg1), (bg2), (g1), (bg1.1), (bg2.1), (g1.1) and (g1.2) followed by the direct descendants of the nodes with the latter four main proofs.

[2] An example where this occurs is with the sequent $p/q \Rightarrow p/q$ when p has a positive and q a negative polarity.

With the theoretical background in these chapters a Python program was written that proves theorems in extended Lambek calculus and uses forward chaining and PPTs. The code for this program can be found in the appendix.

Conclusion & further research

A theorem prover was made with Python code that follows the parsing procedure described in this thesis. Given an unstructured sequent, the polarities of its atomic types and its focusing, this procedure uses backward and forward chaining to efficiently find every possible proof in extended Lambek calculus and translates it to its semantic interpretation.

To give insight into the systems and processes that were used in the prover, the textual part was written, which was divided into two parts. First, Lambek calculus was introduced. The initial deductive rules had problematic properties and thus the calculus was extended and modified to fit its purposes better. In the second part a parsing procedure was given that uses the extended Lambek calculus to parse and prove the before mentioned theorems.

A prover like the one in this thesis has been written before. However, this prover is the first one that is written in Python: a popular programming language, that allows the use of many Natural Language Processing tools.

The theorem prover is useful in the fields of linguistics, logic and Artificial Intelligence, as it systematizes language by calculating properties of linguistic structures, with the use of extended Lambek calculus in combination with a parsing procedure.

Even though the written program works as desired, there are still a couple of points of improvement. Firstly, it is not very efficient when solving relatively long or complex inputs. The algorithm could therefore be improved to limit the expansion of the search tree by finding nodes that will not lead to a solution earlier in the process. A second point of improvement concerns a data structure that was used in the program. After the proof procedure, the found solution(s) are converted to strings that can be compiled to LaTeX. This process is made easier by changing the data structure of the solution to a recursive definition of a tree. It may be beneficial if a tree is represented as this recursive one from the start of the algorithm. To do this, part of the code needs to be rewritten.

Besides adjusting the efficiency of the program, there are still more features that can be added. Five recommendations will be mentioned. First off, more structural rules can be used with the program by writing them in code. However,

with the systematic nature of the deductive rules it is possible to write the form of the rules in a text format. Therefore functions could be implemented that read and convert these rules to a usable format for the program, similar to how the lexicon is compiled. The program could also profit off of a User Interface instead of needing to run the code through queries. Furthermore, a limitation of the program is that currently the lexicon assign a unique type to every word, even though some words have multiple types. It could therefore benefit from a modification where multiple types can be assigned to a single word. Another welcome addition is the before mentioned feature where a proof term can be simplified using β-reduction when proof terms for words are given. The last suggested feature is about the generative power of the proof system. Even though the extended Lambek calculus that is currently used handles a lot of language properties well, it still is not able to derive every grammatical language structure. More extensions could be added to the system, like the connectives that were introduced with the Lambek-Grishin calculus (Moortgat and Moot, 2011).

A last point of discussion is about what further research can be done considering the algorithm. The program seems to be able to derive all of the desired proofs. However, even though the theoretic background was studied and the code was tested thoroughly, a formal analysis and proof are needed to verify this.

References

Bastenhof, A. (2011). Polarized Montagovian Semantics for the Lambek-Grishin calculus. *CoRR*, *abs/1101.5757*.

Ciabattoni, A., Ramanayake, R., & Wansing, H. (2014). Hypersequent and Display Calculi - a Unified Perspective. *Studia Logica: An International Journal for Symbolic Logic*, *102*(6), 1245–1294.

Joshi, A. K. & Kulick, S. (1997). Partial Proof Trees as Building Blocks for a Categorial Grammar. *Linguistics and Philosophy*, *20*(6), 637–667.

Jumelet, J. (2017). *Bottom-up parsing for the extended typelogical grammars*.

Lambek, J. (1958). The Mathematics of Sentence Structure. *The American Mathematical Monthly*, *65*, 154–170.

Lambek, J. (1961). On the calculus of syntactic types. *Structure of Language and Its Mathematical Aspects*, *12*, 166–178.

Moortgat, M. (1996). Multimodal Linguistic Inference. *Journal of Logic, Language and Information*, *5*(3), 349–385.

Moortgat, M. (2014). Typelogical Grammar. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2014). Metaphysics Research Lab, Stanford University.

Moortgat, M. & Moot, R. (2011). Proof nets for the Lambek-Grishin calculus. *CoRR*, *abs/1112.6384*.

Moot, R. (2015). Comparing and evaluating extended Lambek calculi. *CoRR*, *abs/1506.05561*.

Moot, R. (2016). The Grail theorem prover: Type theory for syntax and semantics. *CoRR*, *abs/1602.00812*.

Pentus, M. (1997). Product-Free Lambek Calculus and Context-Free Grammars. *Journal of Symbolic Logic*, *62*, 648–660.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, *8*(3), 333–343.

Shieber, S. M., Schabes, Y., & Pereira, F. C. N. (1994). Principles and Implementation of Deductive Parsing. *CoRR*, *abs/cmp-lg/9404008*.

Appendix A

Examples

In this appendix the examples from chapters 1 and 2 can be found, as well as the outputs from the queries in the file examples.py.

$$\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow np_1}\,\mathsf{Ax} \quad \dfrac{}{s_2 \Rightarrow s_7}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot (np_1 \backslash s_2) \Rightarrow s_7}\,\backslash L \quad \dfrac{\dfrac{\dfrac{}{np_3 \Rightarrow np_5}\,\mathsf{Ax} \quad \dfrac{}{s_6 \Rightarrow s_4}\,\mathsf{Ax}}{np_3 \cdot \otimes \cdot (np_5 \backslash s_6) \Rightarrow s_4}\,\backslash L}{np_5 \backslash s_6 \Rightarrow np_3 \backslash s_4}\,\backslash R}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/(np_3 \backslash s_4)) \cdot \otimes \cdot (np_5 \backslash s_6)) \Rightarrow s_7}\,/L$$

Figure A.1: Derivation for the sentence “John never works”

$$\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow np_1}\,\mathsf{Ax} \quad \dfrac{}{s_2 \Rightarrow s_7}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot (np_1 \backslash s_2) \Rightarrow s_7}\,\backslash L \quad \dfrac{\dfrac{}{np_4 \Rightarrow np_3}\,\mathsf{Ax} \quad \dfrac{}{n_6 \Rightarrow n_5}\,\mathsf{Ax}}{(np_4/n_5) \cdot \otimes \cdot n_6 \Rightarrow np_3}\,/L}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot ((np_4/n_5) \cdot \otimes \cdot n_6)) \Rightarrow s_7}\,/L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow np_1}\,\mathsf{Ax} \quad \dfrac{}{s_2 \Rightarrow s_7}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot (np_1 \backslash s_2) \Rightarrow s_7}\,\backslash L \quad \dfrac{}{np_4 \Rightarrow np_3}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot np_4 \Rightarrow s_7}\,/L \quad \dfrac{}{n_6 \Rightarrow n_5}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot ((np_4/n_5) \cdot \otimes \cdot n_6)) \Rightarrow s_7}\,/L$$

(a) The regular sequent calculus **NL**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow \boxed{np_1}}\,\mathsf{Ax} \quad \dfrac{}{\boxed{s_2} \Rightarrow s_7}\,\mathsf{CoAx}}{\boxed{np_1 \backslash s_2} \Rightarrow np_0 \cdot \backslash \cdot s_7}\,\backslash L \quad \dfrac{}{np_4 \Rightarrow \boxed{np_3}}\,\mathsf{Ax}}{\boxed{(np_1 \backslash s_2)/np_3} \Rightarrow (np_0 \cdot \backslash \cdot s_7) \cdot / \cdot np_4}\,/L}{(np_1 \backslash s_2)/np_3 \Rightarrow (np_0 \cdot \backslash \cdot s_7) \cdot / \cdot np_4}\,\leftharpoondown}{((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot np_4 \Rightarrow np_0 \cdot \backslash \cdot s_7}\,\mathsf{rp}}{np_4 \Rightarrow ((np_1 \backslash s_2)/np_3) \cdot \backslash \cdot (np_0 \cdot \backslash \cdot s_7)}\,\mathsf{rp} \quad \dfrac{}{n_6 \Rightarrow \boxed{n_5}}\,\mathsf{Ax}}{\boxed{np_4} \Rightarrow ((np_1 \backslash s_2)/np_3) \cdot \backslash \cdot (np_0 \cdot \backslash \cdot s_7)}\,\leftharpoondown \quad \dfrac{}{n_6 \Rightarrow \boxed{n_5}}\,\mathsf{Ax}}{\boxed{np_4/n_5} \Rightarrow (((np_1 \backslash s_2)/np_3) \cdot \backslash \cdot (np_0 \cdot \backslash \cdot s_7)) \cdot / \cdot n_6}\,/L}{np_4/n_5 \Rightarrow (((np_1 \backslash s_2)/np_3) \cdot \backslash \cdot (np_0 \cdot \backslash \cdot s_7)) \cdot / \cdot n_6}\,\leftharpoondown}{(np_4/n_5) \cdot \otimes \cdot n_6 \Rightarrow ((np_1 \backslash s_2)/np_3) \cdot \backslash \cdot (np_0 \cdot \backslash \cdot s_7)}\,\mathsf{rp}}{((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot ((np_4/n_5) \cdot \otimes \cdot n_6) \Rightarrow np_0 \cdot \backslash \cdot s_7}\,\mathsf{rp}}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot ((np_4/n_5) \cdot \otimes \cdot n_6)) \Rightarrow s_7}\,\mathsf{rp}}{np_0 \cdot \otimes \cdot (((np_1 \backslash s_2)/np_3) \cdot \otimes \cdot ((np_4/n_5) \cdot \otimes \cdot n_6)) \Rightarrow \boxed{s_7}}\,\rightharpoonup$$

(b) Focused display calculus

Figure A.2: Derivations for the sentence "Alice bakes the cake", with the polarities np, n: + and s: -

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_3 \Rightarrow np_1}\,\mathsf{Ax} \quad \dfrac{}{s_2 \Rightarrow s_4}\,\mathsf{CoAx}}{np_3 \cdot \otimes \cdot (np_1 \backslash s_2) \Rightarrow s_4}\backslash L}{np_1 \backslash s_2 \Rightarrow np_3 \backslash s_4}\backslash R}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((np_3 \backslash s_4) \backslash (np_5 \backslash s_6))) \Rightarrow s_{13}}\backslash L \quad \dfrac{\dfrac{}{np_0 \Rightarrow np_5}\,\mathsf{Ax} \quad \dfrac{}{s_6 \Rightarrow s_{13}}\,\mathsf{CoAx}}{np_0 \cdot \otimes \cdot (np_5 \backslash s_6) \Rightarrow s_{13}}\backslash L}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)) / gp_7) \cdot \otimes \cdot gp_8)) \Rightarrow s_{13}} / L \quad \dfrac{\dfrac{}{gp_8 \Rightarrow gp_7}\,\mathsf{Ax} \quad \dfrac{}{np_{10} \Rightarrow np_9}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((((((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)) / gp_7) \cdot \otimes \cdot ((gp_8 / np_9) \cdot \otimes \cdot np_{10}))) \Rightarrow s_{13}} / L \quad \dfrac{}{n_{12} \Rightarrow n_{11}}\,\mathsf{Ax}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((((((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)) / gp_7) \cdot \otimes \cdot ((gp_8 / np_9) \cdot \otimes \cdot ((np_{10} / n_{11}) \cdot \otimes \cdot n_{12})))) \Rightarrow s_{13}} / L$$

Figure A.3: Derivation for pre-sequent 1.15

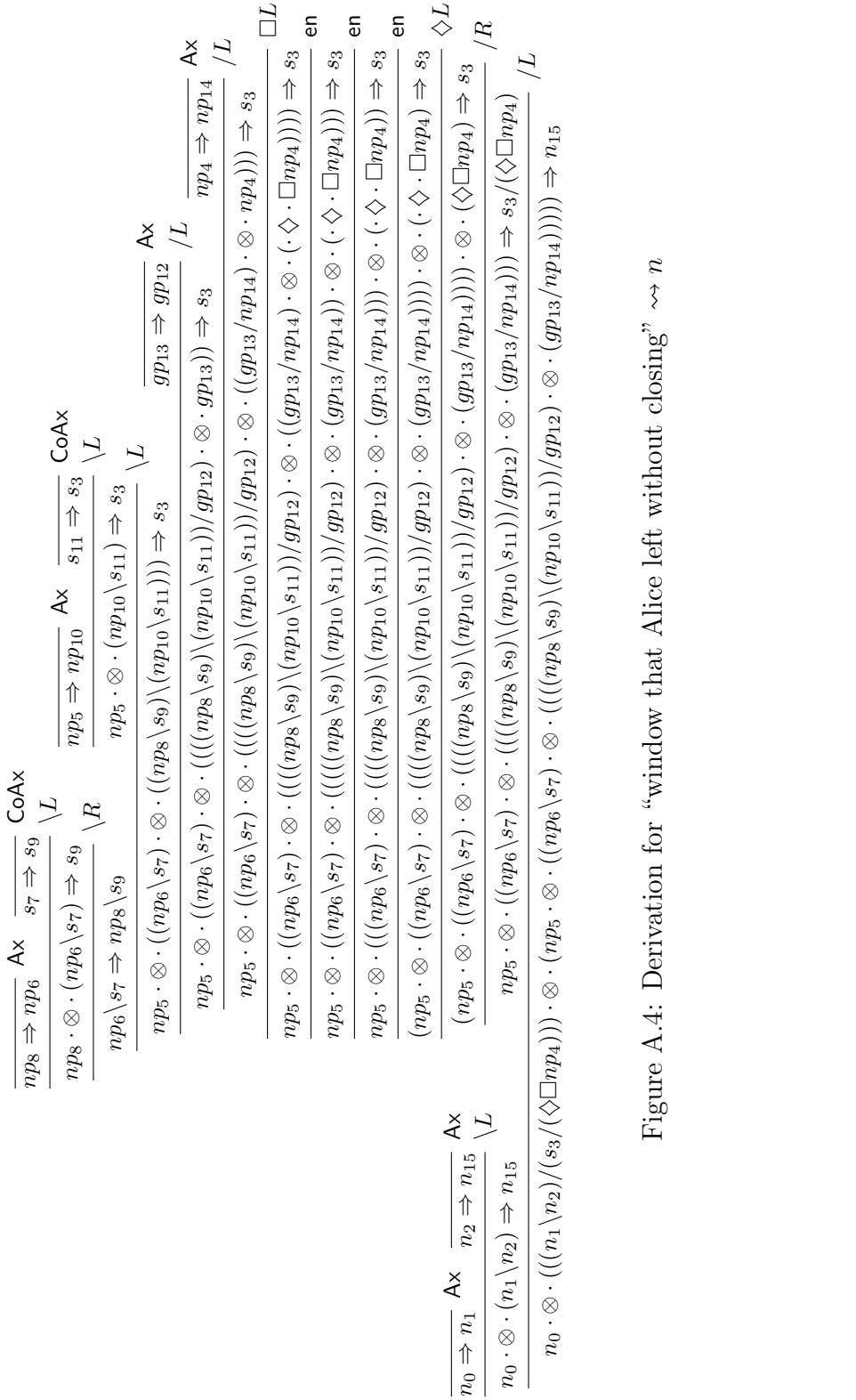

Figure A.4: Derivation for "window that Alice left without closing" $\rightsquigarrow n$

$$\dfrac{\dfrac{\dfrac{}{np_3 \Rightarrow np_1}\,\textsf{Ax} \quad \dfrac{}{s_2 \Rightarrow s_4}\,\textsf{CoAx}}{np_3 \cdot \otimes \cdot (np_1 \backslash s_2) \Rightarrow s_4}\,\backslash L}{np_1 \backslash s_2 \Rightarrow np_3 \backslash s_4}\,\backslash R$$

$$\dfrac{\dfrac{}{np_0 \Rightarrow np_5}\,\textsf{Ax} \quad \dfrac{}{s_6 \Rightarrow s_{13}}\,\textsf{CoAx}}{np_0 \cdot \otimes \cdot (np_5 \backslash s_6) \Rightarrow s_{13}}\,\backslash L$$

$$\dfrac{np_1 \backslash s_2 \Rightarrow np_3 \backslash s_4 \quad np_0 \cdot \otimes \cdot (np_5 \backslash s_6) \Rightarrow s_{13}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((np_3 \backslash s_4) \backslash (np_5 \backslash s_6))) \Rightarrow s_{13}}\,\backslash L$$

$$\dfrac{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot ((np_3 \backslash s_4) \backslash (np_5 \backslash s_6))) \Rightarrow s_{13}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))) \Rightarrow s_{13}}\,\blacksquare L$$

$$\dfrac{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))) \Rightarrow s_{13} \quad \dfrac{}{gp_8 \Rightarrow gp_7}\,\textsf{Ax}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (((\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))/gp_7) \cdot \otimes \cdot gp_8)) \Rightarrow s_{13}}\,/L$$

$$\dfrac{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (((\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))/gp_7) \cdot \otimes \cdot gp_8)) \Rightarrow s_{13} \quad \dfrac{}{np_{10} \Rightarrow np_9}\,\textsf{Ax}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (((\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))/gp_7) \cdot \otimes \cdot ((gp_8/np_9) \cdot \otimes \cdot np_{10}))) \Rightarrow s_{13}}\,/L$$

$$\dfrac{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (((\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))/gp_7) \cdot \otimes \cdot ((gp_8/np_9) \cdot \otimes \cdot np_{10}))) \Rightarrow s_{13} \quad \dfrac{}{n_{12} \Rightarrow n_{11}}\,\textsf{Ax}}{np_0 \cdot \otimes \cdot ((np_1 \backslash s_2) \cdot \otimes \cdot (((\cdot \blacklozenge \cdot \blacksquare((np_3 \backslash s_4) \backslash (np_5 \backslash s_6)))/gp_7) \cdot \otimes \cdot ((gp_8/np_9) \cdot \otimes \cdot ((np_{10}/n_{11}) \cdot \otimes \cdot n_{12})))) \Rightarrow s_{13}}\,/L$$

