FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

— Master's Thesis —

Prototyping NLU Pipelines

A Type-Theoretical Framework

by

Jan Frederik Schaefer

Supervised by

Michael Kohlhase

Submitted on

November 30, 2020

Abstract

Humans use natural language to communicate with each other and to share and conserve knowledge. While computers are widely used for communication and knowledge dissemination, their ability to understand and reason with the available natural language content is still very limited. As a result, we usually have to fall back to shallow statistical processing for natural language processing applications. Applications that require complex reasoning with natural language remain largely out of reach.

A common approach to remedy the situation is to design a pipeline that translates natural language into a semantic representation that can be processed more easily. We call such a pipeline a *natural language understanding* (*NLU*) *pipeline*. While it is possible to implement NLU pipelines for highly specific situations, reaching a reasonably wide coverage is very difficult. More research is needed to determine what semantic representations are most suitable and how they can be obtained. If a researcher has a new idea for an NLU pipeline, they can evaluate it with a prototype implementation. Unfortunately, implementing such a prototype is a lot of work.

To facilitate the implementation of prototypes for NLU pipelines, I present the *Grammatical Logical Inference Framework* (*GLIF*). It combines three different frameworks that can be used to implement only parts of an NLU pipeline: GF, MMT and ELPI. I expect that GLIF will support NLU research by providing a way to easily create prototype implementations of new ideas.

Declaration

I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form[1]. All content that was taken from a third party either verbatim or in substance has been acknowledged as such.

Erlangen, 2020-11-30

Jan Frederik Schaefer

[1]Section 6.1.2 summarizes the work of my master's project. This is also indicated by a disclaimer at the beginning of the section.

Disclaimer on Pre-Published Work

Part of the work presented in this thesis has already been published elsewhere. Here is a brief summary of the relevant publications:

> [KS] introduces the Grammatical Logical Framework (GLF), which combines GF and MMT.
>
> [SAK20] introduces GLF as a tool for prototyping controlled natural languages for mathematics and briefly discusses the GLForTheL case study. It also introduces GLF's Jupyter interface.
>
> [Sch20] is my master's project. It discusses the GLForTheL case study in much greater detail. Section 6.1.2 summarizes the GLForTheL project.
>
> [Koh+20a] presents our work on generating ELPI provers from natural deduction calculi specified in MMT.
>
> [SK20] introduces GLIF as an extension of GLF that supports inference for semantic/pragmatic analysis.

The relevant sections are marked by disclaimers indicating what has been pre-published and what my contribution was. Furthermore, this thesis was strongly influenced by a lecture on logic-based natural language processing ([LBS]). In fact, GLIF was initially developed for and used in that lecture.

Contents

1 Introduction **1**

2 Preliminaries: Studying Natural Language Semantics **4**
2.1 What is the Meaning of Language? 4
2.2 Formal Semantics 5
2.3 Components of an NLU Pipeline 7
2.3.1 Preprocessing 8
2.3.2 Parsing 8
2.3.3 Semantics Construction 8
2.3.4 An Alternative Specification 11
2.3.5 Semantic/Pragmatic Analysis 12
2.4 Existing Frameworks for Language-to-Logic Translation 12
2.4.1 Blackburn and Bos' Prolog/RINL 12
2.4.2 Van Eijck and Unger's Haskell/CSFP 14
2.4.3 The Grammatical Framework GF 15
2.5 Logic Development with MMT 18
2.5.1 Logic Syntax in MMT Theories 19
2.5.2 MMT Views 20
2.5.3 Natural Deduction Calculus in MMT 21
2.5.4 Theorem Proving in MMT 21
2.6 Inference with ELPI via Helper Predicates 23
2.6.1 First, Naive Attempt 24
2.6.2 Helper Predicates and Proof Terms 24
2.6.3 Simple Hypotheses 26
2.6.4 Tracking Hypotheses 27
2.7 Summary 28

3 Grammatical Logical Framework (GLF) **29**
3.1 From GF to MMT 29
3.2 Semantics Construction in MMT 30
3.3 Testing the Pipeline 31
3.4 Summary 32

4 From GLF to GLIF: Adding Inference **34**
4.1 Basic Integration: From MMT to ELPI 34
4.2 Generating ELPI Provers from MMT 35
4.2.1 Intuition: Curry-Howard 35
4.2.2 Actual Translation 36
4.2.3 Overview: Generated Helper Predicates 38
4.3 Tableaux 39
4.3.1 Tableau Rules in LF 39
4.3.2 Tableau Proofs in LF 40

4.3.3 Generating Provers . . . 41
4.3.4 Model Generation . . . 41
4.3.5 Soundness and Completeness . . . 42
4.4 Summary: GLIF . . . 43

5 The GLIF System and Interface **45**
5.1 MMT Extensions . . . 45
5.2 Jupyter Interface . . . 46
5.2.1 Basic Functionality . . . 46
5.2.2 GLIF Commands . . . 46
5.2.3 Other Features . . . 49

6 Case Studies **50**
6.1 Controlled Natural Languages . . . 50
6.1.1 A Controlled Natural Input Language for Sage . . . 51
6.1.2 GLForTheL . . . 52
6.2 Question Answering in $S5_n$ . . . 54
6.2.1 Epistemic Modal Logic and $S5_n$ . . . 54
6.2.2 GLIF Implementation . . . 55
6.3 Translating Family Relations . . . 55
6.4 Tableaux Machine . . . 57

7 Evaluation **61**
7.1 Comparison with Existing Frameworks . . . 61
7.1.1 Parsing . . . 61
7.1.2 Lexicon . . . 63
7.1.3 Target Logic . . . 63
7.1.4 Semantics Construction . . . 64
7.1.5 Conclusion . . . 64
7.2 Jupyter Interface . . . 65
7.3 Limitations in MMT . . . 66
7.4 Semantic/Pragmatic Analysis . . . 68

8 Future Work **70**
8.1 Lexicon Generation . . . 70
8.2 GLIF Commands . . . 71
8.2.1 Using Structured Data . . . 71
8.2.2 User-Defined Commands . . . 72
8.3 Further Ideas . . . 72

9 Conclusion **74**

Bibliography **76**

A Example in Different Frameworks **80**
A.1 Prolog/RINL . . . 80

A.2 Haskell/CSFP 81
A.3 GF 84
A.3.1 Parsing 84
A.3.2 Semantics Construction 85
A.4 GLF 86

B ND and Tableau Rules in MMT **87**

C Notation Guide **89**
C.1 LF 89
C.2 Logical Operators 89

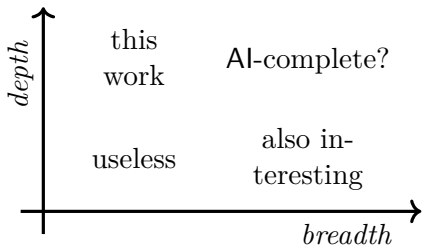
Index **90**

CHAPTER 1

Introduction

The ability to use complex language is one of the most amazing skills humans have. It enables us to transfer and archive knowledge efficiently and to live in a complex society. It sets us apart from the rest of the animal world. And it also sets us apart from software agents. In fact, creating software agents that truly comprehend human language is often considered an AI-complete problem (e.g. [Sha92, p. 56]).

We can classify applications by the depth and breadth of their understanding of natural language. An extreme example is text classification, which requires very little understanding but has a very broad coverage. Applications that require deep understanding, on the other hand, are very limited in their breadth. This can for example be seen in search engines and smart speakers. While they can directly answer simple questions like



"*How many people live in Slovakia?*",

more complex questions like

"*Do more people live in Slovakia than in Thailand?*"

cannot be answered directly at the time of writing[1] – even though they are based on the same knowledge. Another factor that limits the achievable breadth is the required precision. While it is acceptable if a smart speaker plays the wrong song, a digital lawyer should never provide wrong legal advice.

In this work we will focus on deep understanding – specifically, on the problem of translating natural language utterances into formal, semantic representations. We will refer to this as **natural language understanding** (**NLU**). Aside from the translation itself, it is also unclear what semantic representation is most suitable. First-order logic is a common choice but many alternatives exist. In fact, new logics (or new ways of using existing ones) are developed all the time. After the initial translation, the resulting expressions can be refined with a **semantic/pragmatic analysis**, e.g. to resolve ambiguities. We refer to such a system that translates sentences into a particular semantic representation as an **NLU pipeline**.

I believe that research into NLU pipelines could benefit from more prototyping – as an alternative to designing/sketching them on paper. Having prototypes helps researchers with evaluating and experimenting with different ideas – even if the

[1] WolframAlpha actually attempts an answer: "*75.1 million people*", which is the combined population size of Slovakia and Thailand.

natural language coverage of the prototypes is very limited. It also forces researchers to properly deal with all aspects of the pipeline, from the linguistic analysis to the semantic representation and the interplay with world knowledge.

In the past, NLU pipelines were often prototyped in general-purpose programming languages like Prolog or Haskell. That requires a lot of programming work, which often prevents researchers from developing a prototype in the first place. Dedicated frameworks for e.g. creating parsers or developing logics can significantly reduce the programming work required. But, to my knowledge, there are no dedicated frameworks for the implementation of complete NLU pipelines. Note that we are not interested in any solution that fixes the semantic representation.

Research Questions

1. Can we create a framework for prototyping the translation from natural language into a semantic representation by combining existing dedicated frameworks that solve part of the problem?

2. If such a framework is possible, can we seamlessly extend it to allow for semantic/pragmatic analysis?

3. Can the implementation of the semantic/pragmatic analysis be (partly) automated to facilitate rapid prototyping?

Contribution I present the Grammatical Logical Framework (GLF), a declarative framework for prototyping the translation from natural language into a semantic representation. It combines two existing frameworks: GF [GF] for grammar development and MMT [MMTb] for logic development. GLF can be used for the classic and paradigmatic Montagovian approach to natural language semantics. To incorporate the semantic/pragmatic analysis, I extend GLF with an inference component, resulting in the Grammatical Logical Inference Framework (GLIF). The inference component is based on ELPI [SCT15], an extension of λProlog. As the semantic/pragmatic analysis often requires inferential reasoning, I will describe how ELPI theorem provers can be automatically generated from calculi specified in MMT. I will evaluate the potential of GLIF as a prototyping framework for NLU pipelines based on a number of case studies.

Overview Chapter 2 discusses how we can study natural language semantics and what the components of an NLU pipeline are. It also introduces a number of existing frameworks. GLF is introduced in Chapter 3 and its extension GLIF in Chapter 4. The GLF/GLIF user interface and some remarks on its implementation are discussed in Chapter 5. Then we test GLIF in a number of case studies in Chapter 6 and evaluate it in Chapter 7. Chapter 8 discusses some future work and Chapter 9 concludes the thesis.

Acknowledgements First of all, I would like to thank Michael Kohlhase for being an inspiring and supportive supervisor not just for my master's thesis but throughout my journey into academia. I am also grateful to every other member of the KWARC group: Dennis Müller, Deyan Ginev, Florian Rabe, Jonas Betzendahl, Katja Berčič, Max Rapp, Navid Roux and Tom Wiesing, who have helped me in many different ways and create a great atmosphere for doing research. I also want to thank Kai Amann for implementing most of the GLIF Jupyter kernel and Aarne Ranta and Claudio Sacerdoti Coen for many insightful discussions about GF and theorem proving in ELPI. At last, but by no means least, I would like to thank Varvara for being an amazingly supportive partner throughout this endeavour – and, of course, for diligently proof-reading my thesis.

CHAPTER 2

Preliminaries: Studying Natural Language Semantics

Natural languages are languages like English or Spanish but also, for example, sign languages. They have evolved naturally over a long period of time and there is no normative or descriptive set of rules how the language should be used. It is rather amazing how easily children can learn a language without ever being aware of any language rules. In fact, most adults are at most vaguely aware of grammatical rules. But language is much more than just a set of grammatical rules: by combining the right words, we can communicate complex meanings to other people.

In Section 2.1, we will take a look at some language philosophy to to learn a few tools that help us study the meaning of language. Section 2.2 describes a formal setup for natural language understanding experiments that is the basis of this work. Afterwards, we will discuss the components of an NLU pipeline (Section 2.3). In the remaining sections, we will introduce a number of existing frameworks for language-to-logic translation and take a look at GF, MMT and ELPI – the building blocks of GLIF.

2.1 What is the Meaning of Language?

Philosophers have studied the meaning of language for thousands of years, which has resulted in many different theories of meaning with varying goals and approaches. As any comprehensive introduction is clearly out of scope, we will only briefly touch on a few relevant ideas. Let us consider two simple sentences:

(1) "*Venus is a planet.*"

(2) "*Pluto is a planet.*"

Before we think about the actual meanings of these sentences, we can ask ourselves whether they mean the same thing. Creswell's **most certain principle** [Cre82] states that if one sentence is true and another sentence is false, they must mean different things. Therefore, since the first sentence is true (Venus is indeed a planet) and the second sentence is false (Pluto is – nowadays – only considered a dwarf-planet), they must have different meanings.

A reasonable explanation why the sentences have different meanings is that the expressions "*Venus*" and "*Pluto*" refer to different physical objects. Gottlob Frege called the physical object that a name refers to "Bedeutung", which is commonly translated as **referent**. It would now seem intuitive to say that the meaning of

"*Venus*" is simply its referent (i.e. that piece of matter orbiting the sun). However, Frege already identified a problem with this idea: several expressions may refer to the same object, even though they are not interchangeable. As an example, he used the names of Venus in Ancient Greece: Hesperos and Phosphoros – the morning star and the evening star. Since Venus is on a lower orbit around the sun, it cannot be seen throughout the night and the ancient Greeks did not know that "*Hesperos*" and "*Phosphoros*" refer to the same object (i.e. have the same referent). Therefore, the sentence

"*The ancient Greeks knew that Hesperos is Phosphoros.*"

is false. If we now claim that "*Hesperos*" and "*Phosphoros*" contribute exactly the same meaning to the sentence, we should be able to use them interchangeably. However, the sentence

"*The ancient Greeks knew that Hesperos is Hesperos.*"

is true and therefore has a different meaning. Frege solved this by distinguishing between "Bedeutung" (referent) and "Sinn" (**sense**): While "*Hesperos*" and "*Phosphoros*" have the same referent, they have different senses. That also solves another problem: Some names don't have a referent at all. Think e.g. of "*Odysseus*", who is, presumably, a purely ficticious character. Nevertheless, we can tell meaningful stories about him because "*Odysseus*" has a sense.

We still have not answered the main question: What is the meaning of a sentence? So far, we only mentioned that sentences can be true or false. However, if we define the meaning of sentences to be true or false, sentences like "*Venus is a planet*" and "*Pluto isn't a planet*" would have the same meaning, as they are both true.

To get a better understanding of the meaning of sentences, we will take a look at a very different approach established by Donald Davidson in [Dav67]. According to him, the meaning of a sentence can be identified with its **truth conditions**, i.e. the conditions that must be met to make the sentence true. For example, the truth condition of "*Venus is a planet*" is that Venus is indeed a planet. Davidson noticed that the truth conditions of a sentence are not universal, but depend on the context of its utterance. He illustrates this with the sentence

"*I am tired.*"

where the truth conditions depend on the speaker and the time of the statement. In the next section, we will explore how we can express the truth conditions more formally.