Figure A.5: Derivation for pre-sequent 1.16

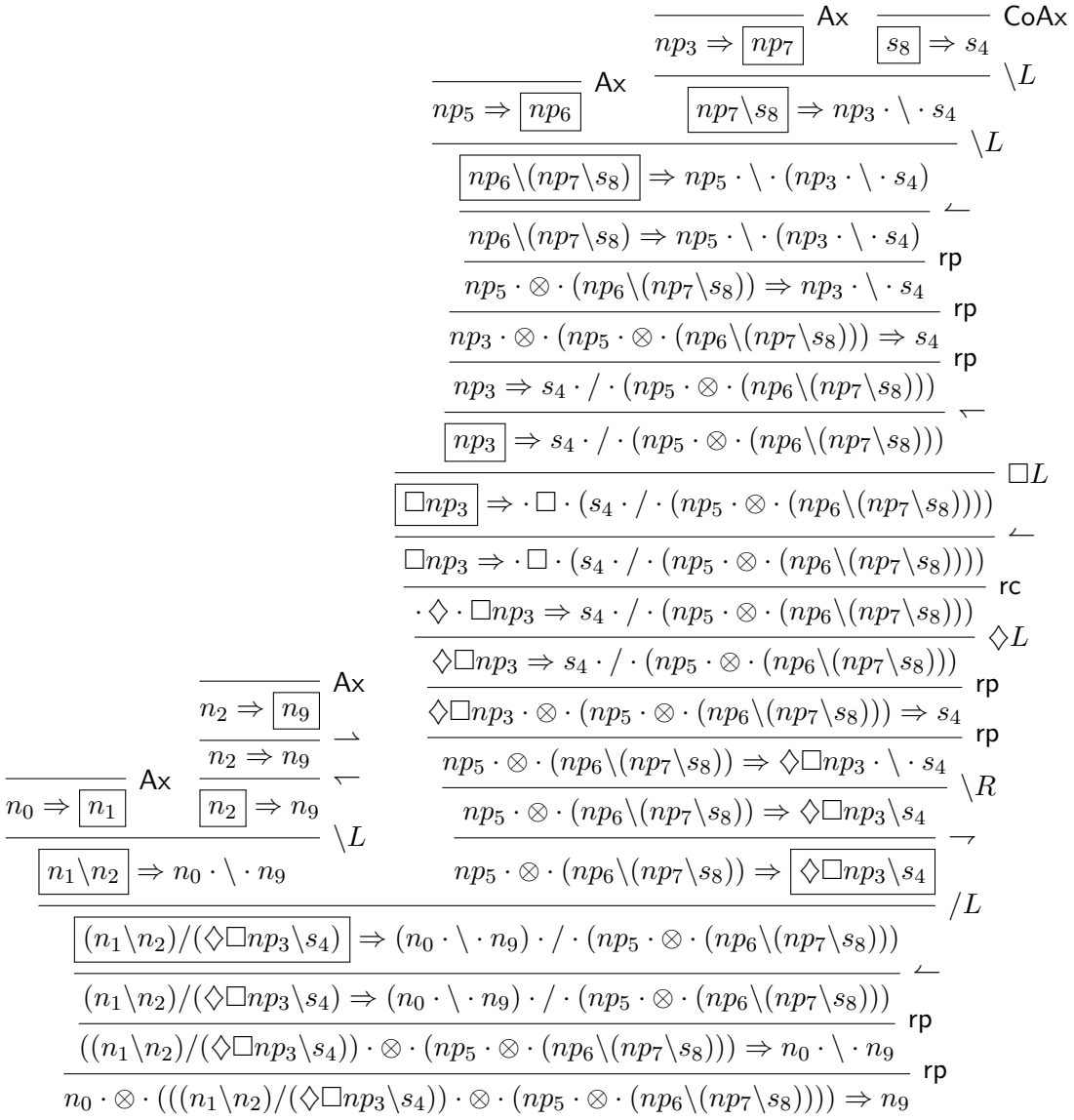

Figure A.6: Derivation for "lakei die Alice plaagt" $\rightsquigarrow n$, with the polarities np, n: + and s: -, which translates to "lackey who teases Alice"

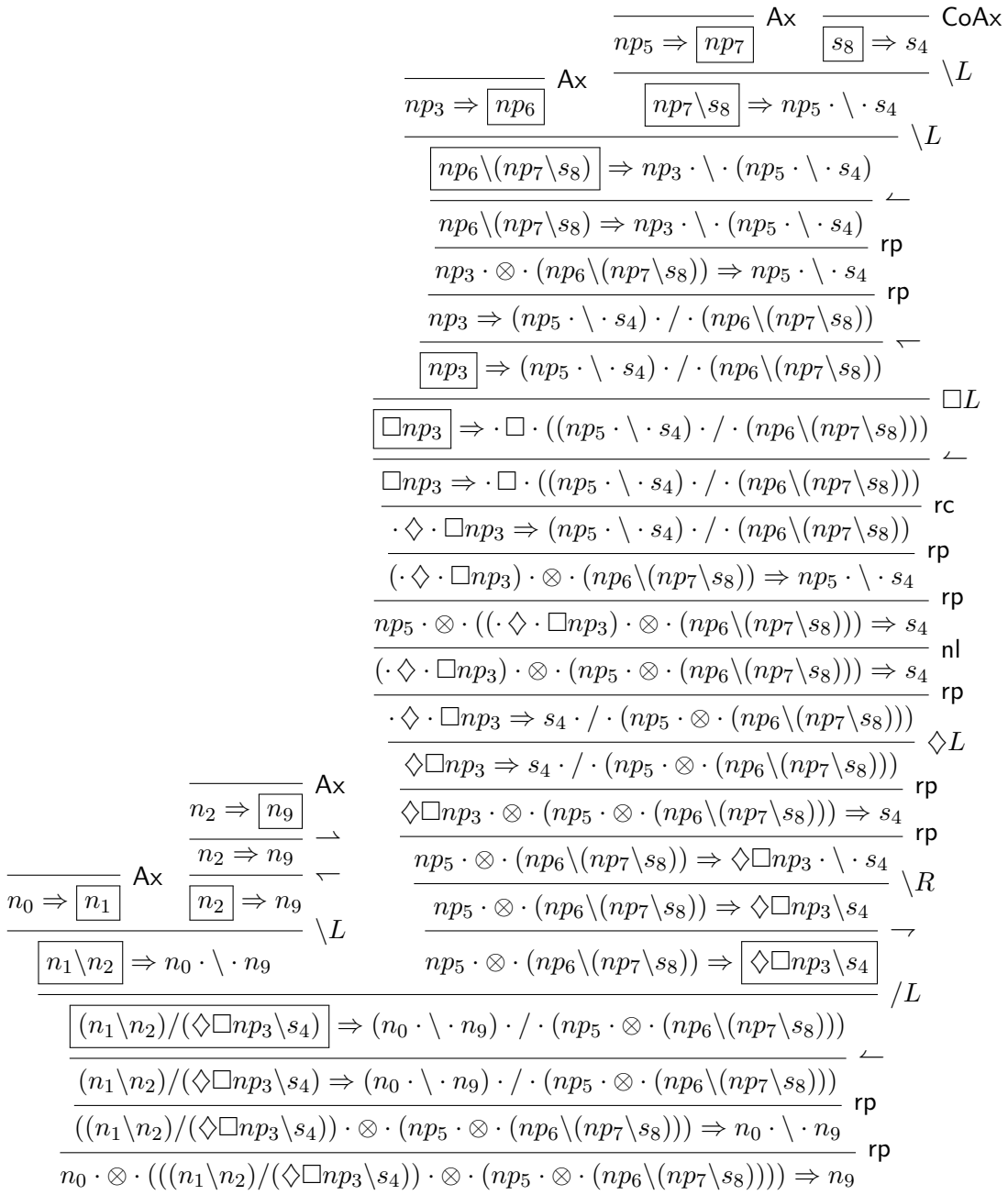

Figure A.6: Derivation for "lakei die Alice plaagt" $\rightsquigarrow n$, with the polarities np, n: + and s: -, which translates to "lackey whom Alice teases"

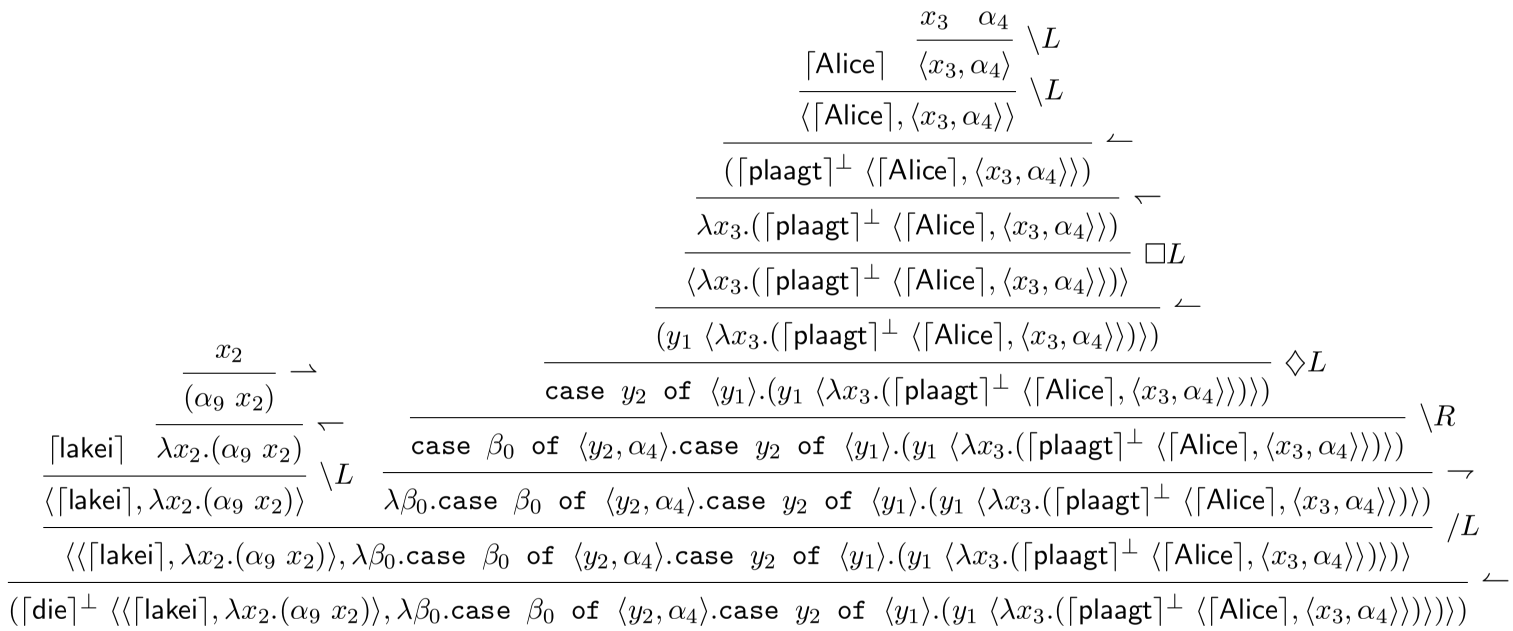

Figure A.7: Formation of the proof term for "lakei die Alice plaagt" $\rightsquigarrow n$

$$n_0 \Rightarrow X_0$$

(a)

$$np_5 \Rightarrow X_5$$

(b)

$$\dfrac{\dfrac{}{n_? \Rightarrow \boxed{n_9}}\,\mathsf{Ax}}{n_? \Rightarrow n_9}\,\rightharpoonup$$

(c)

$$\dfrac{\dfrac{\dfrac{}{n_? \Rightarrow \boxed{n_1}}\,\mathsf{Ax} \quad \dfrac{n_2 \Rightarrow X_1}{\boxed{n_2} \Rightarrow X_1}\,\leftharpoonup}{\boxed{n_1\backslash n_2} \Rightarrow n_? \cdot \backslash \cdot X_1}\,\backslash L \quad \dfrac{\dfrac{X_2 \Rightarrow (\diamondsuit\Box np_3) \cdot \backslash \cdot s_4}{X_2 \Rightarrow (\diamondsuit\Box np_3)\backslash s_4}\,\backslash R}{X_2 \Rightarrow \boxed{(\diamondsuit\Box np_3)\backslash s_4}}\,\rightharpoonup}{\boxed{(n_1\backslash n_2)/((\diamondsuit\Box np_3)\backslash s_4)} \Rightarrow (n_? \cdot \backslash \cdot X_1) \cdot / \cdot X_2}\,/L}{(n_1\backslash n_2)/((\diamondsuit\Box np_3)\backslash s_4) \Rightarrow (n_? \cdot \backslash \cdot X_1) \cdot / \cdot X_2}\,\leftharpoonup$$

(d)

$$\dfrac{\cdot \diamondsuit \cdot \Box np_3 \Rightarrow X_3}{\diamondsuit\Box np_3 \Rightarrow X_3}\,\diamondsuit L$$

(e)

$$\dfrac{\dfrac{\dfrac{np_3 \Rightarrow X_4}{\boxed{np_3} \Rightarrow X_4}\,\leftharpoonup}{\boxed{\Box np_3} \Rightarrow \cdot \Box \cdot X_4}\,\Box L}{\Box np_3 \Rightarrow \cdot \Box \cdot X_4}\,\leftharpoonup$$

(f)

$$\dfrac{\dfrac{\dfrac{}{np_? \Rightarrow \boxed{np_6}}\,\mathsf{Ax} \quad \dfrac{\dfrac{}{np_? \Rightarrow \boxed{np_7}}\,\mathsf{Ax} \quad \dfrac{}{\boxed{s_8} \Rightarrow s_?}\,\mathsf{CoAx}}{\boxed{np_7\backslash s_8} \Rightarrow np_? \cdot \backslash \cdot s_?}\,\backslash L}{\boxed{np_6\backslash(np_7\backslash s_8)} \Rightarrow np_? \cdot \backslash \cdot (np_? \cdot \backslash \cdot s_?)}\,\backslash L}{np_6\backslash(np_7\backslash s_8) \Rightarrow np_? \cdot \backslash \cdot (np_? \cdot \backslash \cdot s_?)}\,\leftharpoonup$$

(g)

Figure A.8: PPTs for pre-sequent 2.4, where PPT (a) is unfolded from the type formula n_0, (b) from np_5, (c) from n_9, (d), (e) and (f) from $(n_1\backslash n_2)/((\diamondsuit\Box np_3)\backslash s_4)$ and (g) from $np_6\backslash(np_7\backslash s_8)$

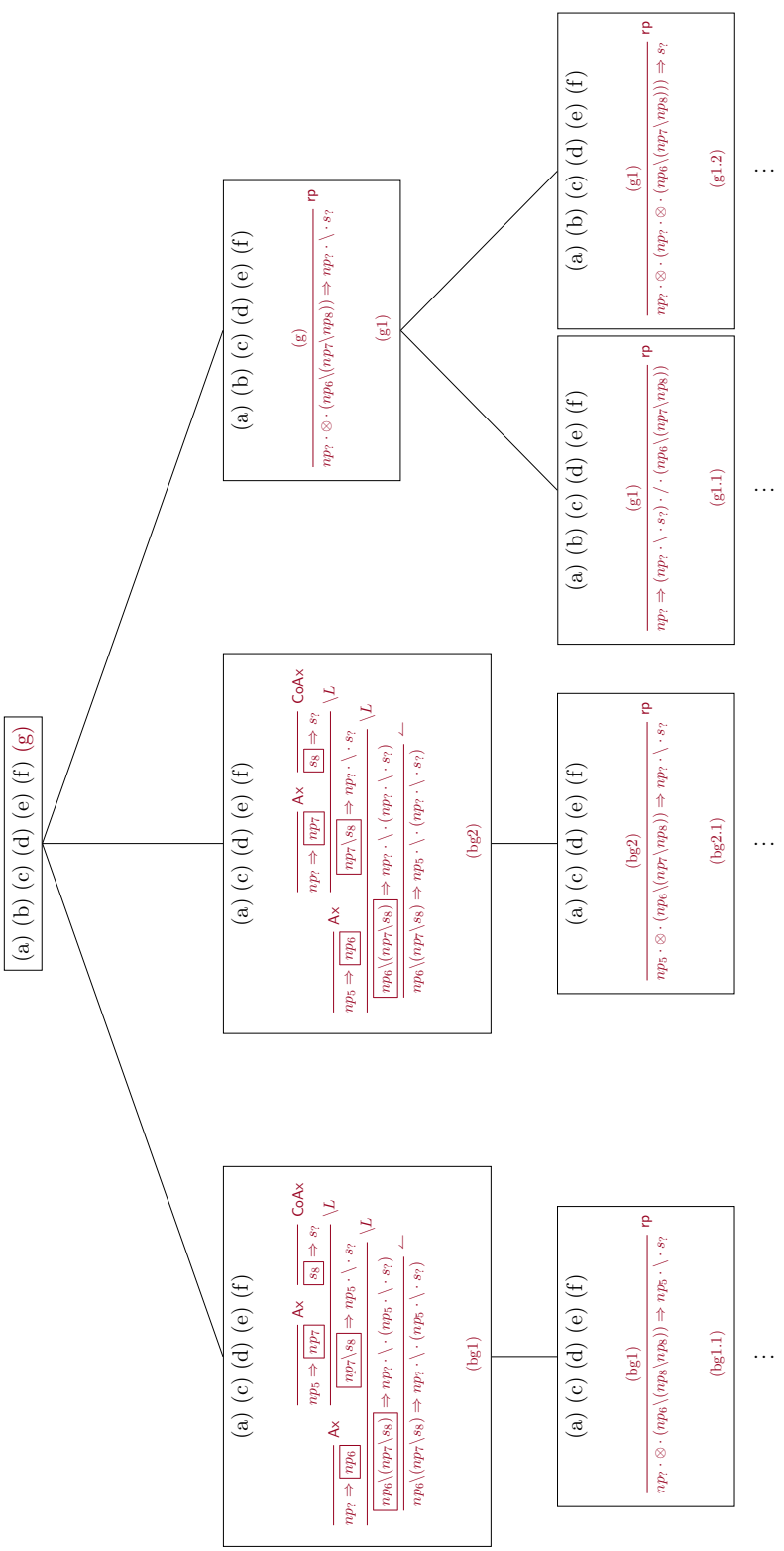

Figure A.9: First few depths of the search tree for pre-sequent 2.4. The letters (a), (b), (c), (d), (e), (f) and (g) represent the PPTs in Figure A.8. The main proof in a node is indicated with a color.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow \boxed{np_2}}\,\mathsf{Ax} \quad \dfrac{}{\boxed{s_3} \Rightarrow s_1}\,\mathsf{CoAx}}{\boxed{np_2\backslash s_3} \Rightarrow np_0 \cdot \backslash \cdot s_1}\,\backslash L}{np_2\backslash s_3 \Rightarrow np_0 \cdot \backslash \cdot s_1}\,\leftharpoondown}{np_0 \cdot \otimes \cdot (np_2\backslash s_3) \Rightarrow s_1}\,\mathsf{rp}}{np_0 \Rightarrow s_1 \cdot / \cdot (np_2\backslash s_3)}\,\mathsf{rp}}{np_0 \Rightarrow s_1/(np_2\backslash s_3)}\,/R$$

$$\texttt{case}\ \beta_0\ \texttt{of}\ \langle \alpha_1, y_0\rangle.(y_0\ \langle x_0, \alpha_1\rangle)$$

Figure A.10: Program output for the query `get_derivations('types', None, 'np', 's/(np\s)', {'np': '+', 's': '-'}, 0, True)`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow \boxed{np_1}}\,\mathsf{Ax} \quad \dfrac{}{\boxed{s_2} \Rightarrow s_3}\,\mathsf{CoAx}}{\boxed{np_1\backslash s_2} \Rightarrow np_0 \cdot \backslash \cdot s_3}\,\backslash L}{np_1\backslash s_2 \Rightarrow np_0 \cdot \backslash \cdot s_3}\,\leftharpoondown}{np_0 \cdot \otimes \cdot (np_1\backslash s_2) \Rightarrow s_3}\,\mathsf{rp}}{np_0 \cdot \otimes \cdot (np_1\backslash s_2) \Rightarrow \boxed{s_3}}\,\rightharpoonup$$

$$\lambda\alpha_3.(y_0\ \langle x_0, \alpha_3\rangle)$$

Figure A.11: Program output for the query `get_derivations('types', None, 'np, (np\s)', 's', {'np': '+', 's': '-'}, 'r', True)`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow \boxed{np_1}}\,\mathsf{Ax}}{\cdot \Diamond \cdot np_0 \Rightarrow \boxed{\Diamond np_1}}\,\Diamond R}{\cdot \Diamond \cdot np_0 \Rightarrow \Diamond np_1}\,\rightharpoonup}{np_0 \Rightarrow \cdot \Box \cdot (\Diamond np_1)}\,\mathsf{rc}}{np_0 \Rightarrow \Box(\Diamond np_1)}\,\Box R}{\boxed{np_0} \Rightarrow \Box(\Diamond np_1)}\,\leftharpoondown$$

$$\lambda x_0.\texttt{case}\ \beta_1\ \texttt{of}\ \langle \beta_0\rangle.(\beta_0\ \langle x_0\rangle)$$

Figure A.12: Program output for the query `get_derivations('types', None, 'np', '::(<>np)', {'np': '+'}, 'l', True)`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_0 \Rightarrow \boxed{np_1}}\,\mathsf{Ax}}{np_0 \Rightarrow np_1}\,\rightharpoonup}{\boxed{np_0} \Rightarrow np_1}\,\leftharpoonup}{\boxed{\Box np_0} \Rightarrow \cdot \Box \cdot np_1}\,\Box L}{\Box np_0 \Rightarrow \cdot \Box \cdot np_1}\,\leftharpoonup}{\cdot \Diamond \cdot (\Box np_0) \Rightarrow np_1}\,\mathsf{rc}}{\Diamond(\Box np_0) \Rightarrow np_1}\,\Diamond L$$

$$\texttt{case}\ y_1\ \texttt{of}\ \langle y_0 \rangle.(y_0\ \langle \lambda x_0.(\alpha_1\ x_0) \rangle)$$

Figure A.13: Program output for the query `get_derivations('types', None, '<>(::np)', 'np', {'np': '+'}, 0, True)`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n_2 \Rightarrow \boxed{n_3}}\,\mathsf{Ax} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n_0 \Rightarrow \boxed{n_5}}\,\mathsf{Ax}}{n_0 \Rightarrow n_5}\,\rightharpoonup}{\boxed{n_0} \Rightarrow n_5}\,\leftharpoonup \quad \dfrac{}{n_4 \Rightarrow \boxed{n_1}}\,\mathsf{Ax}}{\boxed{n_0/n_1} \Rightarrow n_5 \cdot / \cdot n_4}\,/L}{n_0/n_1 \Rightarrow n_5 \cdot / \cdot n_4}\,\leftharpoonup}{(n_0/n_1) \cdot \otimes \cdot n_4 \Rightarrow n_5}\,\mathsf{rp}}{n_4 \Rightarrow (n_0/n_1) \cdot \backslash \cdot n_5}\,\mathsf{rp} \quad \dfrac{}{\boxed{n_4} \Rightarrow (n_0/n_1) \cdot \backslash \cdot n_5}\,\leftharpoonup}{\boxed{n_3 \backslash n_4} \Rightarrow n_2 \cdot \backslash \cdot ((n_0/n_1) \cdot \backslash \cdot n_5)}\,\backslash L}{n_3 \backslash n_4 \Rightarrow n_2 \cdot \backslash \cdot ((n_0/n_1) \cdot \backslash \cdot n_5)}\,\leftharpoonup}{n_2 \cdot \otimes \cdot (n_3 \backslash n_4) \Rightarrow (n_0/n_1) \cdot \backslash \cdot n_5}\,\mathsf{rp}}{(n_0/n_1) \cdot \otimes \cdot (n_2 \cdot \otimes \cdot (n_3 \backslash n_4)) \Rightarrow n_5}\,\mathsf{rp}$$

$$(y_1\ \langle x_2, \lambda x_4.(y_0\ \langle \lambda x_0.(\alpha_5\ x_0), x_4 \rangle) \rangle)$$

Figure A.14: Program output for the query `get_derivations('types', None, r'(n/n)', n, (n\n)', 'n', {'n': '+'}, 0, True)` (1/2)

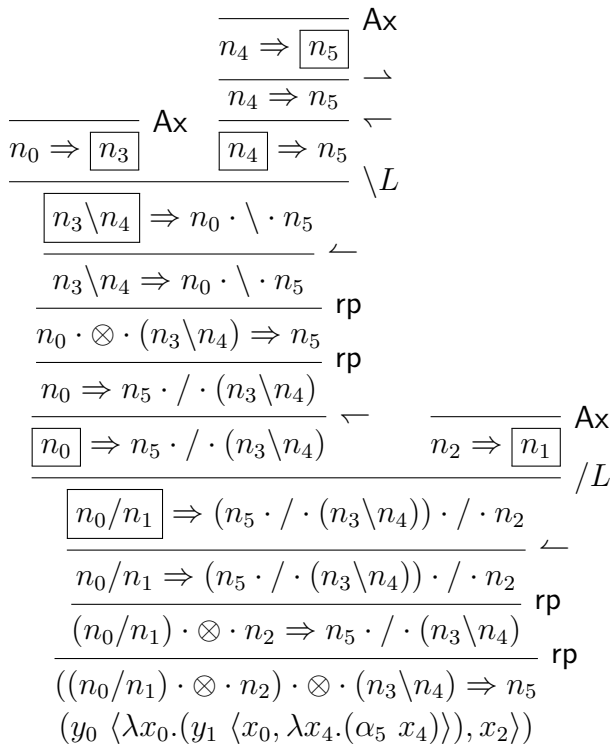

Figure A.14: Program output for the query `get_derivations('types', None, r'(n/n), n, (n\n)', 'n', {'n': '+'}, 0, True)` (2/2)

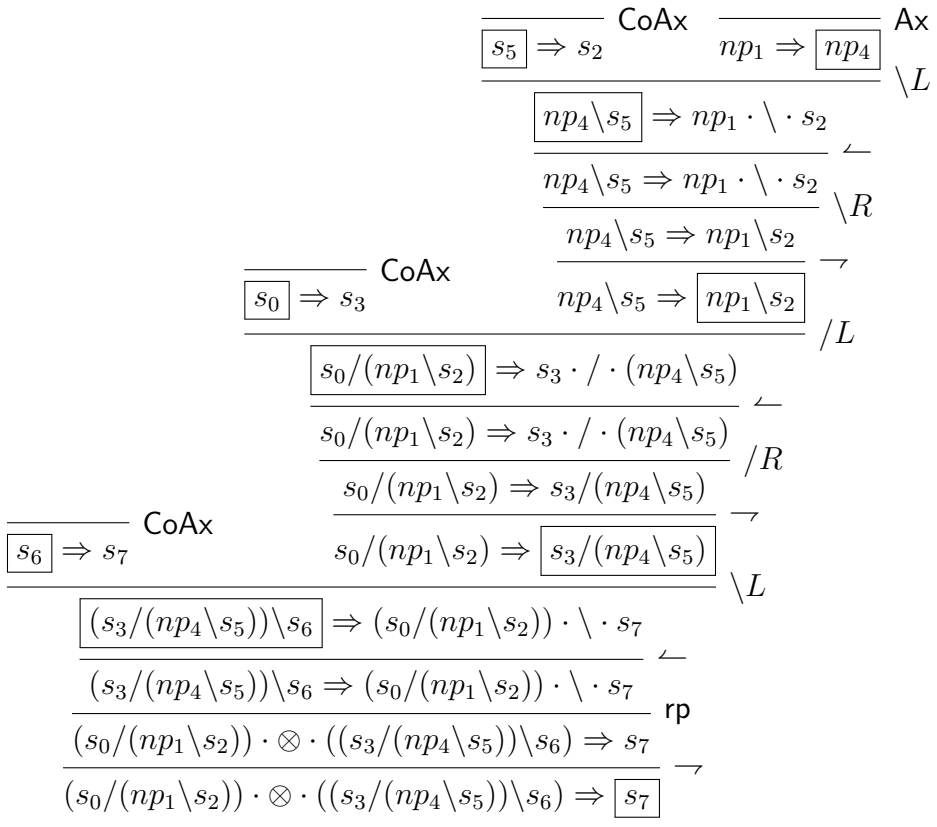

Figure A.15: Program output for the query `get_derivations('types', None, '(s/(np\s)), ((s/(np\s))\s)', 's', {'np': '+', 's': '-'}, 'r', True)` (1/2)

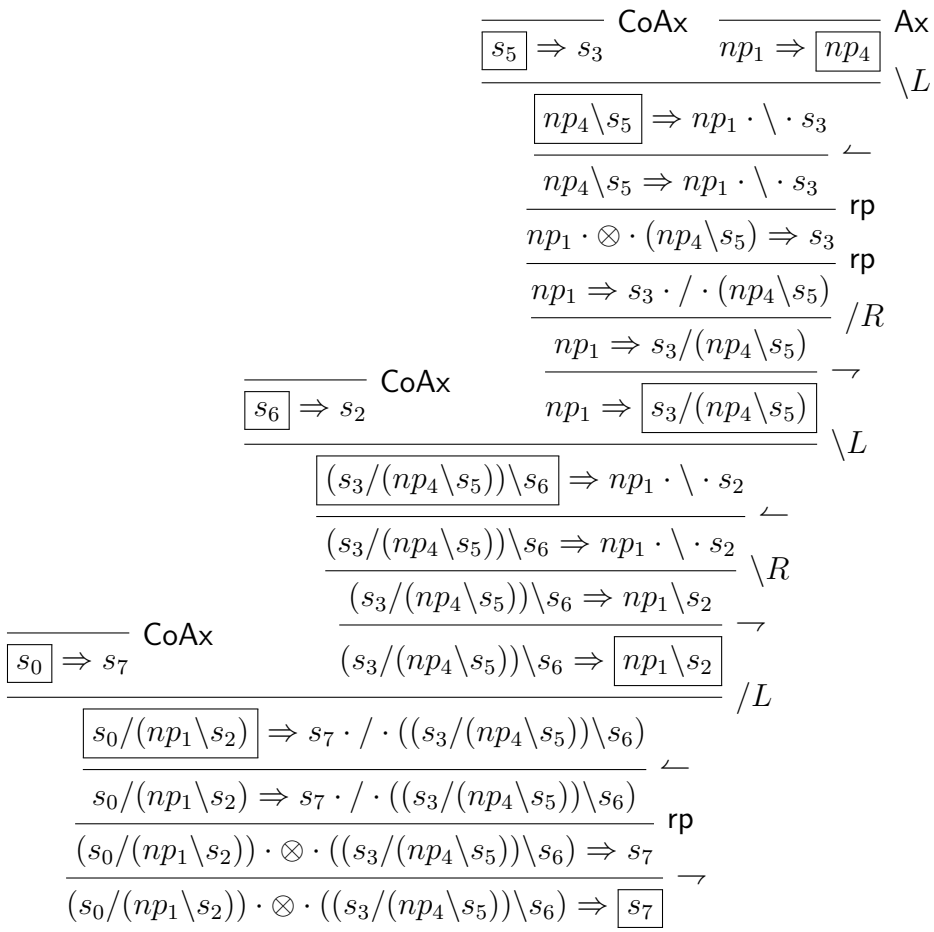

Figure A.15: Program output for the query `get_derivations('types', None, '(s/(np\s)), ((s/(np\s))\s)', 's', {'np': '+', 's': '-'}, 'r', True)` (2/2)

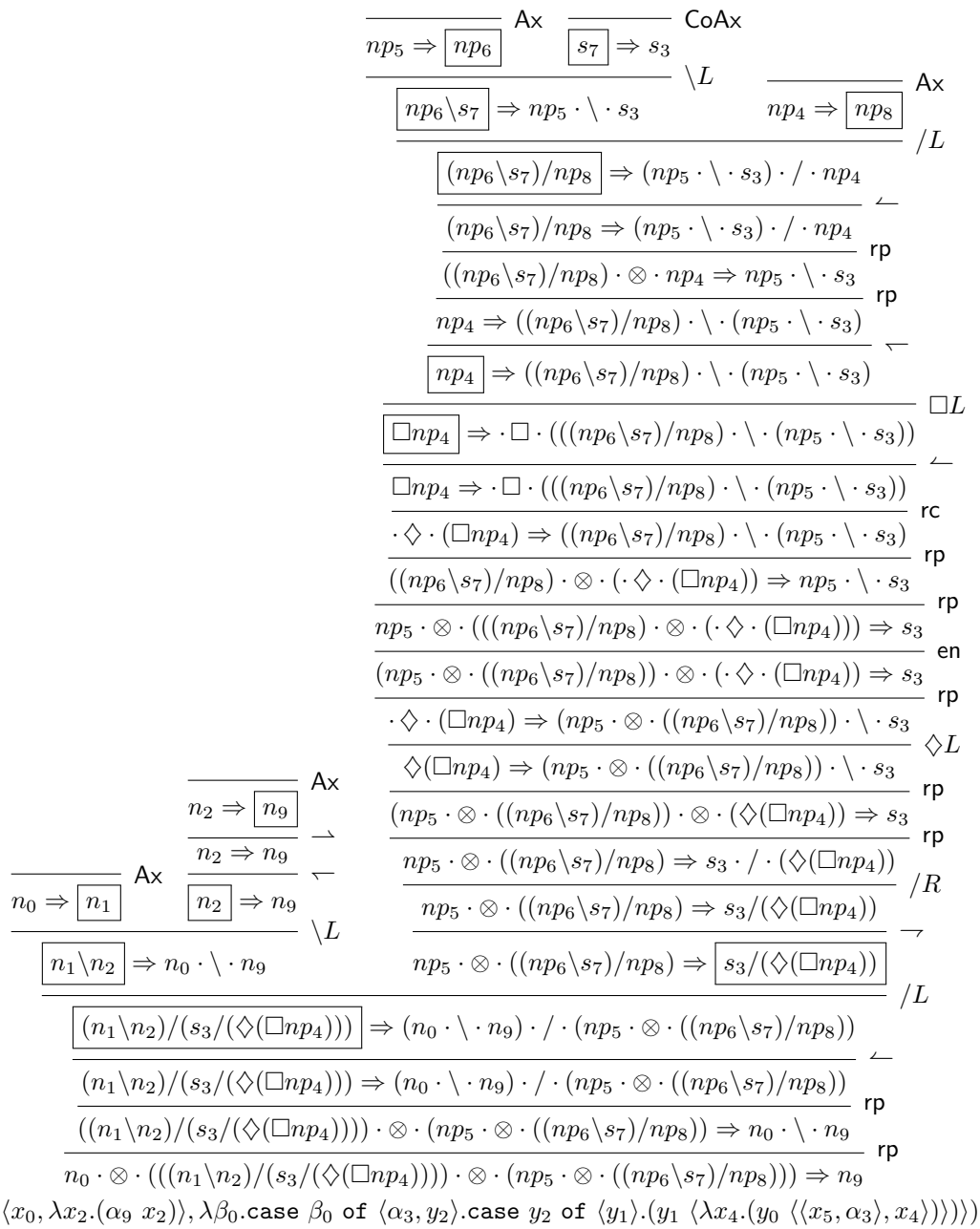

Figure A.16: Program output for the query `get_derivations('words', 'English', 'book that alice read', 'n', {'np': '+', 'n': '+', 's': '-'}, 0, True)`

Appendix B

Theorem Prover

This appendix contains a copy of the written code. The usable code files can be found here.

README.txt

```
This project was made as part of a Bachelor thesis titled "A forward
    chaining theorem prover for the extended Lambek calculus". It was
    written by Emmelie Slotboom under supervision of prof. dr. Michael
    Moortgat at Utrecht University.

The current version of the program is 1911.03-01.


COMMENTS OF NOTE
It is recommended that the thesis is read before using the program to
    familiarize oneself with the theoretic background of Lambek calculus
     and theorem proving.


DESCRIPTION
The code in this project automatically parses and proves a given
    theorem in extended Lambek calculus. It generates every possible
    derivation and proof term as a LaTeX and PDF file.


INSTALLATION
This project was made using Python 3.6.1. It can be installed here:
    https://www.python.org/downloads/. After the installation is
    complete, it should be possible to open and use IDLE, which is all
    that is required to run the program.
```

```
USAGE
---------------------------------
How to run an example input
---------------------------------

1. Open IDLE,
2. click File > Open... > main.py,
3. in main.py click Run > Run Module,
4. in IDLE click File > Open... > examples.py,
4. copy and paste an example input from examples.py in IDLE,
5. press enter to run the example.

---------------------------------
How to run a non-example input
---------------------------------

For the process of opening and setting up IDLE, see 'How to run an
   example input'. A theorem is proven using the function '
   get_derivations', which takes 7 arguments. An overview of what these
    arguments should look like can be found below. Each argument is
   presented with an example value.

input_method = 'types'
    whether the input antecedent consists of types or words

    should be 'types' or 'words'

language = 'English'
    the languge in which the input antecedent should be interpreted
    this argument is also used to decide which lexicon to use
    for more explanation of the lexicons, see 'How to add a word in a
       lexicon'.

    should be 'Dutch', 'English' or None
    may only be None if 'input_method' is 'types' and the input does
       not make use of controlled associaticty or commutativity

antecedent = r"n,((n\n)/(s/(<>(::np)))),np,((np\s)/np)"
    the combination of antecedent types or words in the theorem

    if 'input_method' is 'types':
        should consist of one or more formulas where
        formulas should be separated by commas,
```

```
        atoms should be of the form 'p' and
        complex formulas should be of the form '(#A)' or '(A$B)' where
            'A' and 'B' are formulas,
            '#' is '<>', '<.>', '::' or ':.:' and
            '$' is '/', '\' or '*'
        if there is only one complex formula in the antecedent or
           consequent the outer parentheses may be omitted
    if 'input_method' is 'words':
        should consist of a subset of the words in the lexion with the
           language 'language'
        may not be empty

consequent = r"n"
    the goal type for the antecedent

    the requirements of 'antecedent' apply, except
        should always be one formula
        may not be a word

polarities = {'np': '+', 'n': '+', 's': '-'}
    the polarities of the atomic types in the antecedent and consequent

    should be a dictionary that contains every atomic type in '
       antecedent' and 'consequent' in string form as keys
    every type should be given polarity '"+"' or '"-"' as the value

focus = 0
    whether the antecedent, consequent or neither is focused

    if 'antecedent' is focused:
        should be '"l"'
    if 'consequent' is focused:
        should be '"r"'
    if neither is focused:
        should be '0'

show_indices = True
    whether the indices of the atomic types in the antecedent and
       consequent should be shown in the derivation(s)

    if the indices of the atomic types should be shown in the
       derivation(s):
        should be 'True'
```

```
    else:
        should be 'False'
```

How to add a word in a lexicon

A lexicon is a .txt file that lists the words that may be used in an input, combined with their types. It is very important to preserve the syntax of the lexicon, otherwise the code may not be able to compile it. See dutch_lexicon.txt and english_lexicon.txt for examples.