2.2 Formal Semantics

From a mathematical point of view, we can think of a language as the set of "acceptable" utterances in that language. We have to distinguish two fundamentally different types of languages:

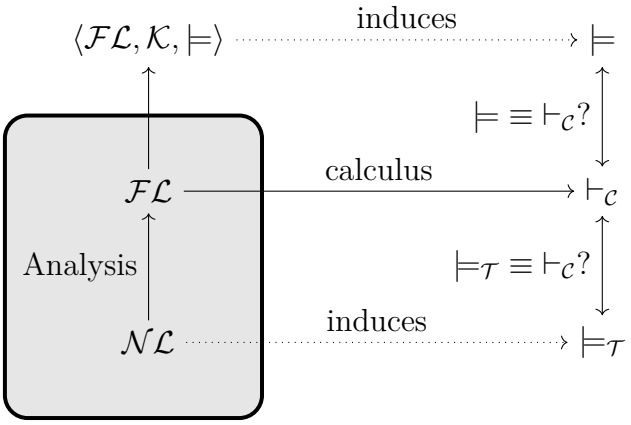


FIGURE 2.1: Inference on different levels (adapted from [LBS]).

- **Formal languages**, which we will denote by $\mathcal{FL}$, have a clearly defined set of rules that state which expressions belong to the language.
- **Natural languages**, denoted by $\mathcal{NL}$, are human languages like English. They have evolved over time and lack any well-defined set of rules. For simplicity, we will nevertheless pretend that there is a well-defined set of acceptable utterances.

So far, we have described the truth conditions of natural language statements using natural language. E.g. "*Venus is a planet*" is true if and only if Venus is a planet. In **formal semantics**, the truth conditions are instead expressed in a formal language $\mathcal{FL}$. For example, the meaning (i.e. truth conditions) of "*Venus is a planet*" might be $planet(venus)$. Research into formal semantics thus focuses on how (parts of) natural language can be translated into $\mathcal{FL}$ expressions. The underlying assumption is that the meaning of $\mathcal{FL}$ expressions is clear, usually because they are expressions in a particular logic that has a formally defined semantics. We can make this more explicit and derive a setting that allows us to treat natural language semantics as a natural science. Figure 2.1 provides an overview of our setup.

Let us assume the formal language $\mathcal{FL}$ belongs to a logical system $\langle \mathcal{FL}, \mathcal{K}, \models \rangle$, where $\mathcal{K}$ is a class of models and $\models \subseteq \mathcal{K} \times \mathcal{FL}$ is a satisfaction relation, indicating which $\mathcal{FL}$ expressions are true in a particular model. The satisfaction relation induces the semantic consequence relation $\models \subseteq \mathcal{P}(\mathcal{FL}) \times \mathcal{FL}$, which describes whether an $\mathcal{FL}$ expression follows from a set of $\mathcal{FL}$ expressions.

The semantic consequence relation allows us to apply the scientific method to natural language understanding. Let us assume we have already made observations and come up with a model (a logical system and the translation into it). We can test the model using **textual entailment**. A text A textually entails a text B (or $A \models_{\mathcal{T}} B$) iff a typical human reader would say that B follows from A. For example

> "*John is happy whenever Mary is happy. Today, Mary is happy because she has a day off.*"

textually entails

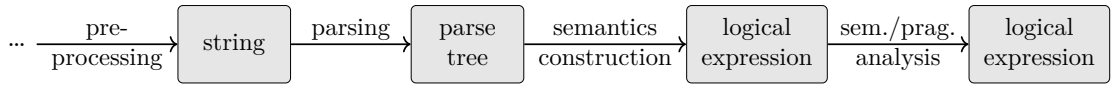


Figure 2.2: The different processing steps in an NLU pipeline.

"*Today, John is happy.*"

To test our model, we can now consider pairs of texts A, B and check whether $A \models_{\mathcal{T}} B$ is equivalent to $\{[\![A]\!]\} \models [\![B]\!]$, where $[\![A]\!]$ and $[\![B]\!]$ are the $\mathcal{FL}$ expressions corresponding to A and B.

In practice, the semantic consequence relation $\models$ is usually replaced with the syntactic consequence relation $\vdash_{\mathcal{C}}$ of some calculus $\mathcal{C}$. If $\mathcal{C}$ is **sound** ($\Gamma \vdash_{\mathcal{C}} \varphi$ implies $\Gamma \models \varphi$) and **complete** ($\Gamma \models \varphi$ implies $\Gamma \vdash_{\mathcal{C}} \varphi$), the two relations are equal. That allows us to check whether $\models_{\mathcal{T}}$ and $\vdash_{\mathcal{C}}$ are equivalent, which is easier than comparing $\models_{\mathcal{T}}$ with $\models$. Figure 2.1 illustrates the different levels of natural language inference we have discussed so far. Much of the natural language semantics research has focused on the highlighted box in Figure 2.1, profiting from the work of logicians on bridging the gap between syntax and semantics.

There is one more aspect to natural language inference that we have completely ignored so far: **world knowledge**. Most human readers would intuitively say that it follows from "*John is happy*" that "*John isn't sad*". In other words "*John is happy*" $\models_{\mathcal{T}}$"*John isn't sad*". From a purely logical point of view though, "*happy*" and "*sad*" are two independent predicates, which means that $\{happy(john)\} \nvdash_{\mathcal{C}} \neg sad(john)$. We can fix this by collecting world knowledge (such as "*if someone is happy, they are not sad*") in a set $\Gamma \subseteq \mathcal{P}(\mathcal{FL})$ and obtain $\Gamma \cup \{happy(john)\} \vdash_{\mathcal{C}} \neg sad(john)$.

2.3 Components of an NLU Pipeline

Translating an entire natural language into a logic that reasonably represents its meaning is extremely hard. It is not even clear what kind of logic would be the right choice. Therefore, researchers often follow the **method of fragments**, established by Richard Montague in [Mon70]: instead of trying to handle all of natural language, they pick a well-behaved **fragment** $\mathcal{F} \subseteq \mathcal{NL}$ of natural language and provide a translation into a (often custom-designed) logic. The goal is to cover ever larger portions of natural language with these fragments.

The translation of a natural language fragment into logical expressions is typically described in two steps: parsing (translating strings into parse trees) and semantics construction (translating parse trees into logical expressions). Sometimes the compositional semantics construction is followed by an inference-based step – the semantic/pragmatic analysis – that provides further processing capabilities. Together, these steps form an **NLU pipeline** (Figure 2.2). In the rest of this section, we will look at each step of an NLU pipeline in more detail.

2.3.1 Preprocessing

This work generally assumes that the natural language we are dealing with is in the form of character strings. We should briefly acknowledge that this is a very strong assumption: language can be spoken, it can occur in a photograph, it can be communicated with sign language, it can be recorded as an audio file, etc. There are ways to convert some of these forms of language into strings – for example, through optical character recognition or speech to text conversion. These technologies might eventually benefit from a deeper language understanding but we will not go into that here. It should also be noted that such preprocessing can result in a loss of useful information about e.g. gestures and intonation. For example, intonation is critical for distinguishing "*I didn't **ride** there*" ("*I walked*") and "*I didn't ride **there***" ("*I rode somewhere else*").

2.3.2 Parsing

After the preprocessing, we need to analyze the resulting string. In a symbolic setting, this is typically done with a **parser**, which transforms a string into a suitable tree representation. The parser is specified in a **grammar**.

Let us say we want to parse sentences like

(1) "*John is happy and Mary is smart.*"

(2) "*John and Mary are happy and smart.*"

Listing 2.3 shows the corresponding grammar in Backus-Naur form. On the highest level, we have sentences, which we will denote as **<S>**. A sentence can be created either by combining two sentences with an "*and*", or by combining a noun phrase (**<NP>**) and an adjective phrase (**<AP>**) with the string "*is*" or "*are*". Sentence (2), for example, has the noun phrase "*John and Mary*" and the adjective phrase "*happy and smart*". Whether we use "*is*" or "*are*" depends on the number of the noun phrase, which we will ignore for now. Later, we will use this distinction as a test case to see how well our grammar mechanisms are suited for the complexity of natural language. Listing 2.3 shows the complete grammar for our example sentences. Noun phrases can be built from proper nouns (**<PN>**) and adjective phrases from adjectives (**<A>**). We will also refer to non-terminal symbols like **<S>** or **<PN>** as **syntactic categories**.

With the grammar we can now parse sentences to obtain parse trees. A **parse tree** represents the syntactic structure of a sentence according to the grammar. Its nodes are labelled with syntactic categories and its leaves with terminal symbols. Figure 2.4 shows the parse tree for sentence (1). An alternative to parse trees are abstract syntax trees, which we will discuss in Section 2.3.4.

2.3.3 Semantics Construction

The **semantics construction** is responsible for translating parse trees into logical expressions. There are many different logics one can choose from, and often a natural

```
<S>  ::= <S> "and" <S> | <NP> "is"/"are" <AP>
<NP> ::= <PN> | <NP> "and" <NP>
<AP> ::= <A> | <AP> "and" <AP>
<PN> ::= "John" | "Mary"
<A>  ::= "happy" | "sad" | "smart"
```

Listing 2.3: A grammar in Backus-Naur form.

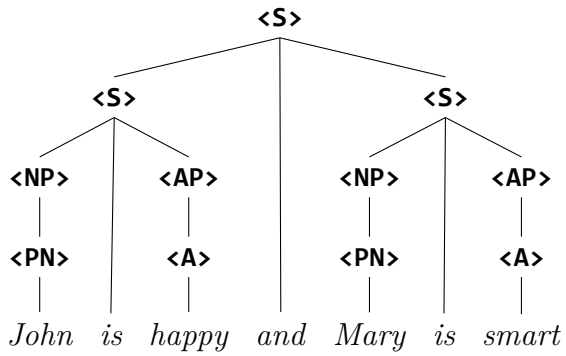


Figure 2.4: Example parse tree.

language semantics paper introduces a new logic that is suited to their particular goals. For our very simple example, the well-known first-order logic suffices. In later sections, we will also briefly look at other logics.

Let us return to sentence (1): "*John is happy and Mary is smart*". It could correspond to the logical expression $happy'(john') \wedge smart'(mary')$. We follow the tradition in linguistics that $john'$ stands for the meaning of "*John*". The wedge ($\wedge$) stands for logical conjunction. Semantics construction has to generate that logical expression from the parse tree in Figure 2.4. Following Montague, we design the semantics construction in a way that it is **compositional**, i.e. the meaning of a tree is computed from the meaning of its subtrees. As a consequence, we can specify the semantics construction by providing rules how each node type should be translated. In our example, the root node combines two subtrees with an "*and*". Let us assume that the meaning of the left subtree ("*John is happy*") is A and that the meaning of the right subtree ("*Mary is smart*") is B. The meaning of the entire tree is then $A \wedge B$. We will write this rule as $[\![A_{\mathsf{S}}\ and\ B_{\mathsf{S}}]\!]_{\mathsf{S}} = [\![A]\!]_{\mathsf{S}} \wedge [\![B]\!]_{\mathsf{S}}$, where $[\![\cdot]\!]_{\mathsf{X}}$ is the (recursively defined) semantics construction function for nodes of type X. For conciseness, we will omit the annotation X when it can be easily inferred.

Ignoring complex noun phrases for a moment, we can use the rule $[\![X_{\mathsf{NP}}\ is\ P_{\mathsf{AP}}]\!]_{\mathsf{S}} = [\![P]\!]([\![X]\!])$ to get the meaning $happy'(john')$ for "*John is happy*". Next, let us take a look at complex adjective phrases. Intuitively, the meaning of "*John is happy and smart*" should be $happy'(john') \wedge smart'(john')$. To remain compositional, we have to somehow define the meaning of "*happy and smart*" independently of "*John*". This is usually solved by introducing λ-expressions, which disappear in the

S_1: $[\![A_{\mathsf{S}} \textit{ and } B_{\mathsf{S}}]\!]_{\mathsf{S}} = [\![A]\!]_{\mathsf{S}} \wedge [\![B]\!]_{\mathsf{S}}$
S_2: $[\![X_{\mathsf{NP}} \textit{ is/are } P_{\mathsf{AP}}]\!]_{\mathsf{S}} = [\![X]\!]([\![P]\!])$
NP_1: $[\![X_{\mathsf{NP}} \textit{ and } Y_{\mathsf{NP}}]\!]_{\mathsf{NP}} = \lambda P.[\![X]\!](P) \wedge [\![Y]\!](P)$
AP_1: $[\![P_{\mathsf{AP}} \textit{ and } Q_{\mathsf{AP}}]\!]_{\mathsf{AP}} = \lambda x.[\![P]\!](x) \wedge [\![Q]\!](x)$

NP_2: $[\![N_{\mathsf{PN}}]\!]_{\mathsf{NP}} = \lambda P.P([\![N]\!])$
AP_2: $[\![P_{\mathsf{A}}]\!]_{\mathsf{AP}} = [\![P]\!]$
PN_1: $[\![T]\!]_{\mathsf{PN}} = T'$
A_1: $[\![T]\!]_{\mathsf{A}} = T'$

Listing 2.5: Semantics construction rules for the grammar in Listing 2.3.

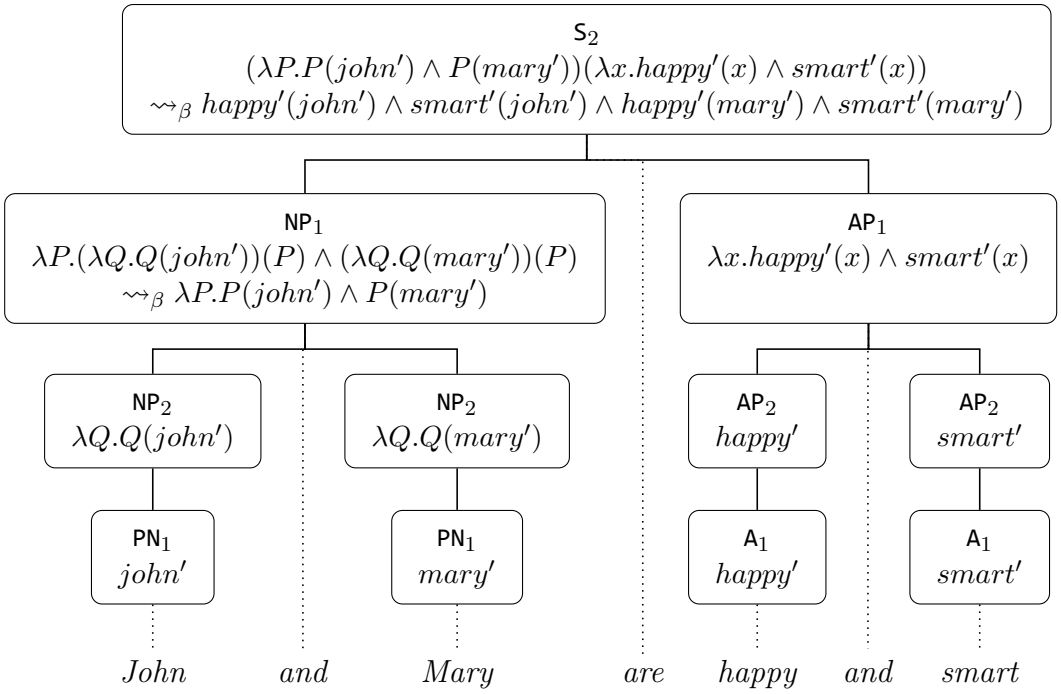


Figure 2.6: Example semantics construction.

final expression through β-reduction. With

$$[\![P_{\mathsf{AP}} \textit{ and } Q_{\mathsf{AP}}]\!]_{\mathsf{AP}} = \lambda x.[\![P]\!](x) \wedge [\![Q]\!](x)$$

the meaning of "*John is happy and smart*" is $(\lambda x.happy'(x) \wedge smart'(x))(john')$, which β-reduces to the desired expression $happy'(john') \wedge smart'(john')$.