A word–type combination in a lexicon is presented on one line with the word first, immediately followed by a colon and space, and ending in the type. A type should not contain any outer parentheses, but every other complex formula must be enclosed in parentheses. A type should not contain any spaces.

Every new word–type combination should be on a new line. Additional newlines may be used between word–type combinations to structure word groups; this has no effect on how the lexicon is compiled.

SUPPORT

For any questions or comments, send an e–mail to emmelieslotboom@live.nl.

PROJECT STATUS

The original author of the project does not actively maintain it. This means that it may not run properly on newer python versions.

dutch_lexicon.txt

```
alice: np
bob: np

de: np/n
het: np/n
dit: np/n

dat: (n\n)/((<>(::np))\s)
die: (n\n)/((<>(::np))\s)
```

```
slimme: n/n

lakei: n
resultaat: n
student: n

plaagt: np\(np\s)
rekent: pp\(np\s)

niet: (np\s)/(np\s)

er: pp/(pp/np)
op: pp/np
waar: (n\n)/((<>(::(pp/(pp/np))))\s)
```

english_lexicon.txt

```
alice: np
bob: np

anyone: s/(np\s)
everyone: s/(np\s)
nobody: (s/np)\s
nothing: (s/np)\s
someone: s/(np\s)

himself: ((np\s)/np)\(np\s)

the: np/n
a: np/n
some: np/n

delicious: n/n
smart: n/n

book: n
cake: n
student: n
window: n

left: np\s
```

```
likes: (np\s)/np
read: (np\s)/np

never: (np\s)/(np\s)

closing: gp/np

that: (n\n)/(s/(<>(::np)))

without: (:.:((np\s)\(np\s)))/gp
```

Lexicon.py

```
"""
This module contains a function that gives the type for every word in
    the lexicon.
"""
from Exception import InputError, LanguageError


def lexicon(language):
    """Returns a dictionary of the existing lexicon that belongs to the
        requested language.

    Parameters
    ----------
    language : str

    Returns
    -------
    dict of {str : str}
        The existing lexicon that belongs to the requested language

    Raises
    ------
    InputError
        If no language is given
    LanguageError
        If a requested language is not implemented
    """
    if language == 'English':
        lex_txt = open('english_lexicon.txt', 'r').read()
    elif language == 'Dutch':
```

```
29          lex_txt = open('dutch_lexicon.txt', 'r').read()
30      elif not language:
31          raise InputError("Please provide a language in the input.")
32      else:
33          raise LanguageError(language)
34
35      lex = {}
36      lex_list = lex_txt.split('\n')
37      for word_with_type in lex_list:
38          if word_with_type:  # ignore excessive newlines in the text
                file
39              key_with_value = word_with_type.split(': ')
40              if key_with_value[1].isalpha():  # the type is atomic
41                  lex[key_with_value[0]] = key_with_value[1]
42              else:  # the type is complex
43                  lex[key_with_value[0]] = f'({key_with_value[1]})'  #
                       put the type between parantheses
44
45      return lex
```

Formula.py

```
1  from Exception import FormulaTypeError
2
3  from sys import maxsize
4
5
6  class Formula:
7      """
8      This class represents a formula.
9
10     Attributes
11     ----------
12     type : str or list
13         Atom types are strings, complex formula types are a list of the
               form [A, $, B] or [#, A], where $ and # are
14         strings representing connectives and A and B are Formulas.
15     polarity : str
16         Formulas can have either a positive or negative polarity,
               represented as '+' or '-'.
17     index : list of int
18         All indices of the atoms that appear within the Formula
19     position : str
```

```
20          The Formula should be either an 'input' or an 'output'.
21
22      Methods
23      -------
24      __eq__(other) : bool
25          Whether two Formulas are identical
26      __hash__() : int
27          The hash value of the Formula
28      __repr__() : str
29          An understandable representation of the Formula
30      __str__() : str
31          A readable representation of the Formula
32      is_atom() : bool
33          Whether the Formula is an atom
34      is_complex() : bool
35          Whether the Formula is complex
36      is_unary() : bool
37          Whether the Formula is unary
38      is_binary() : bool
39          Whether the Formula is binary
40      is_incomplete() : bool
41          Whether the Formula contains unknown indices
42      """
43      def __init__(self, type, polarity, index, position):
44          self.type = type
45          self.polarity = polarity
46          self.index = index
47          self.position = position
48
49      def __eq__(self, other):
50          """Returns whether the attributes of two Formulas are equal.
51
52          Overrides the default method.
53
54          Parameters
55          ----------
56          other : Formula
57              The Formula that is being compared to self
58
59          Returns
60          -------
61          bool
62              Whether the two Formulas are identical
```

```
63
64          Raises
65          ------
66          NotImplemented
67              If the parameter is not a Formula
68          """
69          if isinstance(other, Formula):
70              if self.index == other.index:
71                  if self.is_unary() and other.is_unary():  # the
                        Formulas are of the forms #A and $B
72                      return (self.type[0] == other.type[0] and  #
                            whether the connectives match
73                              self.type[1] == other.type[1])  # whether
                                Formulas A and B match
74                  elif self.is_binary() and other.is_binary():  # the
                        Formulas are of the forms A#B and C$D
75                      return (self.type[0] == other.type[0] and  #
                            whether Formulas A and C match
76                              self.type[1] == other.type[1] and  #
                                whether the connectives match
77                              self.type[2] == other.type[2])  # whether
                                Formulas B and D match
78                  elif self.is_atom() and other.is_atom():
79                      return self.type == other.type
80
81              return False
82          else:
83              return NotImplemented
84
85      def __hash__(self):
86          """Returns the hash of the Formula.
87
88          Overrides the default method.
89
90          Returns
91          -------
92          int
93              The hash of the Formula
94          """
95          return hash(str(self))
96
97      def __repr__(self):
98          """Returns an understandable representation of the Formula.
```

```
99
100         This function returns the necessary information needed for
                debugging. Overrides the default method.
101
102         Returns
103         -------
104         str
105             An understandable representation of the Formula
106         """
107         return f"Formula({self.type!r}, {self.polarity}, {self.index},
                {self.position})"
108
109     def __str__(self):
110         """Returns a readable representation of the Formula.
111
112         Overrides the default method.
113
114         Returns
115         -------
116         str
117             A readable representation of the Formula
118
119         Raises
120         ------
121         FormulaTypeError
122             If the type of the Formula is not of the form a, [#, A] or
                    [A, $, B]
123
124         Examples
125         --------
126         >>> s = Formula('s', '-', [0], 'input')
127         >>> np = Formula('np', '+', [1], 'input')
128         >>> str(Formula([s, '/', np], '-', [0, 1], 'input'))
129         '(s0/np1)'
130         """
131         if self.is_atom():
132             if self.index == [maxsize]:
133                 summary = self.type + '?'  # an unknown Formula's index
                        is represented as '?' (instead of 2147483647)
134             else:
135                 summary = self.type + str(self.index[0])
136         elif self.is_unary():
137             a = str(self.type[1])
```

```
138             summary = f"({self.type[0]}{a})"
139         elif self.is_binary():
140             a = str(self.type[0])
141             b = str(self.type[2])
142             summary = f"({a}{self.type[1]}{b})"
143         else:
144             raise FormulaTypeError(self)
145 
146         return summary
147 
148     def is_atom(self):
149         """Returns whether the Formula is atomic.
150 
151         A Formula is atomic if its type is a string. Note that this
               function also returns True when the type is '?'.
152         This is intentional.
153 
154         Returns
155         -------
156         bool
157             Whether the Formula is atomic
158         """
159         return isinstance(self.type, str)
160 
161     def is_complex(self):
162         """Returns whether the Formula is complex.
163 
164         A Formula is complex if its type is a list.
165 
166         Returns
167         -------
168         bool
169             Whether the Formula is complex
170         """
171         return isinstance(self.type, list)
172 
173     def is_unary(self):
174         """Returns whether the Formula is unary.
175 
176         A Formula is unary if its type is a list of length 2.
177 
178         Returns
179         -------
```

```
180         bool
181             Whether the Formula is unary
182         """
183         return isinstance(self.type, list) and len(self.type) == 2
184 
185     def is_binary(self):
186         """Returns whether the Formula is binary.
187 
188         A Formula is binary if its type is a list of length 3.
189 
190         Returns
191         -------
192         bool
193             Whether the Formula is binary
194         """
195         return isinstance(self.type, list) and len(self.type) == 3
196 
197     def is_incomplete(self):
198         """Returns whether the Formula contains any indices that are
                not filled yet.
199 
200         An unfilled index has the value maxsize: 2147483647.
201 
202         Returns
203         -------
204         bool
205             Whether the formula contains any indices that are not
                    filled yet
206         """
207         return maxsize in self.index
```

Sequent.py

```
1  class Sequent:
2      """
3      This class represents a sequent.
4 
5      Attributes
6      ----------
7      antecedent : list of Formula or None
8          A list of formulas in the antecedent, optional
9      consequent : Formula or None
10         The formula in the consequent, optional
```

```
11      focus : int or str
12          A neutral Sequent has focus 0, a Sequent with a focused
               antecedent has focus "l", and a Sequent with a focused
13          consequent has focus "r", optional
14
15      Methods
16      -------
17      __eq__(other) : bool
18          Whether two Sequents are identical
19      __hash__() : int
20          The hash value of the Sequent
21      __repr__() : str
22          An understandable represenation of the Sequent
23      __str__() : str
24          A readable representation of the Sequent
25      """
26      def __init__(self, antecedent=None, consequent=None, focus=0, term=
            ''):
27          self.antecedent = antecedent
28          self.consequent = consequent
29          self.focus = focus
30          self.term = term
31
32      def __eq__(self, other):
33          """Returns whether the attributes of the antecedents and
               consequents as well as the focus of two Sequents are
34          equal.
35
36          Overrides the default method.
37
38          Parameters
39          ----------
40          other : Sequent
41              The Sequent that is being compared to self
42
43          Returns
44          -------
45          bool
46              Whether the two Sequents are identical
47
48          Raises
49          ------
50          NotImplemented
```

```
51              If the parameter is not a Sequent
52          """
53          if isinstance(other, Sequent):
54              return (self.antecedent == other.antecedent and
55                      self.consequent == other.consequent and
56                      self.focus == other.focus)
57          else:
58              return NotImplemented
59
60      def __hash__(self):
61          """Returns the hash of the Sequent.
62
63          Overrides the default method.
64
65          Returns
66          -------
67          int
68              The hash of the Sequent
69          """
70          return hash((self.antecedent, self.consequent, self.focus))
71
72      def __repr__(self):
73          """Returns an understandable representation of the Sequent.
74
75          This function returns the necessary information needed for
               debugging. Overrides the default method.
76
77          Returns
78          -------
79          str
80              An understandable representation of the Sequent
81          """
82          return f"Sequent({self.antecedent!r}, {self.consequent!r}, {
               self.focus})"
83
84      def __str__(self):
85          """Returns a readable representation of the Sequent.
86
87          Formats the antecedent and consequent as in the class Formula.
               If a side is focused, it is put between square
88          brackets. Overrides the default method.
89
90          Returns
```

```
91          ---------
92          str
93              A readable representation of the Sequent
94
95          See Also
96          ---------
97          Formula.__str__() : A readable representation of a Formula
98
99          Examples
100         ---------
101         >>> from Formula import Formula
102         >>> antecedent = Formula('np', '+', [0], 'input')
103         >>> consequent = Formula('np', '+', [1], 'output')
104         >>> str(Sequent(antecedent, consequent, 'r'))
105         'np0 => [np1]'
106         """
107         antecedent = self.antecedent
108         if isinstance(antecedent, list):  # the antecedent consists of
                multiple Formulas
109             ant = ""
110             for formula in antecedent:
111                 ant += f"{str(formula)},"
112             ant = ant[:-1]  # remove the last comma
113         else:
114             ant = str(antecedent)
115         con = str(self.consequent)
116
117         if self.focus == 'l':
118             ant = f"[{ant}]"
119         elif self.focus == 'r':
120             con = f"[{con}]"
121
122         return f"{ant} => {con}"
```

Tree.py

```
1   from Sequent import Sequent
2
3
4   class Tree:
5       """
6       This class represents a tree.
7
```

```
8       Attributes
9       ----------
10      nodes : list of Sequent
11          The nodes in the Tree, optional
12      edges : list of Edge
13          The Edges in the Tree, optional
14      combined_sequents : list of tuple of (Sequent, int)
15          The overlapping node and its location in the nodes if the Tree
                was merged from two Trees.
16
17      Methods
18      -------
19      __repr__() : str
20          An understandable represenation of the Sequent
21      __str__() : str
22          A readable representation of the Sequent
23      add_node(nodes, top_down=False) : None
24          Adds a node to the existing Tree
25      add_edge(premise, conclusion, rule, top_down=False) : None
26          Adds an Edge to the existing Tree
27      root() : Sequent
28          Returns the root of the Tree
29      leafs() : list of Sequent
30          Returns the leafs of the Tree
31      axioms() : list of Sequent
32          Returns the axioms in the Tree
33      contains_loop() : bool
34          Whether the Tree contains a loop
35      """
36      def __init__(self, nodes=None, edges=None, combined_sequents=None):
37          if nodes is None:
38              nodes = []
39          if edges is None:
40              edges = []
41          if combined_sequents is None:
42              combined_sequents = []
43
44          self.nodes = nodes
45          self.edges = edges
46          self.combined_sequents = combined_sequents
47
48      def __repr__(self):
49          """Returns an understandable representation of the Tree.
```

```
50
51          Overrides the default method.
52
53          Returns
54          ———————
55          str
56              An understandable representation of the Tree
57          """
58          nodes = ""
59          for node in self.nodes:
60              nodes += f"{node}\n"
61
62          edges = ""
63          for edge in self.edges:
64              edges += f"{edge.rule}\n{edge.premise}\n{edge.conclusion}\n
                "
65
66          combined_sequents = ""
67          for sequent in self.combined_sequents:
68              combined_sequents += f"{sequent[0]} {sequent[1]}\n"
69
70          return f"{nodes}\n{edges}\n{combined_sequents}"
71
72      def __str__(self):
73          """Returns a readable representation of the Tree.
74
75          Overrides the default method.
76
77          Returns
78          ———————
79          str
80              A readable representation of the Tree
81          """
82          if not self.edges:
83              return f"{self.nodes[0]}\n"
84          else:
85              edges = ""
86              for edge in self.edges:
87                  edges += f"{edge.rule}\n{edge.premise}\n{edge.
                        conclusion}\n"
88              return edges
89
90      def add_node(self, node, top_down=False):
```

```
91          """Adds a node to the existing Tree.
92
93          Parameters
94          ----------
95          node : Sequent
96          top_down : bool
97              Whether the new node should be at the beginning or end of
                    the list of nodes
98          """
99          if top_down:
100             self.nodes.insert(0, node)
101         else:
102             self.nodes.append(node)
103
104     def add_edge(self, premise, conclusion, rule, top_down=False):
105         """Adds an Edge to the existing Tree.
106
107         Automatically adds the nodes of the Edge to self.nodes if they
                are not in it.
108
109         Parameters
110         ----------
111         premise : Sequent
112         conclusion : Sequent
113         rule : str
114         top_down : bool
115             Whether the Edge should be at the beginning or end of the
                    list of Edges
116         """
117         if top_down:
118             for node in [conclusion, premise]:
119                 if node not in self.nodes:
120                     self.add_node(node, top_down=True)
121
122             self.edges.insert(0, Edge(premise, conclusion, rule))
123         else:
124             for node in [premise, conclusion]:
125                 if node not in self.nodes:
126                     self.add_node(node)
127
128             self.edges.append(Edge(premise, conclusion, rule))
129
130     def root(self):
```

```
131         """Returns the root of the Tree.
132 
133         The root of a Tree is a node without premises.
134 
135         Returns
136         -------
137         Sequent
138             The root of the Tree
139         """
140         premises = []
141         for edge in self.edges:
142             premises.append(edge.premise)
143 
144         for node in self.nodes:
145             if node not in premises:
146                 return node
147 
148     def leafs(self):
149         """Returns the leafs of the Tree.
150 
151         A leaf of a Tree is a node without premises.
152 
153         Returns
154         -------
155         list of Sequent
156             The leafs of the Tree
157         """
158         conclusions = []
159         leafs = []
160 
161         for edge in self.edges:
162             conclusions.append(edge.conclusion)
163 
164         for node in self.nodes:
165             if node not in conclusions:
166                 leafs.append(node)
167 
168         return leafs
169 
170     def axioms(self):
171         """Returns the axioms in the Tree.
172 
173         An axiom of a Tree is a leaf where its antecedent and
```

```
            consequent are of the same type.
174
175         Returns
176         -------
177         list of Sequent
178             The axioms in the Tree
179         """
180         axioms = []
181         premises = []
182         conclusions = []
183
184         if not self.edges:
185             leaf = self.nodes[0]
186             if leaf.antecedent.type == leaf.consequent.type:
187                 axioms.append(leaf)
188         else:
189             for edge in self.edges:
190                 premises.append(edge.premise)
191                 conclusions.append(edge.conclusion)
192
193             for premise in premises:
194                 if premise not in conclusions and premise.antecedent.
                        type == premise.consequent.type:
195                     axioms.append(premise)
196
197         return axioms
198
199     def contains_loop(self):
200         """Returns whether the Tree contains a loop.
201
202         A Tree contains a loop if it does not have a root or a node
                appears more than once in self.nodes.
203
204         Returns
205         -------
206         bool
207             Whether the Tree contains a loop
208         """
209         root = self.root()
210         if not root:
211             return True
212         else:
213             other_nodes = self.nodes[:]
```

```
214                 other_nodes.remove(root)
215                 if root in other_nodes:
216                     return True
217 
218             return False
219 
220 
221 class IncompleteTree:
222     """
223     This class represents an incomplete tree. This can be used to
            represent PPTs.
224 
225     Attributes
226     ----------
227     conclusion: Sequent
228     """
229     def __init__(self, conclusion):
230         self.conclusion = conclusion
231 
232 
233 class NullaryTree:
234     """
235     This class represents a tree without any premises.
236 
237     Attributes
238     ----------
239     rule: str
240     conclusion: Sequent
241     """
242     def __init__(self, rule, conclusion):
243         self.rule = rule
244         self.conclusion = conclusion
245 
246 
247 class UnaryTree:
248     """
249     This class represents a tree with one premise.
250 
251     Attributes
252     ----------
253     rule: str
254     conclusion: Sequent
255     premise: NullaryTree or UnaryTree or BinaryTree
```

```
256     """
257     def __init__(self, rule, conclusion, premise):
258         self.rule = rule
259         self.conclusion = conclusion
260         self.premise = premise
261 
262 
263 class BinaryTree:
264     """
265     This class represents a tree with two premises.
266 
267     Attributes
268     ----------
269     rule: str
270     conclusion: Sequent
271     left_premise: NullaryTree or UnaryTree or BinaryTree
272     right_premise: NullaryTree or UnaryTree or BinaryTree
273     """
274     def __init__(self, rule, conclusion, left_premise, right_premise):
275         self.rule = rule
276         self.conclusion = conclusion
277         self.left_premise = left_premise
278         self.right_premise = right_premise
279 
280 
281 def to_x_ary_tree(root, tree):
282     """Converts a tree in Tree form to a tree in XaryTree form.
283 
284     Parameters
285     ----------
286     root : Sequent
287     tree : Tree
288 
289     Returns
290     -------
291     NullaryTree or UnaryTree or BinaryTree
292         The tree in XaryTree form
293     """
294     if root in tree.axioms():
295         if root.focus == 'r':
296             rule = 'Ax'
297         else:
298             rule = 'CoAx'
```

```
299             return NullaryTree(rule, root)
300         elif root in tree.leafs():
301             return IncompleteTree(root)
302         else:
303             (rules, premises) = tuple(zip(*[(edge.rule, edge.premise) for
                    edge in tree.edges if edge.conclusion == root]))
304             if len(premises) == 1:
305                 return UnaryTree(rules[0], root, to_x_ary_tree(premises[0],
                        tree))
306             elif len(premises) == 2:
307                 return BinaryTree(rules[0], root,
308                                   to_x_ary_tree(premises[1], tree),
309                                   to_x_ary_tree(premises[0], tree))
310
311
312 class Edge:
313     """This class represents an edge in a Tree.
314
315     Attributes
316     ----------
317     premise : Sequent
318         The node closest to a leaf in the Tree
319     conclusion : Sequent
320         The node closest to the root of the Tree
321     rule : str
322         The rule that was applied
323     """
324     def __init__(self, premise, conclusion, rule):
325         self.premise = premise
326         self.conclusion = conclusion
327         self.rule = rule
```

Rules.py

```
1  """
2  This module contains functions related to the application of rules.
3
4  Methods
5  -------
6  correct_connective(formula, connective) : bool
7      Whether a given Formula has the requested connective
8  apply_structural_rule(rule, sequent) : Sequent or None
9      Applies the requested structural rule to the given Sequent.
```

```
10 apply_rule(rule, formula, other_formula) : list of Sequent
11     Applies the requested structural rule to the given couple of
          Formulas.
12 """
13 from Exception import RuleError
14 from Formula import Formula
15 from Sequent import Sequent
16
17 from copy import deepcopy
18
19 basic_structural_rules = ["/RP", "*RP/", "*RP\\", "\RP", "RC<>", "RC<.>
       ", "RC::", "RC:.:"]
20 dutch_rules = ["NL1", "NL2"]
21 english_rules = ["EN1", "EN2"]
22
23 rule_name_as_latex_code = {  # a dictionary that translates the rule
       names to what they should be when typeset in LaTeX
24     "/RP": "\W{rp}",
25     "*RP/": "\W{rp}",
26     "*RP\\": "\W{rp}",
27     "\RP": "\W{rp}",
28     "RC<>": "\W{rc}",
29     "RC<.>": "\W{rc}",
30     "RC::": "\W{rc}",
31     "RC:.:": "\W{rc}",
32
33     "NL1": "\W{nl}",
34     "NL2": "\W{nl}",
35     "EN1": "\W{en}",
36     "EN2": "\W{en}",
37
38     "fL": "\leftharpoonup",
39     "fR": "\\rightharpoonup",
40     "dfL": "\leftharpoondown",
41     "dfR": "\\rightharpoondown",
42
43     "CoAx": "\W{CoAx}",
44     "Ax": "\W{Ax}",
45
46     "/L": "/L",
47     "/R": "/R",
48     "*L": "\otimes␣L",
49     "*R": "\otimes␣R",
```

```
50      "\R": "\\bs R",
51      "\L": "\\bs L",
52      "<>L": "\\fdia L",
53      "<>R": "\\fdia R",
54      "<.>L": "\\fdiaf L",
55      "<.>R": "\\fdiaf R",
56      "::L": "\gbox L",
57      "::R": "\gbox R",
58      ":.:L": "\\gboxf L",
59      ":.:R": "\\gboxf R"
60  }
61
62  # a dictionary that translates the connective representations to what
        they should be when typeset in LaTeX
63  connective_as_latex_code = {
64      './.': '\strs',
65      '.\\.': '\strbs',
66      '.*.': '\strtens',
67      '.<>.': '\strdia',
68      '.<.>.': '\strdiaf',
69      '.::.': '\strbox',
70      '.:.:.': '\strboxf',
71
72      '/': '/',
73      '\\': '\\bs',
74      '*': '\otimes',
75      '<>': '\\fdia',
76      '<.>': '\\fdiaf',
77      '::': '\gbox',
78      ':.:': '\\gboxf'
79  }
80
81
82  def correct_connective(formula, connective):
83      """Returns whether a given Formula has the requested connective.
84
85      Parameters
86      ----------
87      formula : Formula
88      connective : str
89
90      Returns
91      -------
```

```
92      bool
93          Whether a given Formula has the requested connective
94      """
95      if formula.is_unary() and formula.type[0] == connective:
96          return True
97      elif formula.is_binary() and formula.type[1] == connective:
98          return True
99      else:
100         return False
101
102
103 def apply_structural_rule(rule, sequent):
104     """Applies the requested structural rule to the given Sequent.
105
106     Parameters
107     ----------
108     rule : str
109         The rule to be applied
110     sequent : Sequent
111         The sequent to which te requested rule should be applied
112
113     Returns
114     -------
115     Sequent or None
116         The resulting Sequent if the rule is applicable, None otherwise
117
118     Raises
119     ------
120     RuleError
121         If the requested rule does not exist
122
123     Examples
124     --------
125     The rule "*RP/" applied to a Sequent with summary (np0.*.s1) => s2
           returns a Sequent with summary np0 => (s2./.s1).
126     """
127     new_sequent = deepcopy(sequent)
128
129     if sequent.focus == 0:
130         if rule == "/RP":
131             if correct_connective(sequent.consequent, './.'):  # X => Z
                    ./.Y
132                 new_sequent.antecedent.type = [sequent.antecedent, '.*.
```

```
                ', sequent.consequent.type[2]]
133             new_sequent.antecedent.polarity = None
134             new_sequent.antecedent.index = sequent.antecedent.index
                    + sequent.consequent.type[2].index
135
136             new_sequent.consequent = sequent.consequent.type[0]
137
138             return new_sequent  # X.*.Y => Z
139         elif rule == "*RP/":
140             if correct_connective(sequent.antecedent, '.*.'):  # X.*.Y
                    => Z
141                 new_sequent.antecedent = sequent.antecedent.type[0]
142
143                 new_sequent.consequent.type = [sequent.consequent, './.
                        ', sequent.antecedent.type[2]]
144                 new_sequent.consequent.polarity = None
145                 new_sequent.consequent.index = sequent.consequent.index
                        + sequent.antecedent.type[2].index
146
147                 return new_sequent  # X => Z./.Y
148         elif rule == "*RP\\":
149             if correct_connective(sequent.antecedent, '.*.'):  # X.*.Y
                    => Z
150                 new_sequent.antecedent = sequent.antecedent.type[2]
151
152                 new_sequent.consequent.type = [sequent.antecedent.type
                        [0], '.\.', sequent.consequent]
153                 new_sequent.consequent.polarity = None
154                 new_sequent.consequent.index = sequent.antecedent.type
                        [0].index + sequent.consequent.index
155
156                 return new_sequent  # Y => X.\.Z
157         elif rule == "\RP":
158             if correct_connective(sequent.consequent, '.\.'):  # Y => X
                    .\.Z
159                 new_sequent.antecedent.type = [sequent.consequent.type
                        [0], '.*.', sequent.antecedent]
160                 new_sequent.antecedent.polarity = None
161                 new_sequent.antecedent.index = sequent.consequent.type
                        [0].index + sequent.antecedent.index
162
163                 new_sequent.consequent = sequent.consequent.type[2]
164
```

```
165                 return new_sequent  # X.*.Y => Z
166             elif rule == "RC<>":
167                 if correct_connective(sequent.antecedent, '.<>.'):  # .<>.X
                        => Y
168                     new_sequent.antecedent = sequent.antecedent.type[1]
169 
170                     new_sequent.consequent.type = ['.::.', sequent.
                            consequent]
171                     new_sequent.consequent.polarity = None
172 
173                     return new_sequent  # X => .::.Y
174             elif rule == "RC<.>":
175                 if correct_connective(sequent.antecedent, '.<.>.'):  #
                        .<.>.X => Y
176                     new_sequent.antecedent = sequent.antecedent.type[1]
177 
178                     new_sequent.consequent.type = ['.:.:.', sequent.
                            consequent]
179                     new_sequent.consequent.polarity = None
180 
181                     return new_sequent  # X => .:.:.Y
182             elif rule == "RC::":
183                 if correct_connective(sequent.consequent, '.::.'):  # X =>
                        .::.Y
184                     new_sequent.antecedent.type = ['.<>.', sequent.
                            antecedent]
185                     new_sequent.antecedent.polarity = None
186 
187                     new_sequent.consequent = sequent.consequent.type[1]
188 
189                     return new_sequent  # .<>.X => Y
190             elif rule == "RC:.:":
191                 if correct_connective(sequent.consequent, '.:.:.'):  # X =>
                        .:.:.Y
192                     new_sequent.antecedent.type = ['.<.>.', sequent.
                            antecedent]
193                     new_sequent.antecedent.polarity = None
194 
195                     new_sequent.consequent = sequent.consequent.type[1]
196 
197                     return new_sequent  # .<.>.X => Y
198             elif rule == "NL1":
199                 # the antecedent must be of the form ((.<>.A).*.B).*.C
```

```
200             antecedent = sequent.antecedent
201             if correct_connective(antecedent, '.*.'):
202                 a_b = antecedent.type[0]
203                 if correct_connective(a_b, '.*.'):
204                     a = a_b.type[0]
205                     if correct_connective(a, '.<>.'):
206                         b = a_b.type[2]
207                         c = antecedent.type[2]
208                         b_c = Formula([b, '.*.', c], None, b.index + c.
                                index, 'input')
209 
210                         new_sequent.antecedent.type = [a, '.*.', b_c]
                            # (.<>.A).*.(B.*.C)
211                         new_sequent.consequent = sequent.consequent  #
                                use the original consequent, not a deepcopy
212 
213                         return new_sequent
214         elif rule == "NL2":
215             # the antecedent must be of the form B.*.((.<>.A).*.C)
216             antecedent = sequent.antecedent
217             if correct_connective(antecedent, '.*.'):
218                 a_c = antecedent.type[2]
219                 if correct_connective(a_c, '.*.'):
220                     a = a_c.type[0]
221                     if correct_connective(a, '.<>.'):
222                         b = antecedent.type[0]
223                         c = a_c.type[2]
224                         b_c = Formula([b, '.*.', c], None, b.index + c.
                                index, 'input')
225 
226                         new_sequent.antecedent.type = [a, '.*.', b_c]
                            # (.<>.A).*.(B.*.C)
227                         new_sequent.consequent = sequent.consequent  #
                                use the original consequent, not a deepcopy
228 
229                         return new_sequent
230         elif rule == "EN1":
231             # the antecedent must be of the form A.*.(B.*.(.<>.C))
232             antecedent = sequent.antecedent
233             if correct_connective(antecedent, '.*.'):
234                 b_c = antecedent.type[2]
235                 if correct_connective(b_c, '.*.'):
236                     c = b_c.type[2]
```

```
237                     if correct_connective(c, '.<>.'):
238                         a = antecedent.type[0]
239                         b = b_c.type[0]
240                         a_b = Formula([a, '.*.', b], None, a.index + b.
                                index, 'input')
241
242                         new_sequent.antecedent.type = [a_b, '.*.', c]
                              # (A.*.B).*.(.<>.C)
243                         new_sequent.consequent = sequent.consequent  #
                                use the original consequent, not a deepcopy
244
245                         return new_sequent
246         elif rule == "EN2":
247             # the antecedent must be of the form (A.*.(.<>.C)).*.B
248             antecedent = sequent.antecedent
249             if correct_connective(antecedent, '.*.'):
250                 a_c = antecedent.type[0]
251                 if correct_connective(a_c, '.*.'):
252                     c = a_c.type[2]
253                     if correct_connective(c, '.<>.'):
254                         a = a_c.type[0]
255                         b = antecedent.type[2]
256                         a_b = Formula([a, '.*.', b], None, a.index + b.
                                index, 'input')
257
258                         new_sequent.antecedent.type = [a_b, '.*.', c]
                              # (A.*.B).*.(.<>.C)
259                         new_sequent.consequent = sequent.consequent  #
                                use the original consequent, not a deepcopy
260
261                         return new_sequent
262         else:
263             raise RuleError(rule)
264
265     return None
266
267
268 def apply_rule(rule, formula, other_formula):
269     """Applies the requested rule to the given couple of Formulas.
270
271     Parameters
272     ----------
273     rule : str
```

```
274         The rule to be applied
275     formula : Formula
276         The known Formula
277     other_formula : Formula
278         The ncomplete Formula
279 
280     Returns
281     -------
282     list of Sequent
283         The resulting Sequent(s)
284 
285     Raises
286     ------
287     RuleError
288         If the requested rule does not exist
289 
290     Examples
291     --------
292     The rule "/L" applied to formula with summary (s0/np1) and
           other_formula with summary (??./.??) returns a list
293     containing elements with summaries [s0] => s? and np? => [np1]. The
           summary of other_formula is now (s?./.np?).
294     """
295     applied = []
296 
297     if rule == "fL":
298         applied.append(Sequent(formula, other_formula, "l"))
299     elif rule == "fR":
300         applied.append(Sequent(other_formula, formula, "r"))
301     elif rule == "dfL":
302         applied.append(Sequent(formula, other_formula))
303     elif rule == "dfR":
304         applied.append(Sequent(other_formula, formula))
305     elif rule == "<>L":
306         antecedent = Formula([".<>.", formula.type[1]], None, formula.
               index, 'input')
307         applied.append(Sequent(antecedent, other_formula))
308     elif rule == "<.>L":
309         antecedent = Formula([".<.>.", formula.type[1]], None, formula.
               index, 'input')
310         applied.append(Sequent(antecedent, other_formula))
311     elif rule == "::R":
312         consequent = Formula([".:::.", formula.type[1]], None, formula.
```

```
            index, 'output')
313         applied.append(Sequent(other_formula, consequent))
314     elif rule == ":.:R":
315         consequent = Formula([".:.:.", formula.type[1]], None, formula.
                index, 'output')
316         applied.append(Sequent(other_formula, consequent))
317     elif rule == "/R":
318         consequent = Formula([formula.type[0], "./.", formula.type[2]],
                None, formula.index, 'output')
319         applied.append(Sequent(other_formula, consequent))
320     elif rule == "*L":
321         antecedent = Formula([formula.type[0], ".*.", formula.type[2]],
                None, formula.index, 'input')
322         applied.append(Sequent(antecedent, other_formula))
323     elif rule == "\R":
324         consequent = Formula([formula.type[0], ".\.", formula.type[2]],
                None, formula.index, 'output')
325         applied.append(Sequent(other_formula, consequent))
326     elif rule in ("::L", ":.:L"):
327         applied.append(Sequent(formula.type[1], other_formula.type[1],
                "l"))
328     elif rule in ("<>R", ":<.>R"):
329         applied.append(Sequent(other_formula.type[1], formula.type[1],
                "r"))
330     elif rule == "/L":
331         applied.append(Sequent(formula.type[0], other_formula.type[0],
                "l"))
332         applied.append(Sequent(other_formula.type[2], formula.type[2],
                "r"))
333     elif rule == "*R":
334         applied.append(Sequent(other_formula.type[0], formula.type[0],
                "r"))
335         applied.append(Sequent(other_formula.type[2], formula.type[2],
                "r"))
336     elif rule == "\L":
337         applied.append(Sequent(formula.type[2], other_formula.type[2],
                "l"))
338         applied.append(Sequent(other_formula.type[0], formula.type[0],
                "r"))
339     else:
340         raise RuleError(rule)
341
342     return applied
```