To express the meaning of complex noun phrases like "*John and Mary*", we need to use **type-raising**: instead of applying the meaning of the adjective phrase to the meaning of the noun phrase, we do it the other way around. The updated rule is

$$[\![X_{\mathsf{NP}} \textit{ is/are } P_{\mathsf{AP}}]\!]_{\mathsf{S}} = [\![X]\!]([\![P]\!])$$

This allows us to express the meaning of "*John and Mary*" as $\lambda P.P(john') \wedge P(mary')$. The complete set of rules is shown in Listing 2.5. Figure 2.6 illustrates the semantics construction on sentence (2).

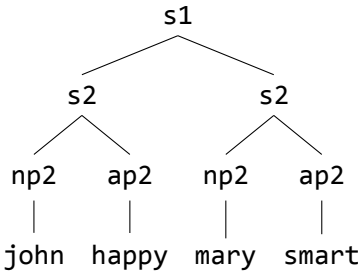


Figure 2.7: Example AST.

2.3.4 An Alternative Specification

There is an alternative way to specify the grammar and semantics construction. Instead of specifying the grammar in Backus-Naur form, we can use named production rules:

```
s1    : <S>  ⟶ <S> "and" <S>
s2    : <S>  ⟶ <NP> "is"/"are" <AP>
np1   : <NP> ⟶ <NP> "and" <NP>
...
happy : <A>  ⟶ "happy"
smart : <A>  ⟶ "smart"
```

From this, we can still generate parse trees like the one in Figure 2.4. However, we can also generate **abstract syntax trees** (short **ASTs**), which describe the rule applications needed to generate the desired string. Figure 2.7 shows the AST corresponding to the parse tree in Figure 2.4. The advantage of ASTs is that it is immediately clear which production rules were used. This could, for example, be of interest when working with homonyms (more precisely, homographs) because we can have different production rules for the different meanings:

```
bank_inst  : <N> ⟶ "bank"
bank_river : <N> ⟶ "bank"
```

The different readings then result in different ASTs (but, of course, the same parse trees).

Fixing the argument order, we can also write the semantics construction differently. Instead of writing e.g.

$$\texttt{NP}_1\text{: } [\![X_{\texttt{NP}}\ and\ Y_{\texttt{NP}}]\!]_{\texttt{NP}} = \lambda P.[\![X]\!](P) \wedge [\![Y]\!](P)$$

as in Listing 2.5, we can now write

$$\texttt{np1} = \lambda X.\lambda Y.\lambda P.X(P) \wedge Y(P),$$

where the first two arguments are the meanings of the constituents.

As we will see in Section 2.4, some frameworks for natural language understanding use specifications closer to the ones shown in the previous sections, while others, including GLF/GLIF, are closer to the approach shown in this section.

2.3.5 Semantic/Pragmatic Analysis

An NLU pipeline should also be able to resolve ambiguities. We can resolve ambiguities syntactically and semantically. A big advantage of the syntactic approach is that linguistic cues, like parallel structures, are available, which can be a very good heuristic. However, syntactic disambiguation means that we loose the opportunity for semantic disambiguation – and some ambiguities can only be resolved semantically. Consider e.g. the following sentence from the Winograd Schema Challenge [LDM12]:

> "*The trophy doesn't fit in the brown suitcase because it's too big.*"

Resolving what "*it*" refers to requires semantic processing. Note that the referent changes if we replace "*too big*" by "*too small*". We will refer to such inferential processing after the semantics construction as **semantic/pragmatic analysis**. It is usually assumed that the semantic/pragmatic analysis involves world knowledge in some form – as our example already shows.

2.4 Existing Frameworks for Language-to-Logic Translation

Historically, NLU pipelines have been implemented in declarative programming languages. In this section we will explore a number of existing frameworks by implementing the running example of Section 2.3, roughly following the grammar rules as shown in Listing 2.3 and the semantics construction from Listing 2.5. The example grammar poses an interesting challenge: noun phrases can be both singular ("*John*") and plural ("*John and Mary*"), which means that we need some mechanism to pick the right form of "*is*"/"*are*" – a useful test to see how suitable the frameworks are for natural language.

First, we will look at the approaches taken in two introductory books on computational semantics. Blackburn and Bos' book "Representation and Inference for Natural Language – A First Course in Computational Semantics" [BB05] uses Prolog for the implementation of NLU pipelines. The book "Computational Semantics with Functional Programming" by van Eijck and Unger [EU10], on the other hand, uses Haskell.

Afterwards, we will try the same thing with the Grammatical Framework GF [Ran11] by Aarne Ranta. Since GLF/GLIF is based on GF, the corresponding section will simultaneously serve as a more detailed introduction to GF.

2.4.1 Blackburn and Bos' Prolog/RINL

In [BB05], Patrick Blackburn and Johan Bos use Prolog as a framework for experimentation with natural language understanding. We refer to their approach as **Prolog/RINL** where RINL is an acronym of the book's title "Representation and Inference for Natural Language".

Parsing is based on Prolog's **definite clause grammars**. In the easiest case, grammar rules can be written like this:

```
s  --> pn, [is], a.
pn --> [john].
pn --> [mary].
a  --> [happy].
```

which is syntactic sugar for Prolog clauses like

```
s(A, Z) :- pn(A, B), [is|C]=B, a(C, Z).
pn([john|Z], Z).
% ...
```

We can use it to parse e.g. "*John is happy*":

```
?- s([john, is, happy], []).
```

By adding more arguments to the rules, we can enforce the agreement of verb and noun phrases. We can also construct a parse tree or (like [BB05]) directly create a semantic expression. Following the conventions of Prolog/RINL, we could have the following rule for combining a noun phrase and an adjective phrase into a statement:

```
s([coord:_, sem:Sem]) -->
    np([num:Num, sem:NP]),
    copula([num:Num]),   % is/are, depending on Num
    ap([coord:_, sem:AP]),
    {combine(s:Sem, [np:NP, ap:AP])}.
```

The argument `sem` tracks the semantic representation and `coord` is used to prevent infinite loops in left-recursive rules like `<S> ::= <S> "and" <S>`. The `combine` predicate is used for the semantics construction. It is decoupled from the grammar rules to allow experimentation with different semantics constructions for the same grammar. In our case, we simply want to apply the meaning of the noun phrase to the meaning of the adjective phrase:

```
combine(s:apply(NP, AP), [np:NP, ap:AP]).
```

For ambiguous sentences, multiple semantic expressions can be generated with Prolog's backtracking. β-reduction requires custom code because Prolog does not natively support variable-binding. After the semantics construction, [BB05] describes the implementation of different inference systems and ends with a number of example dialog systems, where the user can enter sentences and ask questions based on that. Correspondingly, Section 6.2 discusses a case study into simple question answering with GLIF.

The complete example implementation can be found in Appendix A.1.

2.4.2 Van Eijck and Unger's Haskell/CSFP

Very much in the same spirit, Jan van Eijck and Christina Unger use Haskell in [EU10] to experiment with natural language understanding. We refer to their approach as **Haskell/CSFP**. Where Prolog/RINL uses recursive relational techniques, Haskell/CSFP uses recursive functional techniques.

In Haskell/CSFP, a parser is simply a function that converts a list of tokens into a list of parse results. Each parse result is a parse tree along with a list of the remaining tokens:

```
type Parser a b = [a] -> [(b,[a])]
```

We can use **parser combinators** to combine simple parsers into more complex ones. For example, we could have a function `<**>` that applies two parsers in sequence or `<|>` that applies two parsers "in parallel". This allows us to write rules like this one:

```
pS :: Parser String String
pS = (pS <**> symbol "and" <**> pS)
    <|> (pNP <**> symbol "is" <**> pAP)
```

where `symbol "and"` is a parser that only accepts the word "*and*". This parser has several problems: it does not produce meaningful parse trees and the left-recursive rule **`<S> ::= <S> "and" <S>`** results in infinite loops.

Therefore, Haskell/CSFP creates a parse tree and annotates each node with a data object `Cat` that contains information about the syntactic category:

```
data Cat = Cat String CatLabel Agreement
```

For example, the proper noun "*John*" is annotated as `Cat "John" "PN" [Masc,Sg]`. During parsing, the agreement of syntactic features can be checked. For example, the following rule checks whether the syntactic features of the noun phrase and copula agree:

```
sRule :: Parser Cat (ParseTree Cat Cat)
sRule = \ t1 ->
    [ (Branch (Cat "_" "S" []) [np,cop,ap], t4) |
      (np,t2)  <- parseNP True t1,
      (cop,t3) <- parseCop t2,
      (ap,t4)  <- parseAP True t3,
      agree np cop ]
```

Like in Prolog, we can pass Boolean flags to break infinite loops in rules like **`<NP> ::= <NP> "and" <NP>`**.

The target logic can be described in Haskell's data types. Then we define the semantics construction with pattern matching on the parse trees. For λ-expressions we use Haskell's native λ-operator (`\`):

```
abstract Rules = {
  cat
    S; NP; AP; A; PN;
  data
    s1  : S  -> S  -> S;
    s2  : NP -> AP -> S;
    np1 : NP -> NP -> NP;
    np2 : PN -> NP;
    ap1 : AP -> AP -> AP;
    ap2 : A  -> AP;
}
```

```
abstract Lexicon = Rules ** {
  data
    john, mary : PN;
    happy, smart : A;
}

abstract People =
    Rules, Lexicon
```

Listing 2.8: GF abstract syntax for the example grammar.

```
transAP :: ParseTree Cat Cat -> Term -> Proposition
transAP (Branch (Cat _ "AP" _) [p1,and,p2]) =
    \ t -> And (transAP p1 t) (transAP p2 t)
transAP (Branch (Cat _ "AP" _) [adj]) =
    transAdj adj
```

The complete example implementation can be found in Appendix A.2.

2.4.3 The Grammatical Framework GF

The **Grammatical Framework** (**GF**) [Ran11; GF] is a tool for developing multilingual grammar applications. Unlike Prolog and Haskell, GF is not a general-purpose programming language, which is, of course, a restriction. However, being a dedicated framework, GF offers powerful mechanisms for grammar development. A GF grammar consists of an **abstract syntax** and a set of **concrete syntaxes**. The abstract syntax uses type theory to describe the abstract syntax trees (ASTs) covered by the grammar. It contains declarations of type constants, which represent syntactic categories, and function constants, which represent rules. The ASTs are objects constructed from the function constants. For more details on the type-theoretical aspects of GF see [Ran94]. Concrete syntaxes describe the **linearization** of ASTs, i.e. the mapping from ASTs to strings in a particular language. The grammar formalism has been carefully designed such that GF can also generate a parser from the concrete syntax.

Example Abstract Syntax

Listing 2.8 shows the abstract syntax for our example grammar (Listing 2.3). It is split into three modules. The `Rules` module introduces the syntactic categories with the keyword **`cat`**. It then declares the rules for combining different categories with the keyword **`data`**[1]. For example, the rule

[1] If you have used GF before, you might expect the keyword **`fun`**, which would work just as well for parsing. However, the pattern matching for the semantics construction requires us to use **`data`**.

```
concrete RulesEng of Rules = {
  param Number = Sg | Pl;
  lincat
    S=Str; AP=Str; A=Str; PN=Str;
    NP = {s : Str; n : Number};
  oper
    to_be : Number => Str =
        table {Sg=>"is"; Pl=>"are"};
  lin
    s1 a b   = a ++ "and" ++ b;
    s2 np ap = np.s++to_be!np.n++ap;
    np1 a b  = {s = a.s++"and"++b.s;
                n = Pl};
    np2 pn   = {s = pn; n = Sg};
    ap1 a b  = a ++ "and" ++ b;
    ap2 adj  = adj;
}
```

```
concrete RulesRgl of Rules =
        open SyntaxEng in {
  lincat
    S = S; NP = NP; AP = AP;
    A = A; PN = PN;
  lin
    s1 a b = mkS and_Conj a b;
    s2 np ap =
        mkS (mkCl np (mkVP ap));
    np1 a b = mkNP and_Conj a b;
    np2 pn  = mkNP pn;
    ap1 a b = mkAP and_Conj a b;
    ap2 adj = mkAP adj;
}
```

Listing 2.9: Concrete syntax. On the right using GF's Resource Grammar Library.

```
s2 : NP -> AP -> S;
```

combines a noun phrase (`NP`) and an adjective phrase (`AP`) into a sentence (`S`). We will generally think of the syntactic categories as type constants and of the rules as function constants. The `Lexicon` module extends the `Rules` module with a small vocabulary. The `People` module combines the modules into one module. This separation allows us to experiment e.g. with different lexica in a modular way. With the abstract syntax in place, we can e.g. describe the AST corresponding to "*John is happy*" with the expression `s2 (np2 john) (ap2 happy)`. Of course, we still need a concrete syntax to define the mapping between strings and ASTs.

Example Concrete Syntax

Listing 2.9 shows a concrete syntax `RulesEng` for `Rules`. It describes the linearization of ASTs into strings of the English language and illustrates some of the mechanisms GF provides for managing the syntax of natural language. First, we map the syntactic categories to concrete types (keyword **`lincat`**). For this simple example, we can use strings for every category except for `NP`, which is a record type containing not just the string representation, but also a field `n` indicating the number of the noun phrase. The rule linearizations now have to describe how the argument objects should be combined. In the case of rule `s1` it is a simple string concatenation with the operator `++`. For rule `s2`, we pick the correct form of "*is*"/"*are*" based on the number of the noun phrase (`np.n`) by looking it up in the table `to_be`.

Using the Grammar

With the grammar in place, we can linearize ASTs in the GF shell:

```
> linearize s2 (np2 john) (ap2 happy)
John is happy
```

As mentioned earlier, GF also generates a parser:

```
> parse "John and Mary are happy and smart"
s2 (np1 (np2 john) (np2 mary)) (ap1 (ap2 happy) (ap2 smart))
> parse "John is happy and happy and happy"
s2 (np2 john) (ap1 (ap2 happy) (ap1 (ap2 happy) (ap2 happy)))
s2 (np2 john) (ap1 (ap1 (ap2 happy) (ap2 happy)) (ap2 happy))
```

Note that GF generates two ASTs for the second example due to its syntactic ambiguity. One of the traditional applications of GF is high-precision machine translation. If we add e.g. a German concrete syntax, we can use GF to translate sentences between German and English:

```
> parse -lang=Eng "Mary is smart"
makeStmt mary smart
> linearize -lang=Ger makeStmt mary smart
Maria ist klug
```

Summary: Grammar Development with GF

What truly distinguishes GF from most parser frameworks is its ability to handle natural language. On one hand, its grammar formalism is very powerful. It supports parallel multiple context free grammars (PMCFG), which are more expressive than the more well-known context free grammars[2] [Lju04]. On the other hand, GF has a sophisticated type system for the concrete syntaxes, some of which we have seen in Listing 2.9. This makes it much easier to model the complex syntax (and morphology) of natural language.

GF supports the development of large grammars through mechanisms for abstraction and modularization. This also makes it easier to reuse parts of a grammar. Since every natural language grammar will have to implement the morphology of a language, the GF community has developed the **Resource Grammar Library**, which implements the basic syntax and morphology of currently 36 languages [RGL]. Listing 2.9 shows on the right-hand side how we can use it for our example.

Semantics Construction with GF

While GF primarily focuses on grammar development (and that is what GLF/GLIF uses it for), we can in principle also use it to implement the semantics construction. There are two ways one can go about it. The obvious way is to create another

[2]It may be note-worthy that Dutch and Swiss German have been shown to have cross-serial dependencies, which theoretically cannot be expressed in a context-free grammar [Bre+82; Shi85]

```
abstract Logic = {
  cat Prop; Term;
  fun
    Not : Prop -> Prop;
    And : Prop -> Prop -> Prop;
}
```

```
abstract DDT = Logic ** {
  fun
    j,m : Term;  -- John, Mary
    h,s : Term -> Prop;
}
```

Listing 2.10: Logic and discourse domain theory (DDT) in GF.

concrete syntax that generates strings in the formal language of some logic. However, this can be a somewhat error-prone process because GF can't type-check if the generated strings correspond to well-formed logical expressions. The more elegant way is to use the type-theoretical framework underlying the abstract syntaxes to specify a logic (Listing 2.10). Aside from the logic syntax, we also need a **discourse domain theory** that introduces some symbols corresponding to the lexicon entries. We will discuss the role of the discourse domain theory in more detail in Section 3.2. Having specified the semantic representation, we can now define (keyword **def**) the semantics construction as a function in basically the same way as we did in Haskell/CSFP:

```
fun
  transNP : NP -> (Term -> Prop) -> Prop;
def
  transNP (np1 a b) = \p -> And (transNP a p) (transNP b p);
  transNP (np2 pn) = \p -> p (transPN pn);
```

The complete code is in Appendix A.3. For a comprehensive introduction to GF, the reader may be interested in the GF book [Ran11] or the online tutorial [GFT10].

2.5 Logic Development with MMT

This section introduces MMT, which will serve as the logic development component of GLF/GLIF.

MMT [MMTb] is both a language and a system. The MMT language (**M**eta **M**eta **T**heories) aims to be a scalable, foundation-independent format for formal knowledge representation. It is based on the OMDoc (**O**pen **M**athematical **Doc**uments) format [Koh06]. The MMT system (**M**eta **M**eta **T**ool Set) provides a variety of formal knowledge management services based on the MMT language. A detailed explanation of the MMT format can be found at [RK13]. For more advanced examples how MMT can be used to implement formal systems, the reader may be interested in [MR19].

MMT content is represented as theories connected by theory morphisms. A **theory** is a sequence of symbol declarations and a **theory morphism** maps symbols declared in one theory to objects in another theory. This forms the **category of theories and theory morphisms**. We will see some examples in the next sections.

```
theory proplog : ur:?LF =
  proposition : type | # o |
  not : o⟶o     | # ¬ 1 prec 80    |
  and : o⟶o⟶o | # 1 ∧ 2 prec 60 |
  or  : o⟶o⟶o | # 1 ∨ 2 prec 50 |
      = [a,b] ¬ (¬ a ∧ ¬ b) |
  // etc. | |
```

```
theory plnq : ur:?LF =
  include ?proplog |
  individuals : type | # ι |
|
```

Listing 2.11: Syntax of propositional logic and its extension `plnq` in MMT.

Knowledge can be represented in logical systems. Logical systems, in turn, can be represented in logical frameworks. Since MMT avoids committing to a particular logical framework, we call it **foundation independent**. This foundation independence makes MMT a great logic development tool. In practice, however, the Edinburgh Logical Framework **LF** [HHP93] and extensions of it are used most often.

2.5.1 Logic Syntax in MMT Theories

To get a first impression of MMT, we will implement the syntax of predicate logic without quantifiers (`plnq`) in MMT (Listing 2.11). Instead of OMDoc, we use the compact **MMT surface syntax** throughout this thesis. Usually, MMT users are recommended to follow the "little theories" approach and keep individual theories as small as possible to maximize reusability. For brevity, we will use only two theories to describe `plnq`. The first theory, `proplog`, describes the syntax of propositional logic and the second theory, `plnq`, then extends it with a type for individuals.

Every MMT theory is based on another MMT theory, which we then call its **meta theory**. The only exception are a few built-in theories called **urtheories** that don't have a meta theory. The meta theory of `proplog` is LF (`ur:?LF`), which happens to be an urtheory. Effectively, the symbols from the meta theory are simply copied into the theory. LF provides, for example, the symbols *`type`* and $\longrightarrow$. The meta theory relation is an example of a theory morphism.

The theory body contains **declarations**. A typical MMT declaration is of the form

```
SYMBOL : TYPE | = DEFINITION | # NOTATION |
```

but not all of the components have to be provided. The vertical bars (`|`, `|`) act as delimiters. The first declaration in `proplog` introduces the symbol `proposition` as a type with the notation `o`. Next, we declare `not` as a unary operation on propositions. We can place arguments in the notation using integers. The first (and in this case only) argument is denoted by `1`. Setting a precedence (keyword *`prec`*) reduces the number of parentheses we will need in the future. Afterwards, we can declare `and` and `or` as binary operations with infix notation and provide a definition for `or` using De Morgan's law. The square brackets are used for lambda notation, which is also provided by LF. For example, $\lambda x.M$ is written as `[x] M` and $\lambda x.\lambda y.M$ can be

abbreviated as `[x,y] M`. Appendix C provides an overview of the notations used in different systems.

The theory `plnq` extends `proplog` with a type for individuals. With the **`include`** keyword, all symbols from `proplog` are copied into `plnq`. Inclusion is another type of theory morphism, similar to the meta theory morphism described above.

At last, we can implement first-order logic as an extension of `plnq`. To represent the quantifiers, we use **higher-order abstract syntax**. The idea is that we can represent a logical formula φ that contains a free variable x as a function $P := \lambda x.\varphi$. Then we can replace the notation $\forall x.\varphi$ or $\forall x.P(x)$ with the notation $\forall P$ or, equivalently, $\forall \lambda x.P(x)$. Here is the complete theory for first-order logic:

```
theory fol : ur:?LF =
  include ?plnq |
  forall : (ι→o)→o | # ∀ 1 |
  exists : (ι→o)→o | # ∃ 1 | = [p] ¬∀[x]¬(p x) |
|
```

2.5.2 MMT Views

We have seen two types of theory morphisms so far: the meta theory relation and inclusions. Another type of theory morphism are views. A **view** maps all undefined symbols in the source theory to terms in the target theory, which induces a mapping from terms in the source theory to terms in the target theory. Later, we will use views for the semantics construction. For now, we will use a view to define the semantics of propositional logic.

We will interpret a proposition as the set of variable assignments that makes the proposition true. The target of the view will be a theory `proplogModel`, which is based on a typed set theory from the **MitM** library [Mit] and introduces a new type for variable assignments:

```
theory proplogModel : typedsets:?AllSets =
  assignments : type | # A |
|
```

In the view we now map propositions to sets of variable assignments. Conjunction corresponds to set intersection and negation to taking the complement:

```
view proplogSem : ?proplog -> ?proplogModel =
  proposition = set A |
  not = [φ] fullset A \ φ |
  and = [φ, ψ] φ ∩ ψ |
|
```

Note that the theory `proplog` also introduces the symbol `or` (see Listing 2.11), which should be mapped to set union. However, we do not have to define it in the view because `or` was defined in terms of `and` and `not` (which are covered by the view).

$$\frac{A \quad B}{A \wedge B}\wedge I \qquad \frac{A \wedge B}{A}\wedge E_l \qquad \frac{A \wedge B}{B}\wedge E_r$$

$$\frac{\begin{array}{c}[A]^1\\ \vdots\\ B\end{array}}{A \Rightarrow B}\Rightarrow I^1 \qquad \frac{A \vee B \quad \begin{array}{c}[A]^1\\ \vdots\\ C\end{array} \quad \begin{array}{c}[B]^1\\ \vdots\\ C\end{array}}{C}\vee E^1$$

Figure 2.12: Some natural deduction rules for propositional logic.

2.5.3 Natural Deduction Calculus in MMT

Apart from the syntax and semantics of a logic, we can also describe calculi in MMT. Specifically, we will describe a natural deduction calculus. Figure 2.12 shows some of the rules we will implement. First we need a judgment indicating that a proposition is true. Following the **judgments-as-types** approach, we introduce a new constant `ded` that maps propositions to types:

```
ded : o → type | # ⊢ 1 |
```

We think of `⊢A` (the alternative notation for `ded A`) as the type of all proofs of proposition `A`. The $\wedge I$ (conjunction introduction) rule states that we can construct a proof of `A∧B` if we have a proof of `A` and a proof of `B`. We can formalize this in MMT with the following declaration:

```
andI : {A:o} {B:o} ⊢A → ⊢B → ⊢A∧B |
```

where `{A:o}` is the MMT notation for the dependent type operator $\prod_{A:o}$[3]. Using a short-hand notation, we can rewrite `{A:o} {B:o}` as `{A,B}`. The types can be omitted because MMT can infer them.

To prove an implication $A \Rightarrow B$ with $\Rightarrow I$ (implication introduction), we need to have a way of transforming a proof of A into a proof of B, i.e. we need a function $\vdash A \to \vdash B$. We can therefore formalize $\Rightarrow I$ in MMT with

```
implI : {A,B} (⊢A → ⊢B) → ⊢A⇒B |
```

Listing 2.13 shows the implementation of other natural deduction rules in MMT.

2.5.4 Theorem Proving in MMT

Let us introduce some propositions to see how we can use the natural deduction rules:

```
propA : o |  propB : o |
```

To turn a proposition φ into an axiom, we need to state that the type of its proofs $\vdash \varphi$ is not empty. We do so by declaring a symbol of type $\vdash \varphi$:

[3] We can get a different (maybe more intuitive) perspective by applying the Curry-Howard isomorphism, giving us $\forall A.\forall B.\vdash A \Rightarrow \vdash B \Rightarrow \vdash (A \wedge B)$.

```
theory proplogND : ur:?LF =
  include ?proplog |
  ded : o → type | # ⊢ 1 prec 10 | role Judgment |
  andI  : {A,B}   ⊢A → ⊢B → ⊢A∧B |
  andEl : {A,B}   ⊢A∧B → ⊢A |
  andEr : {A,B}   ⊢A∧B → ⊢B |
  implI : {A,B}   (⊢A→⊢B) → ⊢A⇒B |
  orE   : {A,B,C} ⊢A∨B → (⊢A→⊢C) → (⊢B→⊢C) → ⊢C |
  // etc. |

theory folND : ur:?LF =
  include ?fol |
  include ?proplogND |
  forallI : {P} ({X} ⊢P X) → ⊢∀P |
  forallE : {P,X} ⊢∀P → ⊢P X |
  existsI : {P,X} ⊢P X → ⊢∃P |
  existsE : {P,C} ⊢∃P → ({X} ⊢P X → ⊢C) → ⊢C |
```

Listing 2.13: Natural deduction rules in MMT.

```
anAxiom : ⊢ propA ∧ propB |
```

For theorems, we also need to provide a proof, i.e. a term that has the right type:

```
aTheorem : ⊢ propA | = andEl propA propB anAxiom |
```

If we had introduced a notation for `andEl` that ignores the first two arguments (i.e. `# ∧El 3`), MMT could recover the missing arguments and the proof shortens to `∧El anAxiom`. To have an example with hypotheses, we can prove the derived rule $\vdash A \Rightarrow (B \Rightarrow A)$:

```
K : {A,B} ⊢A⇒(B⇒A) |
  = [A,B] implI A (B⇒A) ([a:⊢A] implI B A ([b:⊢B] a)) |
```

In Section 4.3.5 we will see how we can use MMT views to verify the soundness and completeness of a calculus relative to another calculus.

Theory Graphs

As the connections between various theories can get rather complicated, we will often visualize them as **theory graphs** (diagrams in the category of theories and theory morphisms). The different types of theory morphisms are represented with different arrows:

- Inclusions are represented by hooked arrows: $\hookrightarrow$
- Meta-theory morphisms are dotted arrows: $\dashrightarrow$
- Views may be annotated with their name and use squiggly arrows: $\rightsquigarrow$

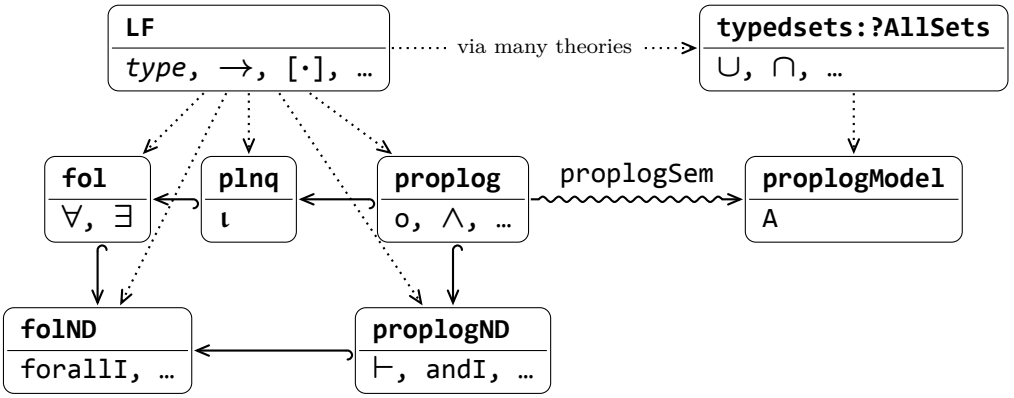


Figure 2.14: Theory graph of the theories used in this section.

Figure 2.14 shows a theory graph containing the theories we have covered so far. It hints already at the idea that logics can be developed in a modular way. That idea has led to the **LATIN** project [Cod+11] (**L**ogic **At**las and **In**tegrator), which implemented many logics in LF in a highly modular fashion. LATIN2 [LATIN] is a reimplementation of LATIN in MMT surface syntax. The content of LATIN/LATIN2 can be represented as a theory graph. Since theories are kept as small as possible to maximise reusability/modularity, that theory graph is very complex. For example, each logical connective is introduced in a separate theory. Since theory graphs are useful to explore e.g. LATIN, specialized tools like TGView3D [MKR20] have been developed.

2.6 Inference with ELPI via Helper Predicates

Disclaimer *This section was heavily influenced by an example implementation by Claudio Sacerdoti Coen [Elp], which in turn was inspired by an approach used in the ProofCert project e.g. in [CMR13]. A brief overview of theorem proving with helper predicates is also given in [Koh+20b]. I am not aware of any incremental introduction to the topic like the one presented here.*

This section introduces ELPI, which will become the inference component in GLIF for the semantic/pragmatic analysis. **ELPI** [SCT15; Dun+15] is an implementation of **λProlog** extended with constraint handling rules. λProlog, in turn, is an extension of Prolog. It is typed and supports λ-terms, which enables us to use higher-order abstract syntax. ELPI is advertised as a platform for implementing the core components of interactive theorem provers – in particular the elaborator – in an extensible fashion (e.g. [GSCT19]). A detailed introduction to λProlog can be found at [MN12].

In this section, we will explore how ELPI (or λProlog in general) can be used for the implementation of natural deduction provers. This is inspired by an approach of the ProofCert project e.g. in [CMR13]. Our long-term goal is to automatically

generate theorem provers from the MMT specification of a natural deduction calculus (see Section 4.2).

2.6.1 First, Naive Attempt

The first thing we need is a predicate `ded` that checks whether a proposition (type `o`) can be proven:

```
type ded o -> prop.
```

We can use `ded` to express natural deduction rules. As an example, we will look at the rules for conjunction (Figure 2.12):

```
ded (and A B) :- ded A, ded B.     % ∧I
ded A         :- ded (and A B).    % ∧E_l
ded B         :- ded (and A B).    % ∧E_r
```

The rule for $\wedge I$ states that $A \wedge B$ is provable if both A and B can be proven, which is exactly what we would expect. The next rule is more problematic: the proposition A is provable if we can prove the proposition $A \wedge B$, where B is a new variable that can be instantiated by the ELPI engine later on. From this rule alone, we can run into infinite loops. So while the implementation of this calculus is sound, it certainly is very far from being complete.