ConvertInput.py

```
"""
This module contains functions related to converting an input string to
    a Sequent.

Methods
-------
str_to_list(s) : list
    Transforms the given antecedent or consequent from a string into a
        list.
get_positions(formula, position) : tuple of (str, str)
    Gets the positions of sub formulas A and B in [A, $, B].
convert_formula(formula, polarities, index, position) : tuple of (
    Formula, int)
    Transforms the given formula from a list or string into a Formula.
sequent(antecedent, consequent, polarities, input_method, language,
    focus=0): Sequent
    Transforms the given antecedent and consequent from strings to a
        Sequent.
"""
from Exception import InputError
from Formula import Formula
from Lexicon import lexicon
from Rules import connective_as_latex_code
from Sequent import Sequent

from ast import literal_eval
from re import findall, sub


def str_to_list(s):
    """Transforms the given antecedent or consequent from a string into
        a list.

    Parameters
    ----------
    s : str
        The formula in string form

    Returns
    -------
    list
```

```
36          The formula in list form. Atoms are strings, complex formulas
               are lists of the form [A, $, B] or [#, A],
37          where $ and # are strings representing connectives and A and B
               are atoms or complex formulas.
38
39      Examples
40      ---------
41      >>> str_to_list("((np\s)/np)")
42      [['np', '\\', 's'], '/', 'np']
43      """
44      if not s.isalpha():  # s is not an atom
45          # convert all formula characters to string form and separate
               them from the surrounding string with commas
46          s = s.replace('(', '[').replace(')', ']')
47          s = s.replace('/', ', "/", ')
48          s = s.replace('\\', ', "\\\\", ')
49          s = s.replace('*', ', "*", ')
50          s = s.replace('<>', '"<>", ')
51          s = s.replace('<.>', '"<.>", ')
52          s = s.replace('::', '"::", ')
53          s = s.replace(':.:', '":.:", ')
54
55          # convert all letters to string form
56          types = set(findall('\w', s))
57          for w in types:
58              s = sub(w, f'\"{w}\"', s)
59          # atom type strings that are more than one character long
               contain too many quotation marks; remove excessive
60          # ones
61          s = s.replace('""', '')
62          s = f'[{s}]'
63
64          try:
65              s = literal_eval(s)
66          except (SyntaxError, ValueError):
67              connectives = list(connective_as_latex_code.keys())[7:]
68              raise InputError('There may be a problem with the
                   connectives in the types. Please make sure that you are
                   '
69                               f'only using the connectives {str(
                                   connectives)[1:-1]}.')
70
71      # return the string as a list
```

```
72      return s
73
74
75  def get_positions(formula, position):
76      """Gets the positions of sub formulas A and B in [A, $, B].
77
78      Parameters
79      ----------
80      formula : list
81          A formula of the form [A, $, B]
82      position : str
83          Formulas should be either an 'input' or an 'output' Formula.
84
85      Returns
86      -------
87      tuple of (str, str)
88          The positions of A and B
89
90      Raises
91      ------
92      InputError
93          If the connective of the Formula or the position is incorrect
94
95      Examples
96      --------
97      >>> get_positions(['np', '\', 's'], 'input')
98      ('output', 'input')
99      """
100     if (formula[1] == '/' and position == 'input' or  # A/B => C
101             formula[1] == '\\' and position == 'output'):  # C => A\B
102         pos_a = 'input'
103         pos_b = 'output'
104     elif (formula[1] == '/' and position == 'output' or  # C => A/B
105           formula[1] == '\\' and position == 'input'):  # A\B => C
106         pos_a = 'output'
107         pos_b = 'input'
108     elif formula[1] == '*' and position == 'input':  # A*B => C
109         pos_a = 'input'
110         pos_b = 'input'
111     elif formula[1] == '*' and position == 'output':  # C => A*B
112         pos_a = 'output'
113         pos_b = 'output'
114     else:
```

```
115             raise InputError(f"Binary connective {formula[1]} is not '/', '\\' or '*' or position {position} is not "
116                              f"'input' or 'output'.")
117
118     return pos_a, pos_b
119
120
121 def convert_formula(formula, polarities, index, position):
122     """Transforms the given formula from a list or string into a
            Formula.
123
124     Parameters
125     ----------
126     formula: str or list
127         A formula as returned by str_to_list
128     polarities : dict of {str : str}
129         The polarities of the atoms
130     index: int
131         The index of the next atom
132     position : str
133         Formulas must be either an 'input' or an 'output' Formula.
134
135     Returns
136     -------
137     tuple of (Formula, int)
138         The converted Formula, the index of the next atom
139
140     Raises
141     ------
142     InputError
143         If something in the given input is incorrect, for example:
144             a Formula is not given a polarity, or
145             the connective of a Formula is incorrect, or
146             a Formula is not of the correct form
147
148     Examples
149     --------
150     >>> convert_formula(['np', '\', 's'], {'np': '+', 's': '-'}, 0, '
            input')
151     (Formula([Formula(np, +, [0], output), '\', Formula(s, -, [1],
            input)], -, [0, 1], input), 2)
152     """
153     if isinstance(formula, str):  # the formula is an atom
```

```
154         if formula in polarities:
155             converted_formula = Formula(formula, polarities[formula], [
                    index], position)
156             new_index = index + 1
157         elif formula in connective_as_latex_code.keys():
158             raise InputError("There may be a problem with the
                    bracketing of some of the types. Please make sure that "
159                              "every complex formula is put between
                                 parentheses.")
160         else:
161             raise InputError(f"The polarity of '{formula}' is missing
                    in {polarities}. Please add it to the dictionary "
162                              "'polarities'.")
163     elif isinstance(formula, list):  # the formula is complex
164         if len(formula) == 2:  # the formula is unary
165             a = convert_formula(formula[1], polarities, index, position
                    )
166             formula[1] = a[0]  # replace the formula in string form
                    with the Formula
167             new_index = a[1]
168 
169             if formula[0] in ('<>', '<.>'):
170                 polarity = '+'
171             elif formula[0] in ('::', ':.:'):
172                 polarity = '-'
173             else:
174                 raise InputError(f"The connective '{formula[0]}' is not
                        unary. Please make sure that every unary "
175                                  "connective is either '<>', '<.>',
                                     '::' or ':.:'.")
176             converted_formula = Formula(formula, polarity, list(range(
                    index, new_index)), position)
177         elif len(formula) == 3:  # the formula is binary
178             positions = get_positions(formula, position)
179             pos_a = positions[0]
180             pos_b = positions[1]
181 
182             a = convert_formula(formula[0], polarities, index, pos_a)
183             formula[0] = a[0]  # replace the left formula in string
                    form with the left Formula
184             new_index = a[1]
185 
186             b = convert_formula(formula[2], polarities, new_index,
```

```
                pos_b)
187             formula[2] = b[0]  # replace the right formula in string
                    form with the right Formula
188             new_index = b[1]
189
190             if formula[1] == '*':
191                 polarity = '+'
192             elif formula[1] in {'/', '\\'}:
193                 polarity = '-'
194             else:
195                 raise InputError(f"The connective '{formula[1]}' is not
                        binary. Please make sure that every binary "
196                                  r"connective is either '*', '/' or
                                        '\'.")
197             converted_formula = Formula(formula, polarity, list(range(
                    index, new_index)), position)
198         else:
199             raise InputError(f"Complex formula {formula} is not of form
                    [#, A] or [A, $, B].")
200     else:
201         raise InputError(f"Formula {formula} is not of form a, [#, A]
                or [A, $, B].")
202
203     return converted_formula, new_index
204
205
206 def sequent(antecedent, consequent, polarities, input_method, language,
        focus=0):
207     """Transforms the given antecedent and consequent from strings to a
            Sequent.
208
209     Parameters
210     ----------
211     antecedent : str
212         The antecedent in string form
213     consequent : str
214         The consequent in string form
215     polarities : dict of {str : str}
216         The polarities of the atoms
217     input_method : str
218         The antecedent can be either a combination of types or a
                combination of words. Which one must be specified.
219     language : str
```

```
220         If the antecedent is a combination of words, the language must
               be specified.
221     focus : str or int, optional
222         The focusing must be 0 if the sequent is neutral, 'l' if the
               antecedent is focused or 'r' if the consequent is
223         focused.
224
225     Returns
226     -------
227     Sequent
228
229     Raises
230     ------
231     InputError
232         If the consequent consists of multiple formulas
233
234     Examples
235     --------
236     >>> sequent('np,(<>(::(np\s)))', 's', {'np': '+', 's': '-'}, 'types
            ', None)
237     Sequent([Formula(np, +, [0], input),
238              Formula(['<>', Formula(
239                              ['::', Formula(
240                                  [Formula(np, +, [1], output), '\\',
                                        Formula(s, -, [2], input)],
241                               -, [1, 2], input)],
242                          -, [1, 2], input)],
243              +, [1, 2], input)],
244             [Formula(s, -, [3], output)],
245             0)
246     >>> sequent('bob likes alice', 's', {'np': '+', 's': '-'}, 'words',
            'English', 'r')
247     Sequent([Formula(np, +, [0], input),
248              Formula([Formula(
249                          [Formula(np, +, [1], output), '\\', Formula(s,
                                -, [2], input)],
250                          -, [1, 2], input),
251                          '/',
252                          Formula(np, +, [3], output)],
253              -, [1, 2, 3], input),
254              Formula(np, +, [4], input)],
255             s5,
256             r)
```

```
257         """
258         if input_method == "words":
259             words_to_types = ""
260 
261             lex = lexicon(language)
262             words_not_in_lex = []
263             for word in antecedent.split(" "):
264                 try:
265                     # get the type of the word from the lexicon, separate
                            the types with commas
266                     words_to_types += lex[word.lower()] + ","
267                 except KeyError:
268                     words_not_in_lex.append(word)
269             if words_not_in_lex:
270                 if len(words_not_in_lex) == 1:
271                     raise InputError(f"The lexicon with language '{language
                            }' does not contain the word "
272                                      f"'{str(words_not_in_lex[0])}'. Please add it
                                         to the lexicon or use another word.")
273                 else:
274                     raise InputError(f"The lexicon with language '{language
                            }' does not contain the words "
275                                      f"{str(words_not_in_lex)[1:-1]}. Please add
                                         them to the lexicon or use other words.")
276             ant = str_to_list(words_to_types[:-1])  # Ignore the last comma
277         elif input_method == "types":
278             ant = str_to_list(antecedent)
279         else:
280             raise InputError(f"Please specify the input method as 'words'
                or 'types' instead of '{input_method}'.")
281         con = str_to_list(consequent)
282 
283         if not (isinstance(con, str) or
284                 con[0] in ('<>', '<.>', '::', ':.:') or
285                 (len(con) == 3 and con[1] in ('/', '\\', '*'))):
286             raise InputError(f"The consequent '{str(con)}' consists of {len
                (con)} formulas. Please make sure that it"
287                              f"consists of only one formula.")
288         else:
289             # an input of the form a, #a or a$b (where a and b are atoms)
                    may be given as "a", "#a" or "a$b" instead of
290             # "(a)", "(#a)" or "(a$b)"
291             # to be able to do this, the formula needs to be put in a list
```

```
            to match the other forms of input
292         if (isinstance(ant, str) or
293                 ant[0] in ('<>', '<.>', '::', ':.:') or
294                 (len(ant) == 3 and ant[1] in ('/', '\\', '*'))):
295             ant = [ant]
296
297         index = 0
298         ant_formulas = []
299         for formula in ant:
300             data = convert_formula(formula, polarities, index, 'input')
301             ant_formulas.append(data[0])
302             index = data[1]
303
304         con_formula = convert_formula(con, polarities, index, 'output')
               [0]
305
306         return Sequent(ant_formulas, con_formula, focus)
```

PartialProofTree.py

```
1   """
2   This module contains functions related to the construction of PPTs.
3
4   Methods
5   -------
6   change_focus(formula, other_formula, sequent, proof, rule) : Sequent
7       Applies (de)focusing rules.
8   fill_structure(formula, other_formula)
9       Fills in parts of other_formula based on formula.
10  ppt(formula, sequent=None, proof=None, focus=0) : list of Tree
11      Constructs PPTs for the given Formula.
12  """
13  from Exception import FormulaTypeError, PositionError
14  from Formula import Formula
15  from Rules import apply_rule
16  from Sequent import Sequent
17  from Tree import Tree
18
19  from copy import deepcopy
20  from sys import maxsize
21
22
23  def change_focus(formula, other_formula, sequent, proof, rule):
```

```
24      """Applies (de)focusing rules.
25
26      Parameters
27      ----------
28      formula : Formula
29          The known Formula
30      other_formula : Formula
31          The possibly incomplete Formula
32      sequent : Sequent
33          The Sequent consisting of formula and other_formula
34      proof : Tree
35          The PPT
36      rule : str
37          The rule to be applied
38
39      Returns
40      -------
41      Sequent
42          The resulting Sequent
43      """
44      new_sequent = apply_rule(rule, formula, other_formula)[0]
45      proof.add_edge(new_sequent, sequent, rule, top_down=True)
46
47      return new_sequent
48
49
50  def fill_structure(formula, other_formula):
51      """Fills in parts of other_formula based on formula.
52
53      In the process of constructing a PPT unknown properties of Formulas
            can sometimes be filled in. This function
54      fills other_formula based on the properties of formula. This
           function should only be called when the Sequent with
55      formula and other_formula is focused.
56
57      Parameters
58      ----------
59      formula : Formula
60          The known Formula
61      other_formula : Formula
62          The incomplete Formula
63
64      Raises
```

```
65      ------
66      FormulaTypeError
67          If the type of formula is not of the correct form
68      """
69      empty_formula = deepcopy(other_formula)
70
71      if formula.is_atom():  # the Sequent is an axiom, so formula and
            other_formula must have the same type and polarity
72          other_formula.type = formula.type
73          other_formula.polarity = formula.polarity
74      elif formula.is_unary():
75          # the Sequent can only form a PPT if the connective of
                other_formula is the structural counterpart of the
76          # connective of formula
77          connective = f".{formula.type[0]}."
78          other_formula.type = [connective, empty_formula]
79      elif formula.is_binary():
80          # the Sequent can only form a PPT if the connective of
                other_formula is the structural counterpart of the
81          # connective of formula
82          connective = f".{formula.type[1]}."
83          empty_formula2 = deepcopy(empty_formula)
84          if formula.type[1] == "/":
85              if other_formula.position == 'input':
86                  empty_formula2.position = 'output'
87              else:
88                  empty_formula2.position = 'input'
89          elif formula.type[1] == "\\":
90              if other_formula.position == 'input':
91                  empty_formula.position = 'output'
92              else:
93                  empty_formula.position = 'input'
94          other_formula.type = [empty_formula, connective, empty_formula2
                ]
95          other_formula.index = empty_formula.index + empty_formula2.
                index
96      else:
97          raise FormulaTypeError(formula)
98
99
100 def ppt(formula, sequent=None, proof=None, focus=0):
101     """Constructs PPTs for the given Formula.
102
```

```
103     Part of a proof is deterministic: a certain form of Sequent has
            exactly one possible rule that can be applied in a
104     top down fashion. This function constructs this deterministic part.
105
106     Parameters
107     ----------
108     formula : Formula
109         The base Formula for the PPT
110     sequent : Sequent
111         The Sequent containing formula, optional
112     proof : Tree
113         The PPT that has formula as a leaf, optional
114     focus : str or int
115         The focus of sequent, optional
116
117     Returns
118     -------
119     list of Tree
120         the PPTs for the given Formula
121
122     Raises
123     ------
124     InputError
125         If the position of the Formula is not 'input' or 'output'
126     FormulaTypeError
127         If the type of the Formula is not of the form a, [#, A] or [A,
                $, B]
128     """
129     if proof is None:
130         proof = Tree()
131     ppts = [proof]
132
133     rule = ""
134     if formula.is_complex():
135         rule = formula.type[len(formula.type) - 2]  # the rule to be
                applied includes the connective
136
137     if formula.position == 'input':
138         rule += "L"  # the rule to be applied includes the side of the
                focused Formula
139         if sequent is None:
140             other_formula = Formula('?', '?', [maxsize], 'output')
141             sequent = Sequent(formula, other_formula, focus)
```

```
142             else:
143                 other_formula = sequent.consequent
144 
145             if formula.polarity == "-":
146                 if sequent.focus == 0:
147                     sequent = change_focus(formula, other_formula, sequent,
                            proof, "fL")
148                     formula = sequent.antecedent
149 
150                 fill_structure(formula, other_formula)
151             elif formula.polarity == "+":
152                 if sequent.focus == "l":
153                     sequent = change_focus(formula, other_formula, sequent,
                            proof, "dfL")
154                     formula = sequent.antecedent
155         elif formula.position == 'output':
156             rule += "R"  # the rule to be applied includes the side of the
                focused Formula
157             if sequent is None:
158                 other_formula = Formula('?', '?', [maxsize], 'input')
159                 sequent = Sequent(other_formula, formula, focus)
160             else:
161                 other_formula = sequent.antecedent
162 
163             if formula.polarity == "+":
164                 if sequent.focus == 0:
165                     sequent = change_focus(formula, other_formula, sequent,
                            proof, "fR")
166                     formula = sequent.consequent
167 
168                 fill_structure(formula, other_formula)
169             elif formula.polarity == "-":
170                 if sequent.focus == "r":
171                     sequent = change_focus(formula, other_formula, sequent,
                            proof, "dfR")
172                     formula = sequent.consequent
173         else:
174             raise PositionError(formula.position)
175 
176         if len(rule) > 1:  # the rule consists of both a connective and a
            side
177             applied = apply_rule(rule, formula, other_formula)
178             for new_sequent in applied:
```

```
179             proof.add_edge(new_sequent, sequent, rule, top_down=True)
180
181         if rule in ("/L", "\L", "::L", ":.:L", "*R", "<>R", "<.>R"):
182             for new_sequent in applied:
183                 if new_sequent.focus == "l":
184                     root_formula = new_sequent.antecedent
185                 elif new_sequent.focus == "r":
186                     root_formula = new_sequent.consequent
187                 else:  # this should never happen as all of the rules
                        above shift the focus to the sub Formulas
188                     continue
189                 # extend the PPT with PPTs for the leafs
190                 ppts = ppt(root_formula, new_sequent, proof)
191         elif formula.is_complex():
192             if formula.is_unary():
193                 # construct PPTs for the sub Formulas in the leaf
194                 additional_ppts = ppt(formula.type[1])
195             elif formula.is_binary():
196                 # construct PPTs for the sub Formulas in the leafs
197                 additional_ppts = ppt(formula.type[0]) + ppt(formula.
                        type[2])
198             else:
199                 raise FormulaTypeError(formula)
200             for tree in additional_ppts:
201                 root = tree.root()
202                 # if a PPT consists of exactly one node, only append it
                        to ppts if it is an axiom
203                 if len(tree.nodes) > 1 or root.antecedent.type == root.
                        consequent.type:
204                     ppts.append(tree)
205     elif sequent not in proof.nodes:
206         proof.add_node(sequent, top_down=True)
207
208     return ppts
```

CombinePPTs.py

```
1  """
2  This module contains functions related to the merging of two Trees.
3
4  Methods
5  -------
6  compatible(root_formula, leaf_formula) : bool
```

```
7       Returns whether two Formulas can be merged into one.
8   combine_sequents(root, leaf) : Sequent or None
9       If compatible, merges two Sequents into one.
10  update_formula_index(formula)
11      Checks and corrects the index property of a Formula.
12  update_indices(ppt)
13      Checks and corrects the index property of all Formulas in a Tree.
14  replace_formula_with(formula, new_formula)
15      Replaces the unknown properties "type" and "polarity" of a Formula
            with the known ones of another Formula.
16  fill_in_blanks(node, new_node)
17      Replaces the unknown properties in the sides of a Sequent with the
            known ones of another Sequent.
18  combine_ppts(root_proof, leaf_proof, leaf, combined) : Tree
19      Merges two Trees.
20  """
21  from Formula import Formula
22  from Sequent import Sequent
23  from Tree import Tree
24
25  from copy import deepcopy
26  from sys import maxsize
27
28
29  def compatible(root_formula, leaf_formula):
30      """Returns whether two Formulas can be merged into one.
31
32      Parameters
33      ----------
34      root_formula : Formula
35          A Formula in the root proof
36      leaf_formula : Formula
37          A Formula in the leaf proof
38
39      Returns
40      -------
41      bool
42          Whether two Formulas can be merged
43
44      Examples
45      --------
46      A Formula with summary np0\s1 is compatible with np?\?? but not np2
            \s? or (np0\s1)/np?.
```

```
47      """
48      if root_formula.type == '?' or leaf_formula.type == '?':
49          return True
50
51      if (root_formula.is_unary() and leaf_formula.is_unary() and
52              root_formula.type[0] == leaf_formula.type[0]):  # check if
                    the connectives match
53          return compatible(root_formula.type[1], leaf_formula.type[1])
54      elif (root_formula.is_binary() and leaf_formula.is_binary() and
55            root_formula.type[1] == leaf_formula.type[1]):  # check if
                  the connectives match
56          return (compatible(root_formula.type[0], leaf_formula.type[0])
                    and
57                  compatible(root_formula.type[2], leaf_formula.type[2]))
58      else:
59          return root_formula.type == leaf_formula.type and (root_formula
                .index == leaf_formula.index or
60                                                             root_formula
                                                                  .index ==
                                                                   [maxsize
                                                                  ] or
61                                                             leaf_formula
                                                                  .index ==
                                                                   [maxsize
                                                                  ])
62
63
64  def combine_sequents(root, leaf):
65      """If compatible, merges two Sequents into one.
66
67      Parameters
68      ----------
69      root : Sequent
70          The root of the root proof
71      leaf : Sequent
72          A leaf of the leaf proof
73
74      Returns
75      -------
76      Sequent or None
77          The merged Sequent if compatible, None otherwise
78
79      Examples
```

```
80      ---------
81      A Sequent with summary np0\s1 => np?\?? is compatible with np?\??
           => np2\s3 but not np0/s1 => np2\s3.
82      """
83      if ((leaf.antecedent.is_incomplete() or
84              root.consequent.is_incomplete() and not leaf.consequent.
                   is_incomplete() or
85              not root.antecedent.is_incomplete() and not leaf.consequent
                   .is_incomplete()) and
86              (root.focus == leaf.focus or
87               # two sequents with a different focus can still combine
                    when one sequent is an axiom and the other an
88               # atomic formula on one side and an empty formula on the
                    other
89               root.antecedent.is_atom() and leaf.consequent.is_atom()
                    and
90               (root.consequent.type == '?' or leaf.antecedent.type == '?
                    ')))):
91          antecedent = root.antecedent
92          consequent = leaf.consequent
93      elif ((leaf.consequent.is_incomplete() or
94            root.antecedent.is_incomplete() and not leaf.antecedent.
                 is_incomplete() or
95            not leaf.antecedent.is_incomplete() and not root.consequent.
                 is_incomplete()) and
96            (root.focus == leaf.focus or
97             # two sequents with a different focus can still combine when
                  one sequent is an axiom and the other an
98             # atomic formula on one side and an empty formula on the
                  other
99             leaf.antecedent.is_atom() and root.consequent.is_atom() and
100            (leaf.consequent.type == '?' or root.antecedent.type == '?')
                  )):
101         antecedent = leaf.antecedent
102         consequent = root.consequent
103     else:
104         return None
105
106     if (not (antecedent.is_atom() and consequent.is_atom() and
        antecedent.type != consequent.type) and
107             compatible(root.antecedent, leaf.antecedent) and compatible
                    (root.consequent, leaf.consequent)):
108         # the focus of the combined sequent should be equivalent to the
```

```
            focus of the root unless it is neutral and the
109         # leaf is not
110         # it can never be the case that the neither the root nor the
                leaf is neutral but both have different focusing;
111         # when sequents with different focusing are allowed to be
                combined, one of them will always be neutral
112         focus = root.focus
113         if leaf.focus != 0:
114             focus = leaf.focus
115         return Sequent(antecedent, consequent, focus)
116     else:
117         return None
118
119
120 def update_formula_index(formula):
121     """Checks and corrects the index property of a Formula.
122
123     Parameters
124     ----------
125     formula : Formula
126     """
127     if formula.is_unary():
128         update_formula_index(formula.type[1])
129         formula.index = formula.type[1].index
130     elif formula.is_binary():
131         update_formula_index(formula.type[0])
132         update_formula_index(formula.type[2])
133         formula.index = formula.type[0].index + formula.type[2].index
134
135
136 def update_indices(ppt):
137     """Checks and corrects the index property of all Formulas in a Tree
            .
138
139     Parameters
140     ----------
141     ppt : Tree
142     """
143     for node in ppt.nodes:
144         update_formula_index(node.antecedent)
145         update_formula_index(node.consequent)
146
147
```

```
148 def replace_formula_with(formula, new_formula):
149     """Replaces the unknown properties "type" and "polarity" of a
        Formula with the known ones of another Formula.
150
151     Parameters
152     ----------
153     formula : Formula
154         The Formula of which the properties get replaced
155     new_formula : Formula
156         The Formula that contains the known properties
157     """
158     if formula.is_incomplete():
159         formula.index = new_formula.index
160
161         if formula.is_atom():
162             formula.type = new_formula.type
163             formula.polarity = new_formula.polarity
164         elif formula.is_unary():
165             replace_formula_with(formula.type[1], new_formula.type[1])
166         elif formula.is_binary():
167             replace_formula_with(formula.type[0], new_formula.type[0])
168             replace_formula_with(formula.type[2], new_formula.type[2])
169
170
171 def fill_in_blanks(node, new_node):
172     """Replaces the unknown properties in the sides of a Sequent with
        the known ones of another Sequent.
173
174     Parameters
175     ----------
176     node : Sequent
177         The Sequent of which the properties get replaced
178     new_node : Sequent
179         The Sequent that contains the known properties
180     """
181     node.focus = new_node.focus
182     replace_formula_with(node.antecedent, new_node.antecedent)
183     replace_formula_with(node.consequent, new_node.consequent)
184
185
186 def combine_ppts(root_proof, leaf_proof, leaf, combined):
187     """Merges two Trees. Two nodes, one of each Tree, that can be
        combined into one must be given.
```

```
188
189     Parameters
190     ----------
191     root_proof : Tree
192         The Tree of which the root can be combined
193     leaf_proof : Tree
194         The Tree of which a leaf can be combined
195     leaf : Sequent
196         The leaf to be combined
197     combined : Sequent
198         The Sequent that contains the properties that will be filled in
                in the root of the root proof
199
200     Returns
201     -------
202     Tree
203         The combination of the two Trees
204     """
205     if not root_proof.edges:
206         combined_ppt = deepcopy(leaf_proof)
207         for current_leaf in combined_ppt.leafs():
208             if current_leaf == leaf:
209                 fill_in_blanks(current_leaf, combined)
210     else:
211         combined_ppt = deepcopy(root_proof)
212         leaf_proof_copy = deepcopy(leaf_proof)
213         root = combined_ppt.root()
214         amount_of_nodes = len(combined_ppt.nodes)
215
216         if not leaf_proof.edges:
217             fill_in_blanks(root, combined)
218         else:
219             root.focus = combined.focus
220             # if the leaf proof keeps track of Sequents that were
                    previously combined
221             # add the Sequents to combined_proof
222             for sequent_and_location in leaf_proof_copy.
                    combined_sequents:
223                 leaf_location = leaf_proof.nodes.index(leaf)
224                 if sequent_and_location[1] >= leaf_location:
225                     sequent_and_location[1] += amount_of_nodes - 1
226                 else:
227                     sequent_and_location[1] += amount_of_nodes
```

```
228                     combined_ppt.combined_sequents.append(
                            sequent_and_location)
229
230                 # put the edges of the leaf proof in combined_proof (nodes
                        will be added automatically)
231                 for edge in leaf_proof_copy.edges:
232                     # leave out the leaf and add an edge from the root to
                            the conclusion of the leaf instead
233                     if edge.premise == leaf:
234                         combined_ppt.combined_sequents.insert(0, [edge.
                                premise, amount_of_nodes - 1])
235                         fill_in_blanks(edge.premise, combined)
236                         combined_ppt.add_edge(root, edge.conclusion, edge.
                                rule)
237                     else:
238                         combined_ppt.add_edge(edge.premise, edge.conclusion
                                , edge.rule)
239
240     # to get the correct indices throughout the combined Tree a list of
            Sequents is kept up and updated after every new
241     # combination of Trees
242     for sequent_and_location in combined_ppt.combined_sequents:
243         current_sequent = combined_ppt.nodes[sequent_and_location[1]]
244         correct_sequent = sequent_and_location[0]
245         if current_sequent != correct_sequent:
246             fill_in_blanks(current_sequent, correct_sequent)
247
248     update_indices(combined_ppt)
249
250     return combined_ppt
```

Prove.py

```
1  """
2  This module contains functions related to finding the derivation for a
       given sequent.
3
4  Methods
5  -------
6  descendants(proof, other_proofs, structural_rules) : list of tuple of (
       Tree, list of Tree)
7      Forms all possible combinations between proof and every Tree in
           other_proofs, as well as all possible Trees
```

```
8       resulting from the application of structural rules to proof.
9   check_for_axiom(seq, focus) : list of Tree
10      If seq is an axiom: returns a Tree containing only the axiom.
11  get_start_proof(ppts) : Tree
12      Returns the first proof in ppts of which all leafs are axioms.
13  nodes_in(ppts) : set of Sequent
14      Returns all unique nodes in ppts.
15  bfs(start_proof, ppts, goal, structural_rules) : list of Tree
16      Derives all possible proofs for the combination of start_proof and
            ppts in a Breath First Search fashion.
17  prove(antecedent, consequent, polarities, input_method, language, focus
        =0) : list of Tree
18      Constructs all proofs that can be derived.
19  """
20  from CombinePPTs import combine_sequents, combine_ppts
21  from ConvertInput import sequent
22  from Exception import *
23  from Rules import basic_structural_rules, dutch_rules, english_rules,
        apply_structural_rule
24  from Sequent import Sequent
25  from PartialProofTree import ppt
26  from ToLaTeX import to_latex
27  from Tree import Tree
28
29  from copy import deepcopy
30  import sys
31
32
33  def descendants(proof, other_proofs, structural_rules):
34      """Forms all possible combinations between proof and every Tree in
            other_proofs, as well as all possible Trees
35      resulting from the application of structural rules to proof.
36
37      Parameters
38      ----------
39      proof : Tree
40      other_proofs : list of Tree
41      structural_rules : list of str
42
43      Returns
44      -------
45      list of tuple of (Tree, list of Tree)
46          The main proof, the proofs that still need to be combined with
```

```
            the main proof
47      """
48      desc = []
49
50      for other_proof in other_proofs:
51          for leaf in proof.leafs():
52              combined = combine_sequents(other_proof.root(), leaf)
53              if combined:
54                  new_proof = combine_ppts(other_proof, proof, leaf,
                        combined)
55                  new_other_proofs = other_proofs[:]
56                  new_other_proofs.remove(other_proof)
57                  desc.append((new_proof, new_other_proofs))
58
59          for leaf in other_proof.leafs():
60              combined = combine_sequents(proof.root(), leaf)
61              if combined:
62                  new_proof = combine_ppts(proof, other_proof, leaf,
                        combined)
63                  new_other_proofs = other_proofs[:]
64                  new_other_proofs.remove(other_proof)
65                  desc.append((new_proof, new_other_proofs))
66
67      for rule in structural_rules:
68          proof_copy = deepcopy(proof)
69          root = proof_copy.root()
70          new_root = apply_structural_rule(rule, root)
71          if new_root:
72              proof_copy.add_edge(root, new_root, rule)
73              desc.append((proof_copy, other_proofs[:]))
74
75      return desc
76
77
78  def check_for_axiom(seq, focus):
79      """If seq is an axiom: returns a Tree containing only the axiom.
80
81      Parameters
82      ----------
83      seq : Sequent
84      focus : str
85
86      Returns
```

```
87      -------
88      list of Tree
89          If seq is an axiom: a Tree containing the axiom
90      """
91      if len(seq.antecedent) == 1:
92          antecedent = seq.antecedent[0]
93          consequent = seq.consequent
94          if antecedent.type == consequent.type:
95              return [Tree([Sequent(antecedent, consequent, focus)])]
96
97      return []
98
99
100 def get_start_proof(ppts):
101     """Returns the first proof in ppts of which all leafs are axioms.
102
103     Parameters
104     ----------
105     ppts : list of Tree
106
107     Returns
108     -------
109     Tree
110         A proof of which all leafs are axioms
111     """
112     for tree in ppts:
113         if tree.leafs() == tree.axioms():
114             return tree
115
116
117 def nodes_in(ppts):
118     """Returns all unique nodes in ppts.
119
120     Parameters
121     ----------
122     ppts : list of Tree
123
124     Returns
125     -------
126     set of Sequent
127         All unique nodes
128     """
129     all_nodes = set()
```

```
130     for tree in ppts:
131         for node in tree.nodes:
132             all_nodes.add(node)
133 
134     return all_nodes
135 
136 
137 def bfs(start_proof, ppts, goal, structural_rules):
138     """Derives all possible proofs for the combination of start_proof
           and ppts in a Breath First Search fashion.
139 
140     The start proof should be a Tree of which all leafs are axioms.
           Through merging with Trees in ppts and
141     applying structural rules this proof should form into derivations.
           The function returns the derivations that are
142     actual proofs.
143 
144     Parameters
145     ----------
146     start_proof : Tree
147         A proof where all of its leafs are axioms
148     ppts : list of Tree
149     goal : tuple of (list of int, list of int, int or str)
150         The integers in the antecedent and consequent and the focus of
               the root of the desired proof
151     structural_rules : list of str
152         The names of the rules that can be applied to start_proof
153 
154     Returns
155     -------
156     list of Tree
157         All possible proofs for the combination of start_proof and ppts
158     """
159     to_explore = [(start_proof, ppts)]
160     discovered = []
161     solutions = []
162 
163     while to_explore:
164         combination = to_explore.pop(0)
165         main_proof = combination[0]
166         other_proofs = combination[1]
167 
168         all_nodes = nodes_in([main_proof] + other_proofs)
```

```
169             if not (all_nodes in discovered or main_proof.contains_loop()):
170                 discovered.append(all_nodes)
171 
172                 if not other_proofs:
173                     root = main_proof.root()
174                     if (root.antecedent.index, root.consequent.index, root.
                            focus) == goal:
175                         solutions.append(main_proof)
176                 elif main_proof.leafs() != main_proof.axioms():  # one or
                        more leafs in main_proof is not an axiom anymore
177                     new_main_proof = get_start_proof(other_proofs)
178                     if new_main_proof:
179                         other_proofs.remove(new_main_proof)
180                         other_proofs.append(main_proof)
181                         to_explore += descendants(new_main_proof,
                                other_proofs, structural_rules)
182                 else:
183                     to_explore += descendants(main_proof, other_proofs,
                            structural_rules)
184 
185     return solutions
186 
187 
188 def prove(input_method, language, antecedent, consequent, polarities,
        focus=0):
189     """Constructs all proofs that can be derived.
190 
191     Parameters
192     ----------
193     input_method : str
194         Whether the antecedent consists of words or types
195     language : str
196         The languge in which the antecedent should be interpreted
197     antecedent : str
198         The antecedent as either words or types
199     consequent : str
200         The goal type for the antecedent
201     polarities : dict of { str : str }
202         The polarities of the atomic types in the antecedent and
                consequent
203     focus : str
204         Whether the antecedent, consequent or neither is focused
205 
```

```
206         Returns
207         -------
208         list of Tree
209             All proofs that can be derived
210         """
211         ppts = []
212         antecedent_indices = []
213 
214         structural_rules = basic_structural_rules
215         if language:
216             language = language.capitalize()
217             if language == "Dutch":
218                 structural_rules += dutch_rules
219             elif language == "English":
220                 structural_rules += english_rules
221 
222         seq = sequent(antecedent, consequent, polarities, input_method,
                language, focus)
223 
224         for formula in seq.antecedent:
225             if focus == 'l':
226                 if len(seq.antecedent) == 1:
227                     antecedent = seq.antecedent[0]
228                     if antecedent.is_atom() and antecedent.polarity == '-':
229                         # there are two possibilities in this situation:
230                         # 1. seq is an axiom and should return a proof with
                              just seq as a node
231                         # 2. the root of the derivation can not be
                              defocused (top down) an thus no proofs are
                              possible
232                         return check_for_axiom(seq, focus)
233                     else:
234                         ppts += ppt(formula, focus='l')
235                 else:  # no proofs are possible
236                     return []
237             else:
238                 ppts += ppt(formula)
239 
240             antecedent_indices += formula.index
241 
242         consequent = seq.consequent
243         if focus == 'r':
244             if consequent.is_atom() and consequent.polarity == '+':
```

```
245             # there are two possibilities in this situation:
246             # 1. seq is an axiom and should return a proof with just
                    seq as a node
247             # 2. the root of the derivation can not be defocused (top
                    down) an thus no proofs are possible
248             return check_for_axiom(seq, focus)
249         else:
250             ppts += ppt(consequent, focus=focus)
251     else:
252         ppts += ppt(consequent)
253
254     goal = (antecedent_indices, consequent.index, focus)
255
256     start_proof = get_start_proof(ppts)
257     if start_proof:
258         ppts.remove(start_proof)
259     else:  # no proofs are possible
260         return []
261
262     return bfs(start_proof, ppts, goal, structural_rules)
263
264
265 def get_derivations(input_method, language, antecedent, consequent,
    polarities, focus, show_indices=False):
266     """Constructs LaTeX and PDF files for all proofs that can be
        derived
267
268     Parameters
269     ----------
270     input_method : str
271         Whether the input antecedent consists of types or words
272     language : str
273         The languge in which the input antecedent should be interpreted
274     antecedent : str
275         The combination of antecedent types or words in the theorem
276     consequent : str
277         The goal type for the antecedent
278     polarities : dict of {str: str}
279         The polarities of the atomic types in the antecedent and
            consequent
280     focus : str
281         Whether the antecedent, consequent or neither is focused
282     show_indices : bool
```

```
283         Whether the indices of the atomic types in the antecedent and
                consequent should be shown in the derivation(s)
284     """
285     try:
286         sequent_proofs = prove(input_method, language, antecedent,
                consequent, polarities, focus)
287     except (FormulaTypeError, InputError, LanguageError, RuleError,
            KeyError) as error:
288         print(error)
289         sys.exit()
290     else:
291         if len(sequent_proofs) == 1:
292             s = ''
293         else:
294             s = 's'
295         print(f'{len(sequent_proofs)} solution{s} found!')
296         if sequent_proofs:
297             print(f"You can find the solution{s} in the file 'proof.pdf
                    ' in the same directory as the python files. If "
298                   "you have opened the file, please make sure that it
                        is closed before you run the prover again, "
299                   "otherwise it may not update properly.\n")
300
301             to_latex(sequent_proofs, show_indices)
```