2.6.2 Helper Predicates and Proof Terms

To guide λProlog's depth-first search, we use **helper predicates**, which act as guards that determine whether a rule should be used. The helper predicates do not affect the soundness of a prover, which means that the soundness of a prover can be established by only checking that the rules of the deduction system were implemented correctly. In the ProofCert project, this idea is used for the verification of proof certificates. A proof certificate is the output of some optimized theorem prover, which somehow describes how a theorem was proven. To verify it, the (trusted) λProlog implementation of the deduction system can be extended with helper predicates that guide the prover based on the certificate. This is e.g. described in [CMR13]. While we are not interested in proof verification, it turns out that this setup is very flexible and can be used to create simple automated theorem provers. Some of the code can be generated automatically (see Section 4.2), making it a very attractive option for GLIF as it facilitates quick prototyping. The disadvantage is that the provers may be rather slow.

Coming back to our example, we can e.g. use the helper predicates and proof certificates (here mere integers) to enforce a depth limit to the search:

```
type helper int -> int -> prop.
helper N N1 :- N > 0, N1 is N - 1.
```

```
kind cert type.
type ded cert -> o -> prop.
% rule: ∧I
type helper/andI cert -> o -> o -> cert -> cert -> prop.
ded X (and A B) :- helper/andI X A B X1 X2,
    ded X1 A, ded X2 B.
% rule: ∧E_l
type helper/andEl cert -> o -> o -> cert -> prop.
ded X A :- helper/andEl X A B X1, ded X1 (and A B).
```

Listing 2.15: A generic implementation allowing for arbitrary helper predicates.

```
type ded int -> o -> prop.
ded N (and A B) :- helper N N1, ded N1 A, ded N1 B.
ded N A         :- helper N N1, ded N1 (and A B).
ded N B         :- helper N N1, ded N1 (and A B).
```

The provability predicate `ded` now always has to pass an integer around, which will be used by `helper` to enforce the depth limit. We will refer to such information that gets exchanged with the helper predicates as **proof certificates**.

In the example above, the type of the proof certificate and the helper predicate were hard-coded. We can make the approach much more flexible by creating a new type for proof certificates (`cert`) and picking the helper predicate based on the provided certificate. That allows us to support different strategies based on different certificates. To make everything as general as possible, we need to

- have a separate helper predicate for each rule,
- provide the helper predicates with information on the formulae involved,
- and let them generate different proof certificates for each subsequent call to `ded`.

Listing 2.15 shows an implementation of `ded` based on these ideas. It provides the signatures of helper predicates for each rule. We can now implement the helper predicate that limits the search depth like this:

```
type idcert int -> cert.
helper/andI (idcert N) _ _ (idcert N1) (idcert N1) :-
                                        N > 0, N1 is N - 1.
```

with similar implementations for the other rules. The certificate is called `idcert` (short for **iterative deepening certificate**) because we will use it for iterative deepening. There are other types of proof certificates we may be interested in. For example, we can record the generated proof with a **proof term certificate**:

```
kind pt type.  % proof term
type ptcert pt -> cert.
type andI pt -> pt -> pt.
helper/andI (ptcert (andI X Y)) _ _ (ptcert X) (ptcert Y).
```

While the iterative deepening certificate hands down integers, the proof term certificate hands down variables, which get instantiated by the ELPI engine at a later point. What helper predicate is used, is determined by ELPI via pattern matching on the certificate.

Being able to generate proof terms is useless if the prover immediately gets stuck in an infinite loop. We can use **product certificates** to combine different certificates – like `idcert` for guiding the search and `ptcert` for recording the results:

```
type prodcert cert -> cert -> cert.
helper/andI
    (prodcert X Y) A B (prodcert X1 Y1) (prodcert X2 Y2) :-
        helper/andI X A B X1 X2, helper/andI Y A B Y1 Y2.
```

For example, the certificate `prodcert (idcert 5) (ptcert X)` runs the proof search with a depth-limit of 5 and records a proof term in the variable `X`.

2.6.3 Simple Hypotheses

At this point our prover still can't prove anything at all (unless we add the law of excluded middle), because we don't handle hypotheses. Figure 2.12 shows the natural deduction rule $\Rightarrow I$, which requires the use of hypotheses. Ignoring proof certificates and helper predicates for a moment, we can write the $\Rightarrow I$ rule using the `=>` operator:

```
ded (impl X Y) :- ded X => ded Y.
```

`ded X` is temporarily added to the context as an axiom. This rule alone is sufficient to prove tautologies like $\varphi \Rightarrow \varphi$ or $\varphi \Rightarrow (\psi \Rightarrow \varphi)$.

In general, however, it is desirable to separate hypotheses from provable propositions. Therefore, we introduce the predicate `ded/hyp` to track hypotheses. `ded` can recover hypotheses with the following rule:

```
type ded/hyp o -> prop.
type helper/hyp cert -> o -> prop.
ded X A :- ded/hyp A, helper/hyp X A.
```

`helper/hyp` allows us e.g. to include the use of hypotheses in the proof term certificates. We can now write $\Rightarrow I$ and $\vee E$ (see Figure 2.12) as

```
type helper/implI cert -> o -> o -> cert -> prop.
ded X (impl A B) :-
    helper/implI X A B X1, ded/hyp A => ded X1 B.

type helper/orE
    cert -> o -> o -> o -> cert -> cert -> cert -> prop.
ded X C :- helper/orE X A B C X1 X2 X3, ded X1 (or A B),
    ded/hyp A => ded X2 C, ded/hyp B => ded X3 C.
```

Once the necessary helper predicates are implemented, we have a small theorem prover. For example, the query

```
ded (prodcert (idcert 2) (ptcert X)) (impl (and a b) a)
```

returns

```
X = implI (and a b) (andEl (usehyp (and a b)))
```

Of course, here we have hard-coded the search depth 2. In general, we want to have controller code that iteratively increments the search depth.

2.6.4 Tracking Hypotheses

In a way the hypothesis handling described in the previous section works perfectly well: hypotheses can be added to the context and are only retrievable in the right places. However, it is unclear where a hypothesis came from when we use it, which is information we might want to include in our proof terms. Here is a sketch of how we can track hypotheses:

```
ded X A :- ded/hyp N A, X = N.
ded X (impl A B) :- helper X A B X1,
                    pi x \ ded/hyp x A => ded (X1 x) B.
helper (implI A B X) A B X. %X maps proofs of A to proofs of B
```

With this, the query `ded X (impl a a)` results in

```
X = implI a a c0 \ c0
```

where `x \ M` is ELPI's notation for $\lambda x.M$ (see also Appendix C). Similarly, the query `ded K (impl x (impl y x))` results in

```
K = implI x (impl y x) c0 \ implI y x c1 \ c0
```

Note that this is equivalent to the proof of the derived rule `⊢X⇒(Y⇒X)` in Section 2.5.3:

```
K : {X,Y} ⊢X⇒(Y⇒X) |
  = [X,Y] implI X (Y⇒X) ([x:⊢X] implI Y X ([y:⊢Y] x)) |
```

2.7 Summary

We started this chapter with an introduction to formal semantics, in an attempt to justify the use of logic to represent the meaning of language and to establish a setting for applying the scientific method to natural language understanding. Afterwards, we looked at how an NLU pipeline can be described in three steps: parsing, semantics construction and semantic/pragmatic analysis. The first two steps correspond to the traditional Montagovian approach to natural language semantics. We then looked at how a simple Montague fragment can be implemented in different frameworks, including GF, which we will use for parsing in GLF/GLIF. We finished the chapter by looking at the two remaining components we will need for GLIF: MMT for the logic development (and semantics construction) and ELPI for the semantic/pragmatic analysis. Specifically, we explored how natural deduction provers can be implemented in ELPI using helper predicates, paving the way for automated, logic-independent prover generation (Section 4.2), which could speed up the prototyping of the semantic/pragmatic analysis.

CHAPTER 3

Grammatical Logical Framework (GLF)

Disclaimer *The idea behind GLF has been published in [KS]. It was developed with my supervisor, Michael Kohlhase, and is based on an early prototype by Dennis Müller. My contribution is the concrete design, as well as coming up with examples to illustrate and test the functionality of GLF. Figure 3.1 is based on a figure that I used for presenting [KS] at LFMTP 2019.*

The **Grammatical Logical Framework** (**GLF**) combines the grammar development capabilities of GF with the logic development capabilities of MMT. Figure 3.1 illustrates the GLF pipeline: we use GF for parsing and MMT for semantics construction. We have already described how to use GF for parsing (Section 2.4.3) and how to use MMT for developing the target logic (Section 2.5). In the next sections we will discuss how we can connect GF and MMT by exploiting the similarity of the underlying logical frameworks and how to do the semantics construction in MMT.

3.1 From GF to MMT

The key idea is that we can represent GF abstract syntax modules as LF-based MMT theories. We refer to such theories as **language theories**. Listing 3.2 shows an abstract syntax and the corresponding language theory in MMT surface syntax. GF categories correspond to type constants and GF functions to function constants. Thanks to this similarity, we can reinterpret a GF AST (e.g. `s2 (np2 mary) (ap2 smart)`) as an MMT term in the corresponding language theory. The AST and the MMT term even have the same string representation: `s2 (np2 mary) (ap2 smart)`.

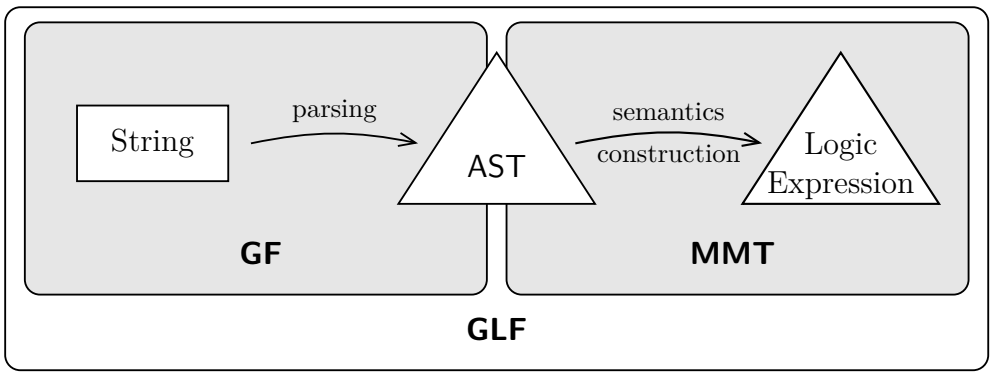


FIGURE 3.1: The GLF pipeline.

```
abstract Example = People ** {
  cat
    V;  -- verbs
  fun
    s3 : NP -> V -> S;
    run : V;
}
```

```
theory Example : ur:?LF =
  include ?People ❙
  V : type ❙

  s3 : NP → V → S ❙
  run : V ❙
❙
```

Listing 3.2: GF abstract syntax and corresponding language theory.

```
theory Rules : ur:?LF =
  S  : type ❙
  NP : type ❙
  // etc. ❙
  s1  : S  → S  → S  ❙
  s2  : NP → AP → S  ❙
  np1 : NP → NP → NP ❙
  // etc. ❙ ❙
```

```
view RulesSem : ?Rules -> ?plnq =
  S   = o ❙
  NP  = (ι→o)→o ❙
  // etc. ❙
  s1  = [a,b] a ∧ b ❙
  s2  = [np,ap] np ap ❙
  np1 = [a,b] [p] (a p) ∧ (b p) ❙
  // etc. ❙ ❙
```

Listing 3.3: Language theory and semantics construction in MMT.

Aside from the type-theoretical compatibility, the modular structures of GF and MMT are also compatible. The abstract syntax module in Listing 3.2 extends the `People` module, which corresponds to including the `People` theory on the MMT side. This gives us parallel dependency graphs for the abstract syntax and the language theory (and also for each concrete syntax; see also Figure 3.5).

Since the language theory can be generated automatically from the GF abstract syntax, it does not need to be specified in MMT surface syntax. For simplicity, we often won't distinguish between an abstract syntax and the language theory generated from it. More details on the functionality and implementation of GLF (and GLIF) are provided in Chapter 5.

3.2 Semantics Construction in MMT

We can implement the semantics construction as a view from the language theory into a theory describing the target representation. For the grammatical rules, we can describe the semantics construction as a view into `plnq`, which was defined in Listing 2.11. Listing 3.3 shows the (usually automatically generated) language theory `Rules` and the semantics construction. Recall that `[•]` is the MMT notation for λ-abstraction. We follow exactly the approach described in Section 2.3.4.

For the rest of the semantics construction, `plnq` does not suffice because we also need predicates and constants for words like "*happy*" and "*John*". Therefore, we have to introduce the needed symbols in a new theory. We will refer to such a theory as **discourse domain theory** (or **DDT**). Aside from new symbols, a discourse domain theory can also introduce world knowledge or state situation-specific axioms. That

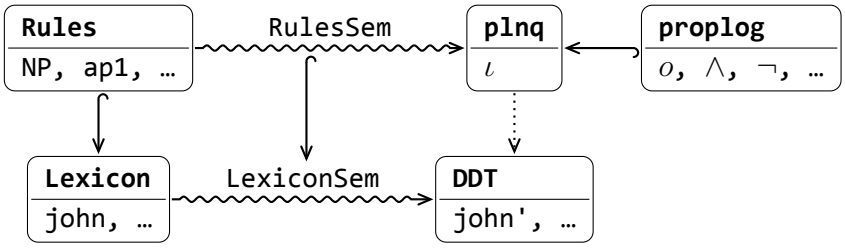


FIGURE 3.4: Theory graph showing semantics construction in MMT.

requires us to import a judgment like $\vdash$ (see also Section 2.5). For our example, we will only introduce predicates and constants in the discourse domain theory, using `plnq` as meta theory:

```
theory DDT : ?plnq =
  john  : ι | # john' |
  // etc. |  |
```

In Section 2.3.3 we used apostrophes to denote the meaning of words (e.g. the meaning of "*happy*" is *happy*′). We can emulate that convention using MMT notations. The semantics construction view for the lexicon entries is straight-forward. However, `Lexicon` also imports all symbols from `Rules`, so we need to map these symbols as well. We can simply include the view `RulesSem`:

```
view LexiconSem : ?Lexicon -> ?DDT =
  include ?RulesSem |
  john = john' |
  // ... |
|
```

Figure 3.4 shows all theories and morphisms involved in the semantics construction.

3.3 Testing the Pipeline

With all this in place, we now parse the sentence "*John and Mary are happy*" to obtain the AST

```
s2 (np1 (np2 john) (np2 mary)) (ap2 happy).
```

If we apply the semantics construction view and β-reduce the resulting expression, we obtain

```
(happy' john')∧(happy' mary')
```

We can collect the content of a discourse in a theory. So if the sentence above occurs in a discourse, we can add the declaration:

```
sentence1 : ⊢ (happy' john')∧(happy' mary') |
```

Now let us perform a small test of our model as outlined in Section 2.2. "*John and Mary are happy*" textually entails "*Mary is happy*", for which we obtain the logical expression `happy' mary'`. Therefore, `happy' mary'` should be provable in our discourse theory by using `sentence1`. Using the natural deduction calculus specified in Listing 2.13, we can in fact prove it using conjunction elimination on `sentence1`:

```
theorem1 : ⊢ happy' mary' |
         = andEr (happy' john') (happy' mary') sentence1 |
```

Therefore, the observation

"*John and Mary are happy*" $\models_{\mathcal{T}}$ "*Mary is happy*"

is compatible with our model because

$[\![$"*John and Mary are happy*"$]\!] \vdash [\![$"*Mary is happy*"$]\!]$.

If we had added any world knowledge to the discourse domain theory, we could have used that as well for our inference (see also Section 4.4).

3.4 Summary

To summarize, we have seen how GF can be connected to MMT by generating language theories from abstract syntax modules. The semantics construction is then simply a view from the language theory into a discourse domain theory. The abstract syntax as an interface between the parser generated from a concrete syntax and the semantics construction (see Figure 3.5). We can easily add more concrete syntaxes without changing any other component of the pipeline. In the same way, we can easily add a different semantics construction without changing the grammar.

We finished with a brief sketch in Section 3.3 of how we can test our model using the theoretical setup described in Section 2.2.

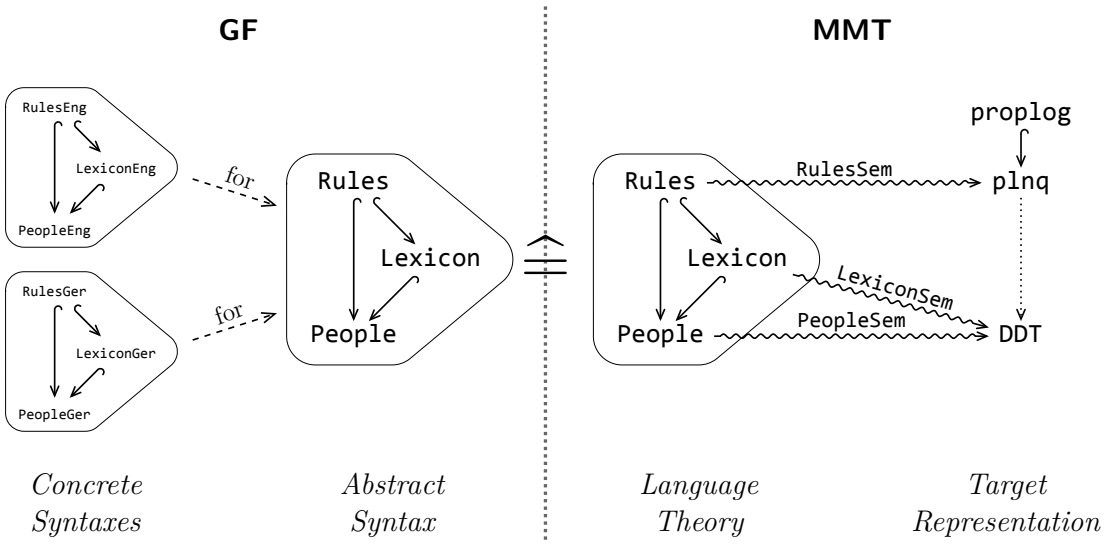


FIGURE 3.5: Components of our example GLF specification (inspired by a figure in [Sch20]).

CHAPTER 4

From GLF to GLIF: Adding Inference

Disclaimer *The idea of combining MMT and ELPI for prover generation has been published in [Koh+20b]. My contribution were large parts of the implementation and the extension to tableau provers with significant guidance from Florian Rabe, as well as working out the details of the prover generation. This chapter describes the prover generation in much more detail. The extended report [Koh+20a] contains more ideas on how it could be applied to natural language semantics, specifically for the implementation of tableaux machines (see also Section 6.4).*

A system description paper for GLIF has been published in [SK20]. My contribution was coming up with the example and most of the writing.

In Chapter 3 we introduced GLF as a framework for parsing along with compositional semantics construction. Historically, these steps were the main focus in the development of Montague-style fragments. However, we believe that a true NLU framework also requires an inference-based processing step for the semantic/pragmatic analysis (Section 2.3.5). This led to the development of **GLIF** – the **Grammatical Logical Inference Framework** – which augments GLF with an inference component.

Concretely, we will use ELPI (see Section 2.6) for the inference component. After a brief description on the basic integration of MMT and ELPI, we will discuss how theorem provers can be generated from natural deduction calculi specified in MMT. I believe that the ability to generate theorem provers automatically can significantly help with the prototyping of NLU pipelines as it allows the researcher to experiment with different logics and calculi more easily. We will also see how the same methods can be applied to a tableau calculus.

4.1 Basic Integration: From MMT to ELPI

The basic integration is very straight-forward: instead of generating MMT terms at the end of the semantics construction, we generate ELPI terms, which can then be processed by some custom ELPI code. To convert an MMT term into an ELPI term, we simply have to replace MMT notations by the symbol names and use a different notation for λ-abstraction. For example, instead of

```
∀ [x] (happy' x)
```

we have to write

```
forall x \ happy x
```

```
% propositional logic
kind o type.
type not o -> o.
type and o -> o -> o.
% or, impl, ...
```

```
% extension to first-order logic
kind i type.
type forall (i -> o) -> o.
type exists (i -> o) -> o.
```

Listing 4.1: Logic signatures in ELPI.

MMT can also generate the corresponding typing rules in ELPI. For example, the MMT declaration

```
forall : (ι⟶o)⟶o | # ∀ 1 |
```

gets converted to the ELPI rule

```
type forall (individual -> proposition) -> proposition.
```

Listing 4.1 shows a more complete signature of propositional logic and first-order logic in ELPI.

The GLIF system provides commands for e.g. filtering the output of the semantics construction using an ELPI predicate. Section 5.2.2 describes the available commands in more detail. In the next sections we will explore how ELPI provers can (sometimes) be generated from calculi specified in MMT.

4.2 Generating ELPI Provers from MMT

In Section 2.6 we have seen how ELPI provers can be implemented with helper predicates in a very systematic way. To facilitate quick experimentation with different logics, we want to generate these provers automatically.

4.2.1 Intuition: Curry-Howard

Our goal is now to generate ELPI provers from the natural deduction rules specified in MMT. The disjunction elimination rule ($\vee E$, see also Figure 2.12) is relatively complex, which makes it a good example to show the basic idea. In MMT it is described with the following declaration:

```
orE : {A,B,C} ⊢A∨B ⟶ (⊢A⟶⊢C) ⟶ (⊢B⟶⊢C) ⟶ ⊢C |
```

Applying the Curry-Howard isomorphism, which maps Π binders (denoted in MMT with `{·}`) to universal quantifiers and function types to implications, we get the following logical expression:

$$\forall A, B, C. \vdash (A \vee B) \Rightarrow (\vdash A \Rightarrow \vdash C) \Rightarrow (\vdash B \Rightarrow \vdash C) \Rightarrow \vdash C.$$

This expression can be directly written in ELPI:

```
pi a \ pi b \ pi c \ ded (or a b) =>
    (ded a => ded c) => (ded b => ded c) => ded c.
```

Proof certificates, helper predicates and the use of `ded/hyp` aside, this is exactly the expression we need. However, we can use the following tweaks to make it look closer to the way we wrote it in Section 2.6:

- Remove leading **`pi`**s and use free variables instead.
- Write `(`φ_1`, ..., `φ_n`) => `ψ instead of φ_1 `=> ... =>` φ_n `=>` ψ, exploiting that $\varphi_a \Rightarrow \varphi_b \Rightarrow \psi$ is logically equivalent to $(\varphi_a \wedge \varphi_b) \Rightarrow \psi$.
- Rewrite the top-level implication $\varphi \Rightarrow \psi$ as ψ `:-` φ.

This way, we end up with

```
ded C :- ded (or A B), ded A => ded C, ded B => ded C.
```

From now on, we will use the original and the tweaked representation interchangeably. In the implementation, the tweaks are applied automatically during the linearization of ELPI ASTs.

4.2.2 Actual Translation

In the previous section we have seen how applying the Curry-Howard isomorphism provides a meaningful λProlog expression. In practice, however, we need to do more complex processing to account for helper predicates and proof certificates.

Anatomy of ND Rules in MMT

Before describing the actual translation, we need to analyse the structure of a natural deduction rule written in MMT. The $\vee E$ rule is a nice example for this:

```
orE : {A,B,C} ⊢A∨B → (⊢A → ⊢C) → (⊢B → ⊢C) → ⊢C ❙
```

We identify three parts of the rule:

- The **parameters**: `A`, `B` and `C`.
- The **assumptions**: `⊢A∨B`, `⊢A→⊢C` and `⊢B→⊢C`.
- The **conclusion**: `⊢C`.

The conclusion (`⊢C`) is an example of an **atomic judgment**. To identify atomic judgments and distinguish deduction rules from other declarations, the ⊢ operator was declared with a special annotation, marking it as a judgment operator:

```
ded : o → type | # ⊢ 1 prec 10 | role Judgment ❙
```

Like rules, assumptions also consist of three parts:

- The **parameters** (analogous to rule parameters): no assumptions in $\vee E$ have parameters, but $\forall I$ (Listing 2.13) has the assumption `{X} ⊢P X`, which has the parameter `X`.

- The **hypotheses** (analogous to rule assumptions): e.g. $\vdash\mathsf{A}$ in the assumption $\vdash\mathsf{A}\to\vdash\mathsf{C}$.
- The **thesis** (analogous to rule conclusion): e.g. $\vdash\mathsf{C}$ in the assumption $\vdash\mathsf{A}\to\vdash\mathsf{C}$.

Translation

Let us now consider a generic rule r in MMT, consisting of parameters $p_1, \ldots, p_m$, assumptions $a_1, \ldots, a_n$ and a conclusion c:[1]

$$r : \prod_{p_1,\ldots,p_m} a_1 \to \ldots \to a_n \to \vdash c.$$

Assumptions a_i are of the form

$$a_i = \prod_{q_1^i,\ldots,q_m^i} \vdash h_1^i \to \vdash h_{n_i}^i \to \vdash t^i.$$

If we follow Section 4.2.1 and ignore the need for helper predicates for a moment, the translation of r should be

$$\prod_{p_1} \cdots \prod_{p_m} a_1' \Rightarrow \ldots \Rightarrow a_n' \Rightarrow \text{ded } c$$

where a_i' stands for the translation of a_i:

$$a_i' = \prod_{q_1^i} \cdots \prod_{q_m^i} \text{ded } h_1^i \Rightarrow \ldots \Rightarrow \text{ded } h_{n_i}^i \Rightarrow \text{ded } t^i.$$

For the actual translation, we will need one proof certificate X for the conclusion and one proof certificate X_i for the thesis t_i of each assumption a_i. The helper predicate has to be called with all certificates as well as all rule parameters:

$$\text{helper}/r\ X\ p_1\ \cdots\ p_m\ X_1\ \cdots\ X_n.$$

The actual rule translation is now

$$\prod_{p_1,\ldots,p_m,X,X_1,\ldots,X_n} (\text{helper}/r\ X\ p_1\ \cdots\ p_m\ X_1\ \cdots\ X_n) \Rightarrow a_1' \Rightarrow \ldots \Rightarrow a_n' \Rightarrow \text{ded } X\ c$$

with

$$a_i' = \prod_{q_1^i,\ldots,q_m^i} \text{ded/hyp } h_1^i \Rightarrow \ldots \Rightarrow \text{ded/hyp } h_{n_i}^i \Rightarrow \text{ded } X_i\ t^i$$

or (using the hypothesis handling from Section 2.6.4)

$$a_i' = \prod_{q_1^i,\ldots,q_m^i,y_1,\ldots,y_{n_i}} \text{ded/hyp } y_1\ h_1^i \Rightarrow \ldots \Rightarrow \text{ded/hyp } y_{n_i}\ h_{n_i}^i \Rightarrow \text{ded}\,(X_i\ y_1\ \cdots\ y_{n_i})\ t^i.$$

[1] We will not use MMT surface syntax here as we need to use indices and ellipses. For an overview of different notations, see also Appendix C.

We can describe the generation of helper predicates in the same way. Here is the general form of a helper predicate for iterative deepening:

$$\prod_{p_1,\ldots,p_m,N,N'} N > 0 \Rightarrow N' \text{ is } N-1 \Rightarrow$$
$$\text{helper}/r\ (\text{idcert}\ N)\ p_1\ \cdots\ p_m\ (\lambda y_1,\ldots,y_{n_1}.\,\text{idcert}\ N')\ \cdots\ (\lambda y_1,\ldots,y_{n_n}.\,\text{idcert}\ N')$$

We will omit the general form of the other helper predicates.

Caveat: $\forall E$

The above translation rules require an extra tweak to be able to handle the natural deduction rule

```
forallE : {P,C} ⊢∀P → ⊢P C
```

The reason is that the naive translation

```
ded X (P C) :- helper/forallE X P C X1, ded X1 (forall P).
```

requires unification on `P C`, which is not covered by λProlog's pattern fragment. Instead, we modify the translation to get

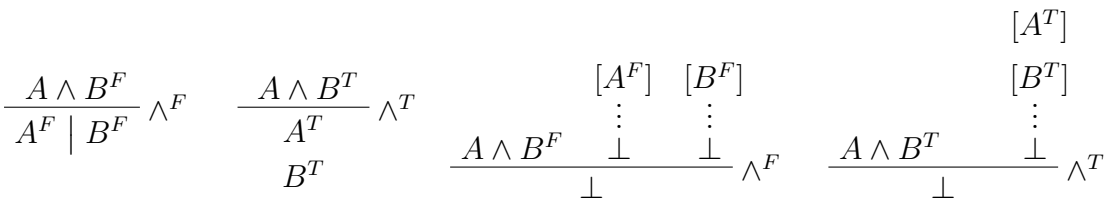
```
ded X A :- helper/forallE X A P C X1, A = P C, ded X1 (forall P).
```

Now we just moved the unification problem after the call to the helper predicate. This solves the problem if the helper predicate instantiates `P` thus simplifying the unification problem.

4.2.3 Overview: Generated Helper Predicates

Currently, six different helper predicates can be generated for every rule. Each helper predicate gets triggered by its own certificate type:

- `idcert` enforces a specified depth-limit to the search. With some controller code it can be used for iterative deepening search.
- `ptcert` records a proof term.
- `prodcert` combines two different certificates. The corresponding helper predicates get called in sequence.
- `hdcert` hands down an arbitrary value. We will use it for collecting models in a tableau setting (Section 4.3.4).
- `ucert` limits how often a rule can be applied to a particular formula. We will use it to make tableau provers more efficient (Section 4.3.3).
- `bccert` handles non-analytic elimination rules more carefully, resulting in more efficient provers. For more details, the reader is referred to [Koh+20b].



(a) Compact notation. (b) Closer to implementation in LF/MMT.

FIGURE 4.2: Different notations for tableau rules.

4.3 Tableaux

So far, we have only talked about natural deduction calculi. In this section we will take a look at tableau calculi, which are of particular interest for natural language understanding because they can be used for model generation. For a case study on tableau-based model generation see Section 6.4. In this section we will discuss how tableau provers can be generated using the mechanisms described in the previous section.

4.3.1 Tableau Rules in LF

Before we can generate tableau provers, we have to specify a tableau calculus in LF/MMT – just like we specified a natural deduction calculus (Listing 2.13). We start by introducing three judgments: $\cdot^T : o \to \mathit{type}$ to mark positive formulae (i.e. to indicate that a proposition is true), $\cdot^F : o \to \mathit{type}$ to mark negative formulae and $\bot : \mathit{type}$ to indicate a closed branch (i.e. a contradiction). In a tableau prover, a proposition A is proven by marking it as false and deriving a contradiction (i.e. closing all branches). In LF we write this as

$$\text{start} : \prod_{A:o} (A^F \to \bot) \to A^T.$$

A branch can be closed if a formula is marked as both positive and negative.

$$\text{close} : \prod_{A:o} A^T \to A^F \to \bot.$$

With this in place, we can start specifying the actual tableau rules. Figure 4.2a shows the rules for conjunction as one might know them. To write them in LF, we have to be a bit more explicit about what these rules mean. For example, the $\wedge^F$ rule states that we can close a branch with the entry $A \wedge B^F$ if

- we can close the branch after adding A^F, and
- we can also close the branch after adding B^F.