ToTerm.py

```
1   """
2   This module contains functions related to calculating the proof term
        corresponding to a derivation.
3
4   Methods
5   -------
6   to_term(tree): str
7       Calculates the proof term of a tree.
8   """
9
10  from Exception import PositionError, RuleError
11  from Formula import Formula
12  from Rules import basic_structural_rules, dutch_rules, english_rules
13  from Tree import IncompleteTree, NullaryTree, UnaryTree, BinaryTree
14
15  terms = {}  # to keep track of Formulas that are assigned a proof term
```

```
    already


def get_formula_term(formula, input_index, output_index):
    """Returns the proof term of a Formula.

    Parameters
    ----------
    formula : Formula
    input_index : int
        The index of the next input term
    output_index : int
        The index of the next output term

    Returns
    -------
    str
        The proof term of the Formula
    """
    try:
        return terms[formula], input_index, output_index  # if the
            Formula was assigned a proof term already
    except KeyError:  # the Formula is not assigned a proof term yet
        if formula.position == 'input':
            terms[formula] = f'y_{{{input_index}}}'
            return terms[formula], input_index + 1, output_index
        elif formula.position == 'output':
            terms[formula] = f'\\beta_{{{output_index}}}'
            return terms[formula], input_index, output_index + 1
        else:
            raise PositionError(formula.position)


def to_term(tree, input_index=0, output_index=0):
    """Calculates the proof term of a tree.

    Parameters
    ----------
    tree : NullaryTree or UnaryTree or BinaryTree
    input_index : int
        The index of the next input term
    output_index : int
        The index of the next output term
```

```
57
58      Returns
59      -------
60      str
61          The proof term of the tree
62      """
63      if isinstance(tree, (NullaryTree, UnaryTree, BinaryTree)):
64          rule = tree.rule
65          antecedent = tree.conclusion.antecedent
66          consequent = tree.conclusion.consequent
67          if isinstance(tree, NullaryTree):
68              terms[antecedent] = f'x_{{{antecedent.index[0]}}}'
69              terms[consequent] = f'\\alpha_{{{consequent.index[0]}}}'
70              if rule == "Ax":
71                  parts_of_term = (terms[antecedent],)
72              elif rule == "CoAx":
73                  parts_of_term = (terms[consequent],)
74              else:
75                  raise RuleError(rule)
76          elif isinstance(tree, UnaryTree):
77              # calculate the proof term of the premise first and update
                    the indices of the next input and output terms
78              term_and_indices = to_term(tree.premise, input_index,
                    output_index)
79              premise_term = term_and_indices[0]
80              input_index = term_and_indices[1]
81              output_index = term_and_indices[2]
82              # structure what the proof term will look like using a
                    tuple
83              # any element in the tuple that is not a string will be
                    converted to its proof term later
84              if rule in basic_structural_rules + dutch_rules +
                    english_rules:
85                  parts_of_term = (premise_term,)
86              elif rule == "fL":
87                  parts_of_term = ('(', antecedent, '\\ ', premise_term,
                        ')')
88              elif rule == "fR":
89                  parts_of_term = ('(', consequent, '\\ ', premise_term,
                        ')')
90              elif rule == "dfL":
91                  parts_of_term = ('\lambda ', antecedent, '.',
                        premise_term)
```

```
92              elif rule == "dfR":
93                  parts_of_term = ('\lambda␣', consequent, '.',
                        premise_term)
94              elif rule in ("::R", ":.:R"):
95                  parts_of_term = (('\\texttt{{case}}\␣', consequent, '\␣
                        \\texttt{{of}}\␣'
96                                    '\Zip{{', consequent.type[1], '}}.',
                                        premise_term))
97              elif rule in ("<>L", "<.>L"):
98                  parts_of_term = (('\\texttt{{case}}\␣', antecedent, '\␣
                        \\texttt{{of}}\␣'
99                                    '\Zip{{', antecedent.type[1], '}}.',
                                        premise_term))
100             elif rule in ("/R", "\R"):
101                 parts_of_term = (('\\texttt{{case}}\␣', consequent, '\␣
                        \\texttt{{of}}\␣'
102                                   '\Zip{{', consequent.type[0], ',',
                                        consequent.type[2], '}}.',
                                        premise_term))
103             elif rule == "*L":
104                 parts_of_term = (('\\texttt{{case}}\␣', antecedent, '\␣
                        \\texttt{{of}}\␣'
105                                   '\Zip{{', antecedent.type[0], ',',
                                        antecedent.type[2], '}}.',
                                        premise_term))
106             elif rule in ("<>R", "<.>R", "::L", ":.:L"):
107                 parts_of_term = ('\Zip{{', premise_term, '}}')
108             else:
109                 raise RuleError(rule)
110         elif isinstance(tree, BinaryTree):
111             if rule in ("/L", "\L", "*R"):
112                 parts_of_term = ('\Zip{{', tree.left_premise, ',', tree
                        .right_premise, '}}')
113             else:
114                 raise RuleError(rule)
115         else:
116             return NotImplemented
117         term = ''
118         for part_of_term in parts_of_term:
119             # calculate the proof term of the Formula or tree and
                    update the indices of the next input and output terms
120             if isinstance(part_of_term, (Formula, NullaryTree,
                    UnaryTree, BinaryTree)):
```

```
121                 if isinstance(part_of_term, Formula):
122                     term_and_indices = get_formula_term(part_of_term,
                            input_index, output_index)
123                 else:
124                     term_and_indices = to_term(part_of_term,
                            input_index, output_index)
125                 part_of_term = term_and_indices[0]
126                 input_index = term_and_indices[1]
127                 output_index = term_and_indices[2]
128             term += part_of_term  # build the proof term
129         return term, input_index, output_index
130     elif isinstance(tree, IncompleteTree):
131         return '', input_index, output_index
132     else:
133         return NotImplemented
```

ToLaTeX.py

```
1  """
2  This module contains functions related to creating LaTeX and pdf files
       to present the given proofs in the most readable
3  way.
4
5  Methods
6  -------
7  to_tree(node, tree) : NullaryTree or UnaryTree or BinaryTree
8      Converts a tree in Tree form to a tree in XPremiseTree form.
9  formula_as_latex_code(formula, show_indices) : str
10     Formats a Formula in a way that is can be compiled as a proof in
           LaTeX.
11 sequent_as_latex_code(sequent, show_indices) : str
12     Formats a Sequent in a way that it can be compiled as a proof in
           LaTeX.
13 format_proof(root, edges, show_indices) : str
14     Formats a Tree in a way that it can be compiled as a proof in LaTeX
           .
15 to_latex(proofs, show_indices=False)
16     Creates LaTeX and pdf files to present the given proofs in the most
           readable way.
17 """
18 from Rules import connective_as_latex_code, rule_name_as_latex_code
19 from Sequent import Sequent
20 from ToTerm import to_term
```

```
21 from Tree import Tree, to_x_ary_tree, IncompleteTree, NullaryTree,
       UnaryTree, BinaryTree
22
23 import os
24 import subprocess
25 from sys import maxsize
26
27
28 def formula_as_latex_code(formula, show_indices):
29     """Formats a Formula in a way that is can be compiled as a proof in
           LaTeX.
30
31     Parameters
32     ----------
33     formula : Formula
34     show_indices : bool
35         Whether the indices of the atoms should be shown
36
37     Returns
38     -------
39     str
40         The formula that can be compiled in LaTeX
41     """
42     formatted_formula = ''
43
44     to_format = [formula]
45     while to_format:
46         part_of_formula = to_format.pop()
47         if isinstance(part_of_formula, str):
48             if part_of_formula in ('(', ')'):
49                 formatted_formula += part_of_formula
50             else:
51                 formatted_formula += f'{connective_as_latex_code[
                       part_of_formula]} '
52         elif part_of_formula.is_atom():
53             formatted_formula += part_of_formula.type
54             if show_indices:
55                 if part_of_formula.index[0] == maxsize:
56                     if not part_of_formula.type == '?':
57                         formatted_formula += '_?'
58                 else:
59                     formatted_formula += f'_{{{part_of_formula.index
                           [0]}}}'
```

```
60          elif part_of_formula.is_complex():
61              # because the function append() adds an element to the end
                    of a list, the parts of the complex formula
62              # should be added in reversed order
63              to_format.append(')')
64              for part_of_type in part_of_formula.type[::-1]:
65                  to_format.append(part_of_type)
66              to_format.append('(')
67
68      return formatted_formula
69
70
71  def sequent_as_latex_code(sequent, show_indices):
72      """Formats a Sequent in a way that it can be compiled as a proof in
            LaTeX.
73
74      Parameters
75      ----------
76      sequent : Sequent
77      show_indices : bool
78          Whether the indices of the atoms should be shown
79
80      Returns
81      -------
82      str
83          The sequent that can be compiled in LaTeX
84      """
85      antecedent = formula_as_latex_code(sequent.antecedent, show_indices
            )
86      consequent = formula_as_latex_code(sequent.consequent, show_indices
            )
87
88      # remove unnecessary outer brackets from complex formulas
89      if not sequent.antecedent.is_atom():
90          antecedent = antecedent[1:-1]
91      if not sequent.consequent.is_atom():
92          consequent = consequent[1:-1]
93
94      if sequent.focus == 'l':
95          antecedent = f'\\focus{{{antecedent}}}'
96      elif sequent.focus == 'r':
97          consequent = f'\\focus{{{consequent}}}'
98
```

```
99      return f'\seq{{{antecedent}}}{{{consequent}}}'
100
101
102 def tree_as_latex_code(tree, show_indices):
103     """Formats a tree in a way that it can be compiled as a proof in
            LaTeX.
104
105     Parameters
106     ----------
107     tree : NullaryTree or UnaryTree or BinaryTree
108     show_indices : bool
109         Whether the indices of the atoms should be shown
110
111     Returns
112     -------
113     str
114         The tree that can be compiled in LaTeX
115     """
116     if isinstance(tree, IncompleteTree):
117         return f'{sequent_as_latex_code(tree.conclusion, show_indices)}
                ␣\n'
118     elif isinstance(tree, NullaryTree):
119         return f'\infer[{rule_name_as_latex_code[tree.rule]}]' \
120                f'{{{sequent_as_latex_code(tree.conclusion, show_indices
                    )}}}{{}}␣\n'
121     elif isinstance(tree, UnaryTree):
122         return (f'\infer[{rule_name_as_latex_code[tree.rule]}]'
123                 f'{{{sequent_as_latex_code(tree.conclusion, ␣
                        show_indices)}}}{{␣\n'
124                 f'{tree_as_latex_code(tree.premise, show_indices)}}}')
125     elif isinstance(tree, BinaryTree):
126         return (f'\infer[{rule_name_as_latex_code[tree.rule]}]'
127                 f'{{{sequent_as_latex_code(tree.conclusion, ␣
                        show_indices)}}}{{␣\n'
128                 f'{tree_as_latex_code(tree.left_premise, show_indices)}
                        ␣&␣\n'
129                 f'{tree_as_latex_code(tree.right_premise, show_indices)
                        }}}')
130
131
132 def to_latex(proofs, show_indices=False):
133     """Creates LaTeX and pdf files to present the given proofs in the
            most readable way.
```

```
134
135     Template taken from http://baptisteravina.com/blog/python-to-latex/
136
137     Parameters
138     ----------
139     proofs : list of Tree
140     show_indices : bool
141         Whether the indices of the atoms should be shown
142     """
143     doc_class = '\documentclass{article}\n'
144     packages = ('\\usepackage[pdftex, active, tightpage]{preview}\n'
145                 '\\usepackage{amsmath}\n'
146                 '\\usepackage{amssymb}\n'
147                 '\\usepackage{proof}\n')
148     commands = ('\\newcommand{\seq}[2]{#1\Rightarrow #2}\n'
149                 '\\newcommand{\\focus}[1]{\\fbox{$#1\\rule{0pt}{6pt}$
                        }}\n'
150                 '\\newcommand{\\bs}{\\backslash}\n'
151                 '\\newcommand{\Zip}[1]{\langle #1 \\rangle}'
152                 '\\newcommand{\\fdia}{\diamondsuit}\n'
153                 '\\newcommand{\\fdiaf}{\\blacklozenge}\n'
154                 '\\newcommand{\gbox}{\Box}\n'
155                 '\\newcommand{\gboxf}{\\blacksquare}\n'
156                 '\\newcommand{\W}[1]{\\textsf{#1}}\n'
157                 '\\newcommand{\strs}{\cdot /  \cdot}\n'
158                 '\\newcommand{\strbs}{\cdot \\bs \cdot}\n'
159                 '\\newcommand{\strtens}{\cdot \otimes \cdot}\n'
160                 '\\newcommand{\strdia}{\cdot \, \\fdia \cdot}\n'
161                 '\\newcommand{\strdiaf}{\cdot \, \\fdiaf \cdot}\n'
162                 '\\newcommand{\strbox}{\cdot \, \gbox \cdot}\n'
163                 '\\newcommand{\strboxf}{\cdot \, \\gboxf \cdot}\n')
164     setup = ('\pagestyle{empty}\n'
165              '\n'
166              '\\begin{document}\n'
167              '\n'
168              '\\addtolength{\inferLineSkip}{1pt}\n'
169              '\setlength{\PreviewBorder}{.5in}\n')
170     proofs_as_latex_code = ''
171     for proof in proofs:
172         tree = to_x_ary_tree(proof.root(), proof)
173         proofs_as_latex_code += ('\\begin{preview}\n'
174                                  f'$\deduce{{{to_term(tree)[0]}}}\n'
175                                  f'{{{tree_as_latex_code(tree, 
```

```
                                        show_indices)}}}$'
176                                   '\end{preview}\n')
177     end = '\end{document}'
178
179     content = ''
180     for part in (doc_class, packages, commands, setup,
            proofs_as_latex_code, end):
181         content += f'{part}\n'
182
183     with open('proof.tex', 'w') as file:  # create and open the file
            proof.tex
184         file.write(content)  # write the LaTeX code to proof.tex
185         file.close()
186
187     command_line = subprocess.Popen(['pdflatex', 'proof.tex'])  #
            create the pdf file
188     command_line.communicate()  # show the process in the terminal
189     # we don't need the files below: delete them
190     os.unlink('proof.aux')
191     os.unlink('proof.log')
```

main.py

```
1  from Prove import get_derivations
2
3
4  def main():
5      pass
6
7
8  main()
```

Exception.py

```
1  class Error(Exception):
2      """Base class for exceptions in this module"""
3      pass
4
5
6  class FormulaTypeError(Error):
7      """Exception raised for errors in the type of a Formula
8
9      Attributes
```

```
10      ----------
11      formula : Formula
12          the Formula with an incorrect form
13      """
14      def __init__(self, formula):
15          self.formula = formula
16
17      def __str__(self):
18          """Returns a readable representation of the Error.
19
20          Overrides the default method.
21
22          Returns
23          -------
24          str
25              A readable representation of the Error
26          """
27          return f"Formula type '{self.formula.type}' is not of form 'a',
                ' '[#, A]' or ' '[A, $, B]'."
28
29
30  class InputError(Error):
31      """Exception raised for errors in a user's input
32
33      Attributes
34      ----------
35      message : str
36          explanation of the error
37      """
38      def __init__(self, message):
39          self.message = message
40
41      def __str__(self):
42          """Returns a readable representation of the Error.
43
44          Overrides the default method.
45
46          Returns
47          -------
48          str
49              A readable representation of the Error
50          """
51          return self.message
```

```
52
53
54  class LanguageError(Error):
55      """Exception raised when a requested language is not implemented
56
57      Attributes
58      ----------
59      language : str
60      """
61      def __init__(self, language):
62          self.language = language
63
64      def __str__(self):
65          """Returns a readable representation of the Error.
66
67          Overrides the default method.
68
69          Returns
70          -------
71          str
72              A readable representation of the Error
73          """
74          return (f"The requested language '{self.language}' is not
                available in the lexicon. Please use an available "
75                  f"language or create a lexicon for '{self.language}'.")
76
77
78  class PositionError(Error):
79      """Exception raised when a Formula's position is not 'input' or '
            output'
80
81      Attributes
82      ----------
83      position : str
84      """
85      def __init__(self, position):
86          self.position = position
87
88      def __str__(self):
89          """Returns a readable representation of the Error.
90
91          Overrides the default method.
92
```

```
93          Returns
94          -------
95          str
96              A readable representation of the Error
97          """
98          return f"Position '{self.position}' is not 'input' or 'output'."
99
100
101 class RuleError(Error):
102     """Exception raised when a requested rule does not exist
103
104     Attributes
105     ----------
106     rule : str
107     """
108     def __init__(self, rule):
109         self.rule = rule
110
111     def __str__(self):
112         """Returns a readable representation of the Error.
113
114         Overrides the default method.
115
116         Returns
117         -------
118         str
119             A readable representation of the Error
120         """
121         return f"Requested rule '{self.rule}' does not exist."
```

examples.py

```
1  """
2  This module contains example inputs to give when running the main
       module.
3  """
4
5  from main import get_derivations
6
7  # use these examples after following the instructions in the README.txt
       file
8  get_derivations('types', None, 'np', 's/(np\s)', {'np': '+', 's': '-'},
```

```
    0, True)
get_derivations('types', None, 'np, (np\s)', 's', {'np': '+', 's': '-'
    }, 'r', True)
get_derivations('types', None, 'np', '::(<>np)', {'np': '+'}, 'l', True
    )
get_derivations('types', None, '<>(::np)', 'np', {'np': '+'}, 0, True)
get_derivations('types', None, r'(n/n), n, (n\n)', 'n', {'n': '+'}, 0,
    True)
get_derivations('types', None, '(s/(np\s)), ((s/(np\s))\s)', 's', {'np'
    : '+', 's': '-'}, 'r', True)

get_derivations('words', 'Dutch', 'lakei die alice plaagt', 'n', {'np':
    '+', 'n': '+', 's': '-'}, 0, True)
get_derivations('words', 'English', 'book that alice read', 'n', {'np':
    '+', 'n': '+', 's': '-'}, 0, True)
```