Figure 4.2b uses a notation that makes this more explicit. In LF, we can thus write the rule as

$$\wedge^F : \prod_{A:o, B:o} A \wedge B^F \to (A^F \to \bot) \to (B^F \to \bot) \to \bot$$

and for $\wedge^T$ as

$$\wedge^T : \prod_{A:o,B:o} A \wedge B^T \to (A^T \to B^T \to \bot) \to \bot.$$

Similarly, we can write the rules for negation as

$$\neg^F : \prod_{A:o} \neg A^F \to (A^T \to \bot) \to \bot$$
$$\neg^T : \prod_{A:o} \neg A^T \to (A^F \to \bot) \to \bot$$

Appendix B has the rules in MMT surface syntax.

4.3.2 Tableau Proofs in LF

Let us try to express an actual proof using these rules. As it is difficult to come up with short and interesting examples using only conjunction and negation, we will add rules for implication and disjunction and prove

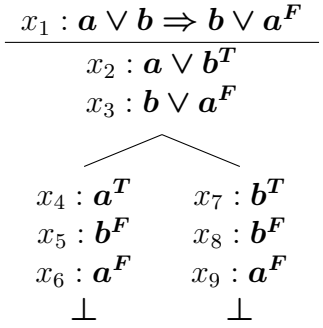
$$a \vee b \Rightarrow b \vee a.$$

Figure 4.3 shows the necessary tableau rules and an example proof represented as a tree. If we want to represent that proof in LF, we get a relatively complex expression:

$$\begin{aligned}
\text{theorem} \ : \ & \Pi_{a:o,b:o}\ (a \vee b \Rightarrow b \vee a)^T \\
= \ & \lambda a, b.\ \text{start}\ (a \vee b \Rightarrow b \vee a) \\
& \quad \lambda x_1.\ \Rightarrow^F (a \vee b)\ (b \vee a)\ x_1 \\
& \quad\quad \lambda x_2, x_3.\ \vee^T a\ b\ x_2 \\
& \quad\quad\quad (\lambda x_4.\ \vee^F b\ a\ x_3\ \lambda x_5, x_6.\ \text{close}\ a\ x_4\ x_6) \\
& \quad\quad\quad (\lambda x_7.\ \vee^F b\ a\ x_3\ \lambda x_8, x_9.\ \text{close}\ b\ x_7\ x_8)
\end{aligned}$$

If we now drop the Π-bound arguments, we obtain a more readable expression. In MMT we can use notations for that as it can infer the omitted arguments. By annotating the variables with their types, we obtain a very similar representation to the one shown in Figure 4.3:

$$\begin{aligned}
\text{theorem} \ : \ & \Pi_{a:o,b:o}\ (a \vee b \Rightarrow b \vee a)^T \\
= \ & \lambda a, b.\ \text{start} \\
& \quad \lambda x_1 : \boldsymbol{a \vee b \Rightarrow b \vee a^F}.\ \Rightarrow^F x_1 \\
& \quad\quad \lambda x_2 : \boldsymbol{a \vee b^T}.\ \lambda x_3 : \boldsymbol{b \vee a^F}.\ \vee^T x_2 \\
& \quad\quad\quad (\lambda x_4 : \boldsymbol{a^T}.\ \vee^F x_3\ \lambda x_5 : \boldsymbol{b^F}.\ \lambda x_6 : \boldsymbol{a^F}.\ \textbf{close}\ x_4\ x_6) \\
& \quad\quad\quad (\lambda x_7 : \boldsymbol{b^T}.\ \vee^F x_3\ \lambda x_8 : \boldsymbol{b^F}.\ \lambda x_9 : \boldsymbol{a^F}.\ \textbf{close}\ x_7\ x_8)
\end{aligned}$$



$$\vee^F : \prod_{A:o,B:o} A \vee B^F \to (A^F \to B^F \to \bot) \to \bot$$
$$\vee^T : \prod_{A:o,B:o} A \vee B^T \to (A^T \to \bot) \to (B^T \to \bot) \to \bot$$
$$\Rightarrow^F : \prod_{A:o,B:o} A \Rightarrow B^F \to (A^T \to B^F \to \bot) \to \bot$$
$$\Rightarrow^T : \prod_{A:o,B:o} A \Rightarrow B^T \to (A^F \to \bot) \to (B^T \to \bot) \to \bot$$

FIGURE 4.3: Example proof and the tableau rules for disjunction and implication.

4.3.3 Generating Provers

If we apply the translation rules described in Section 4.2.2 to the tableau rules specified above, we get everything we need for a usable tableau prover. Here is the translation of the start and $\wedge^F$ rule:

```
marktrue X A :- helper/start X A X2,
    (pi (x1 \ markfalse/hyp x1 A => closedbranch (X2 x1))).

closedbranch X :- helper/andF X A B X1 X2 X3,
    markfalse X1 (and A B),
    (pi (x1 \ markfalse/hyp x1 A => closedbranch (X2 x1))),
    (pi (x2 \ markfalse/hyp x2 B => closedbranch (X3 x2))).
```

If we use the proof certificate for iterative deepening (the relevant helper rules are generated), we get a simple tableau prover. However, it is not the most efficient tableau prover. Instead, we can use another certificate (`ucert`), which tracks for all marked formulae how often rules have been applied to them. The helper predicates then prevent rules from being applied too often. In the case of propositional logic, rules only have to be applied once. The resulting theorem prover is more efficient than the one that uses iterative deepening.

4.3.4 Model Generation

Tableau calculi can also be used for model generation by marking the initial formula as true and collecting branches that cannot be closed (see also Figure 6.4). With a few tricks, we can do the same thing with the generated rules. Apart from a small amount of boilerplate code, we can use another type of proof certificate, `hdcert`, that simply passes on a value. In this case, we will pass a reference to an ELPI "safe" that can store data. Whenever a branch cannot get closed we will collect all formulae on the current branch and store them in that safe. In Section 6.4 we will discuss a GLIF case study that uses tableaux-based model generation to maintain a discourse model.

4.3.5 Soundness and Completeness

There are of course better, dedicated tools for generating tableau provers (see e.g. [AG09]). The idea of generating provers from MMT is based on the idea that MMT can act as a logic-development framework. In Sections 2.5 and 4.3.1 we have already seen how to implement in MMT

- the logic syntax,
- the semantics construction as a view,
- the natural deduction rules, and
- the tableau rules.

In this section we will see how, using MMT views, we can also verify the equivalence of the natural deduction and tableau rules, which gives us relative soundness and completeness.

Let us assume that the natural deduction calculus we specified is sound and complete. We can check the soundness of the tableau rules with a view from the theory `tab`, which contains the tableau rules, into the theory `nd`, which contains the natural deduction rules. First, we have to map the judgments:

```
marktrue     = [A] ⊢A
markfalse    = [A] ⊢¬A
closedbranch = ↯
```

And now we test the soundness of the tableau rules by proving them with the natural deduction rules. As an example, we will look at the $\wedge^T$, which was declared with the type

```
{A,B} A∧B^T → (A^T → B^T → ⊥) → ⊥
```

In the view we can prove this with

```
andT = [A, B, ab, abr] abr (andel ab) (ander ab)
```

In this case, `ab` has type `⊢A∧B` and `abr` has type `⊢A → ⊢B → ↯`. To prove $\lightning$ we just need to supply `abr` with an argument of type `⊢A` and an argument of type `⊢B`. We can get those by applying $\wedge E_l$ (`andel`) and $\wedge E_r$ (`ander`) to `ab`.

As one might expect, we can only show the soundness of the tableau rules if we have natural deduction rules like negation elimination, i.e. minimal or intuitionistic logic are insufficient.

To show the completeness of the tableau rules (again assuming the soundness and completeness of the natural deduction rules), we need to describe how any natural deduction proof can be expressed as a tableau proof. We do this with a view from `nd` to `tab`. First, we again have to map the judgments:

```
ded    = [p] p^T
falsum = ⊥
```

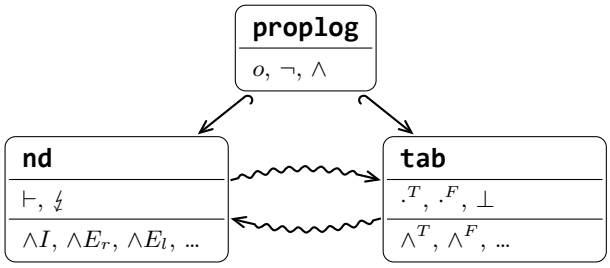


FIGURE 4.4: Theory graph: ND and tableau rules.

And now we can map the rules, exemplified by $\neg I$ (negation introduction) which has type `{A} (⊢A→↯) → ⊢¬A`:

```
notI = [A, ar:A^T→⊥] start ([naf:¬A^F] (notf naf ar))
```

Since we have to proof $\neg A^T$, we use the `start` rule and supply it with a proof that $\neg G^F \rightarrow \bot$ using the $\neg^F$ (`notf`) rule.

Figure 4.4 shows the theory graph of the theories and views we have created. The complete implementation is in Appendix B.

4.4 Summary: GLIF

In the previous sections we have seen how we can automatically generate ELPI theorem provers from MMT. Let us now see how that helps us with the prototyping of NLU pipelines.

The whole GLIF pipeline consists of three steps: parsing, semantics construction and semantic/pragmatic analysis, which is based on ELPI. As the semantic/pragmatic analysis usually uses inference, having automatically generated theorem provers should help with the implementation. Figure 4.5 illustrates the different specifications needed for a GLIF pipeline. The language theory, the ELPI prover and the world knowledge for the semantic/pragmatic analysis can be generated automatically.

World knowledge can be entered into the discourse domain theory as axioms, like

```
fact1 : ⊢ (happy mary) ⇒ ¬(sad mary)
```

The translation rules from Section 4.2.2 then create the relevant ELPI code.

In Section 3.3 we saw how we can use natural deduction rules to check in MMT whether textual entailment coincides with the syntactic consequence relation for a particular example. Using a generated prover, we can now do the same thing automatically. For example, given the world knowledge from above, we can now automatically verify that

$$[\![\text{“}\textit{Peter and Mary are happy}\text{”}]\!] \vdash [\![\text{“}\textit{Mary isn't sad}\text{”}]\!].$$

which, of course, coincides with the textual entailment relation.

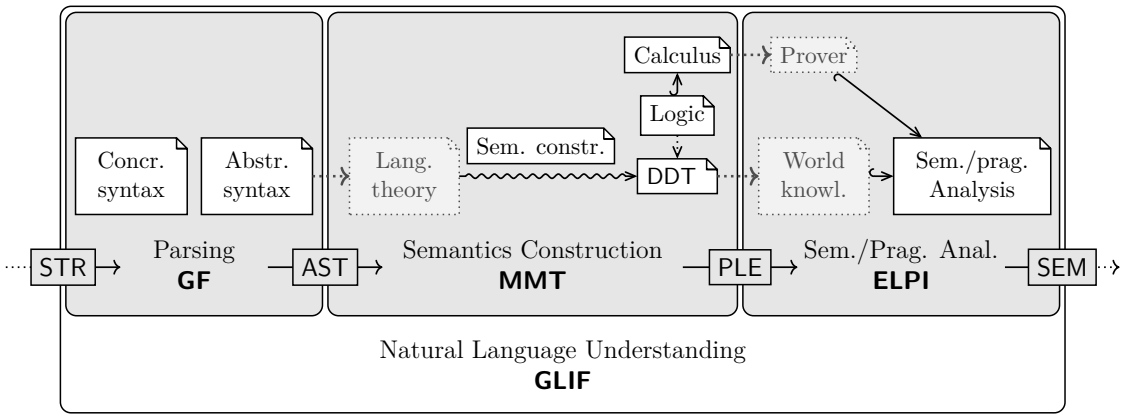


Figure 4.5: Specification of an NLU pipeline in GLIF. $\boxed{A} \dashrightarrow \boxed{B}$ indicates that B was generated from A. The input to the pipeline is a natural language string (STR). Parsing transforms the string into an AST and the semantics construction creates a preliminary logical expression (PLE) from the AST. The semantic/pragmatic analysis refines that expression to obtain the final semantic representation (SEM).

CHAPTER 5

The GLIF System and Interface

The GLF/GLIF system has changed and matured significantly since GLF was first introduced in [KS]. In the following sections we will take a closer look at the GLIF system and its Jupyter-based interface.

5.1 MMT Extensions

The core functionality of GLIF is to connect GF, MMT and ELPI in a way that users can easily implement the NLU pipelines described in Section 2.3. To make the connection as seamless as possible, a number of MMT extensions were implemented, which can be found in the MMT repository [MMTa]:

- `GfImporter`: Parses GF abstract syntaxes and imports them as LF theories. In other words, it generates language theories from GF abstract syntax modules.
- `GlfBuildServer`: Makes it possible to run MMT as a server and use HTTP requests for building GF files (with the `GfImporter`) and MMT files (with the standard importer for MMT surface syntax).
- `GlfConstructServer`: Can be used to apply a semantics construction view. The `GlfConstructServer` parses the ASTs and creates the corresponding terms in the language theory. Then it applies the semantics construction view and optionally simplifies the resulting expressions with definition expansion and β-reduction. The resulting expressions are returned either using MMT's notations (e.g. `a ∧ b`) or not using them (`conjunction a b`) to allow for further processing with ELPI.
- `ElpiGenerationServer`: Generates ELPI code. Parameters indicate whether provers or just typing rules should be generated.

With these extensions it is now possible to implement a simple script that connects the different systems to get the basic GLIF functionality. To execute a GLIF pipeline it would basically:

1. Launch MMT with the necessary extensions,
2. import the GF and MMT files with the `GlfBuildServer`,
3. maybe generate some ELPI code using the `ElpiGenerationServer`,
4. run the GF shell in the background and import the grammar,
5. parse sentences with the GF shell,
6. apply the semantics construction via the `GlfConstructServer`
7. and optionally run the ELPI code on the results of the semantics construction.

Using such scripts turned out to be rather inconvenient as they require a lot of configuration and the different components (GF, MMT, ELPI) are developed in separate places. Therefore, they are rarely used in practice – mostly for larger projects. Instead, we have developed a Jupyter interface that provides a more unified interface to GLIF. We will discuss it in the next section.

5.2 Jupyter Interface

Disclaimer *The Jupyter kernel has been largely implemented by Kai Amann as he had previous experience with Jupyter. I was acting more in a supervising role, deciding what features are needed, though I have also contributed code for stub generation, communication with GF and ELPI, etc. Tom Wiesing has also helped, especially, with packaging and installation. A description of the Jupyter interface has already been pre-published at [SAK20].*

Understanding how to make a system like GLIF more usable is a challenge on its own. Originally, GLIF pipelines could only be tested with rather unwieldy command line scripts. With Jupyter notebooks we now have a more unified and intuitive user interface for both implementing and testing GLIF pipelines.

5.2.1 Basic Functionality

Jupyter notebooks contain a sequence of cells (see Figure 5.1a). **Markdown cells** can be used to add formatted text to the notebook. The most important type of cells are **code cells**. On execution, their content is handed to the Jupyter kernel (in this case our custom **GLIF kernel**), which may return some output. The GLIF kernel also provides syntax highlighting for the code cells. When a code cell is executed, the GLIF kernel first decides what kind of content the cell contains using pattern matching. Depending on the content type, the next action is chosen:

- *GF modules* are imported into GF and MMT. If any errors occur, they are displayed (Figure 5.1b).
- *MMT theories and views* are imported into MMT. Errors are displayed as well.
- *ELPI content* is stored in a file. Currently, ELPI content is only tested (compiled, type-checked) when it is used.
- *GLIF commands*, which are an extension of the GF shell commands, are handled by the GLIF kernel or passed on to the GF shell (see Section 5.2.2).

GF and MMT are always running in the background, while ELPI is called on demand.

5.2.2 GLIF Commands

The GF shell provides many useful commands for using GF grammars. For example, the command

Example Notebook

This is a Markdown cell. Let's enter the `GF` grammar using a code cell:

```
abstract Grammar = {
  cat       -- categories
    Pers; Property; Stmt;
  fun
    makeStmt : Pers -> Property -> Stmt;
    and      : Stmt -> Stmt -> Stmt;
    everyone : Pers;
}
```

```
Defined Grammar
```

(a) A markdown cell followed by a code cell.

```
1 abstract Lexicon = Grammar ** {
2   fun
3     john, mary : Person;
4     happy, sad, smart : Property;
5 }
```

```
/home/jfs/mmt/content/MathHub/comma/jupyter/source/people/Lexicon.gf:3:
  Happened in the renaming of mary
   constant not found: Person
   given Grammar, Lexicon
```

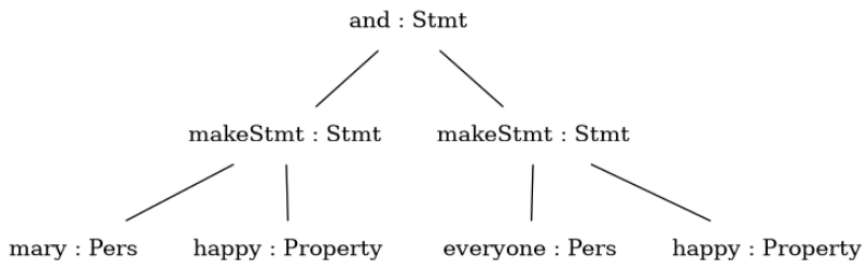
(b) An error in a GF module. Line numbers are enabled here.

```
parse -cat=Stmt "everyone is sad" | construct -v PeopleSemantics
```

```
∀[x]¬(happy' x)
```

(c) Parsing and semantics construction in one command.

```
parse -cat=Stmt "Mary is happy and everyone is happy" | view_tree | show
```



(d) An AST visualized inside the notebook.

FIGURE 5.1: Screenshots of a Jupyter notebook.

```
parse -Lang=Eng "Nadya and Timothy laugh"
```

parses a sentence using the English concrete syntax and outputs an AST:

```
sentence (andPerson nadya timothy) laugh
```

A different command can linearize that AST (also in a different language):

```
linearize -Lang=Tur sentence (andPerson nadya timothy) laugh
```

Instead of copy-pasting the output of one command as an input to another command, GF commands can be concatenated with the pipe operator (`|`). For example, we can obtain the same translation with the command

```
parse -Lang=Eng "Nadya and Timothy laugh" | linearize -Lang=Tur
```

GLIF commands extend the set of commands provided by GF with a number of other commands. GF commands will be passed to the GF shell, while the added commands will be handled by the kernel. One of the most important additions is the **`construct`** command, which applies the semantics construction to an AST (see also Figure 5.1c):

```
> parse -Lang=Eng "Nadya and Timothy laugh" | construct
laugh nadya∧laugh timothy
```

Since ELPI can be used in many ways, there are a number of different commands for using an ELPI predicate $\mathcal{P}$ to process a list of logical expressions $\mathcal{L}$ obtained from the semantics construction.

- **`elpi run`** is, in a way, the most general command. It simply applies $\mathcal{P}$ to $\mathcal{L}$. This is only useful if $\mathcal{P}$ writes some relevant output to `stdout`.
- **`elpi runeach`** applies $\mathcal{P}$ to each element of $\mathcal{L}$ individually.
- **`elpi filter`** returns all elements $e \in \mathcal{L}$ for which $\mathcal{P}(e)$ holds.
- **`elpi map`** requires $\mathcal{P}$ to be a binary predicate. It returns all expressions $?X$ for which $\mathcal{P}(e, ?X)$ holds for some $e \in \mathcal{L}$.

For example

```
parse "what is the capital of Slovakia" | construct
                                        | elpi map ... answer
```

could provide an answer to the question. The `...` should be the name of the ELPI file. And to discard contradictory readings, we could use the command

```
parse "John feeds ducks at the bank" | construct
                                     | elpi filter ... check
```

Note that multi-line commands are currently not supported.

Aside from **`construct`** and the **`elpi`** commands, there are a number of kernel commands for things like setting the MMT archive but also for visualizing parse

```
concrete ExampleIta of Example = PeopleIta ** {
  lincat
    V = _ ;

  lin
    -- s3 : NP -> V -> S
    s3 _ _ = _ ;
    -- run : V
    run = _ ;
}
```

Listing 5.2: Stub generated from Listing 3.2.

trees and ASTs inside the notebook (Figure 5.1d). Section 8.2 discusses how GLIF commands could be improved.

5.2.3 Other Features

Jupyter code cells support customizable tab completion. We use this to facilitate entering unicode characters for the MMT theories. For example, `\wedge` gets completed to ∧.

The completion is also used for **stub generation**. Let us say that the user has provided an abstract syntax module `Example`. Tab completing `ExampleIta` then results in a stub for a concrete syntax module `ExampleIta` (Listing 5.2). Similarly, tab completing `ExampleSemantics` results in a stub for a semantics construction view. It turned out that it is helpful if the stubs contain comments indicating the types of function constants.

CHAPTER 6

Case Studies

In this chapter we will try to test GLIF's capabilities in a variety of case studies. First, we will explore the implementation of controlled natural languages in Section 6.1. Controlled natural languages use symbolic methods to map a subset of natural language into a non-ambiguous semantic representation, which makes them a very natural case study for GLIF. In the remaining case studies we also want to use GLIF's inference component. [BB05], which introduces Prolog/RINL, ends with some inference-based question answering. Therefore, we will also use GLIF for some basic question answering (Section 6.2). To make the case studies more diverse, we will use epistemic logic. The actual goal of the inference component is of course the semantic/pragmatic analysis. In Section 6.3 we will use it for semantic pruning and show how it can be used for high-precision translation. We will finish with a case study on tableaux machines in Section 6.4. Tableaux machines maintain and update a discourse model and implement a relatively comprehensive semantic/pragmatic analysis.

6.1 Controlled Natural Languages

So far, we have used GLIF in the spirit of Montague's method of fragments. It allows us to experiment with ideas for natural language understanding on various fragments of natural language. However, these fragments are limited in their expressivity, and it appears unlikely that GLIF can be used to process "real texts" that were not written with a GLIF implementation in mind – at least not without significant extensions to GLIF. But maybe the processing capabilities of GLIF can sometimes justify the effort of writing texts that lie in a particular, manageable fragment of natural language? That brings us to the concept of **controlled natural languages**, which are formal languages that represent a fragment of natural language with well-defined semantics. The main difference to a Montague fragment is that the semantics construction does not leave any ambiguities. A particularly well-known general-purpose controlled natural language is Attempto Controlled English (ACE) [FSS98]. In ACE, ambiguities are avoided through a set of interpretation rules. For example, anaphoric references – a common source of ambiguity – are always resolved to "the most recent suitable noun phrase that agrees in number and gender" [FSS98].

In this section we will discuss two case studies in which GLIF was used to implement a controlled natural language.

```
Enter command: Let | G | be a notation for the cardinality of G.
def bar(gVar): return gVar.cardinality()

Enter command: Let A_N be a notation for the alternating group on N symbols.
def aVar(nVar): return AlternatingGroup(nVar)

Enter command: What is |A_5|?
print(bar(aVar(int(5))))
sage: 60

Enter command: What is A_5?
print(aVar(int(5)))
sage: Alternating group of order 5!/2 as a permutation group
```

Figure 6.1: Interacting with SageMath via a controlled natural language.

6.1.1 A Controlled Natural Input Language for Sage

Disclaimer *This case study was already presented at the International Congress on Mathematical Software (ICMS) 2020. It was not included in the accompanying publication ([SAK20]) though. I have both created and presented the case study. I am very grateful to the participants of the ICMS Jupyter session, who offered many good suggestions and have certainly influenced the write-up of this case study.*

SageMath (or Sage) [Sag] is a Python-based computer algebra system. In this case study, we will prototype a controlled natural language that gets translated into Sage commands. For example, the question

"*What is the cardinality of the alternating group on 5 symbols?*"

gets translated to the Sage command

```
print(AlternatingGroup(int(5)).cardinality())
```

In the prototype implementation, the Sage command gets passed to the Sage shell, which returns the answer 60. Figure 6.1 shows a more interesting interaction.

The "target logic" is a theory that contains declarations like

```
multiply : term → term → term
definition : term → term → command
```

In the prototype implementation, Sage commands were created using MMT notations and some post-processing. It would be more elegant to consider the target logic as the language theory of another grammar and use a GF concrete syntax for generating valid Sage commands (Figure 6.2). In that case, it might even be possible to eliminate MMT from the pipeline and use GF for the semantics construction (see also Section 2.4.3) – unless we need ELPI.

The prototype implementation lets users introduce some notations and ask basic questions (Figure 6.1). With better coverage, this could result in an alternative

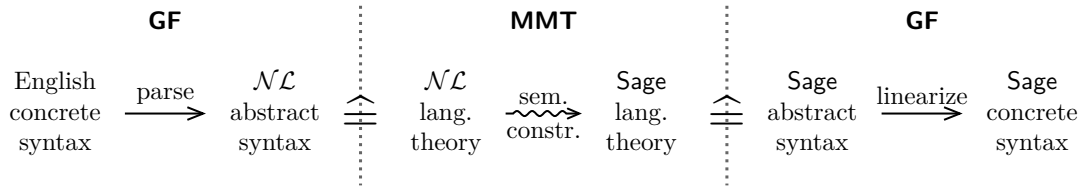


FIGURE 6.2: A better architecture for natural language ($\mathcal{NL}$) to Sage translation.

way of accessing Sage (akin to WolframAlpha). Since the corresponding Sage commands are shown as well, it would be interesting to explore applications in teaching computer algebra systems. For any larger attempt at a controlled natural language for Sage, the GLIF implementation should probably be as flexible and extensible as possible to make it easy to add coverage for a specific area.

6.1.2 GLForTheL

Disclaimer *The GLForTheL project has already been discussed in great detail and submitted as my master's project in [Sch20]. A summary of GLForTheL has also been pre-published in [SAK20]. Since GLForTheL is the largest test of GLIF's capabilities so far, the project will be summarized here.*

The **GLForTheL** project [Sch20; SAK20] was an experiment to reimplement an existing controlled natural language, ForTheL, in GLIF. **ForTheL** [Pas07a] is used by the proof assistant SAD (System for Automated Deduction) [VLP07]. It was originally implemented with a custom parser in Haskell. Part of the motivation behind GLForTheL was to explore the possibility of using GLIF for easily experimenting with different ForTheL extensions. With over 50 production rules in the grammar, GLForTheL is the largest project that has been implemented in GLIF so far. Since ForTheL is primarily used for mathematical texts, GLForTheL was also the first experiment of using GLIF for mathematical language.

Grammar The ForTheL grammar over-generates; it does not enforce use of the right word forms. For example, "*there is an empty set*" could as well be written as "*there are a empty sets*". GF made it easy to reject "ungrammatical" sentences. Aside from grammar-checking the user, this opens up a number of interesting possibilities as it allows us to generate grammatically correct linearizations. With a second concrete syntax, for example, it would be possible to provide accurate translations. Initially, the GLForTheL grammar was using the Resource Grammar Library and had concrete syntaxes in English and German. Later, the Resource Grammar Library, along with the German concrete syntax, were dropped because of two problems:

1. The Resource Grammar library treats common nouns (`CN`) as continuous constituents. In a mathematical setting, however, mathematical objects like "*sub-*

group of G" often get an identifier by inserting a variable ("*every subgroup* ***H*** *of G*").

2. Some ForTheL sentences are not really proper natural language (e.g. "*let x is even stand for x is divisible by two*").

The GLForTheL grammar only covers a subset of ForTheL. Except for the proofs, it can parse the `emptyset.ftl` example file from [Sad]. While ForTheL allows the introduction of new words in definitions etc., GLForTheL currently uses a hard-coded lexicon.

Semantics Construction GLForTheL (like ForTheL) translates sentences into first-order logic. While [Pas07a] describes the semantics construction via textual transformation rules, GLForTheL uses lambda functions. An advantage of the textual transformation rules might be that they can be more easily communicated to potential users. Implementing the GLForTheL semantics construction was mostly straight-forward. An interesting problem, however, was the handling of identifiers and variables. For sentences like "*every integer is even*", the semantics construction can easily create bound variables for the quantifier. But it gets tricky if the text introduces an identifier. Consider the sentence "*for every set S,* $\emptyset \subseteq S$". On the logic side we would ideally want the expression $\forall S.set(S) \Rightarrow \subseteq(\emptyset, S)$. However, the identifier "S" represented as a constant in the AST – and we cannot turn it into a variable during the semantics construction. The current (rather hacky) solution is to use a function $\widetilde{\lambda} : A \to B \to (A \to B)$, which we call a **fake lambda**. Analogous to $\lambda x.f(x) : A \to B$ where $x : A$ is a *variable*, we would have $\widetilde{\lambda} x.f(x) : A \to B$ for a *constant* x. In a post-processing step, after β-normalization, the fake lambdas are replaced by real lambdas and the "bound identifier constants" are turned into bound variables. Another problem were identifier enumerations like in "*let* x, y, z *be pairwise linearly independent vectors*". [Sch20] sketches two different ways to handle such cases:

1. Use "a lot of lambdas", which is rather tricky but ultimately works. However, we reach the limits of LF (see also Section 7.3) and have to use the more general meta theory `LFHierarchy`. Unfortunately, this is then incompatible with the automatically generated (LF-based) language theories. There is a prototype implementation that works with hand-written language theories.

2. Use the semantics construction to generate a preliminary logical representation that still contains the enumerations. Then, the inference step can be used to translate the preliminary logical representation into the desired first-order expressions via some custom ELPI code.

Document Processing GLIF and, in particular, its Jupyter interface were designed to handle hand-written example sentences. Processing a real ForTheL text requires a certain amount of preprocessing, which cannot be done in the Jupyter

interface. Instead, a custom, GLIF-based script was used for processing entire documents. The reader is referred to [Sch20] for more details. Let us look at an example sentence:

> "*a subset of S is a set T such that every element of T belongs to S*".

The GLForTheL implementation translates that sentence to the logical expression

$$\forall V_T.\,\text{subset}\ V_T\ V_S \quad \Leftrightarrow \quad \text{set}\ V_T \wedge \forall V_{\text{new}}.\,\text{elementOf}\ V_{\text{new}}\ V_T \wedge \top \Rightarrow \text{belongTo}\ V_{\text{new}}\ V_S \wedge \top.$$

The redundant $\wedge\top$ could be eliminated in an (ELPI-based) postprocessing step.

6.2 Question Answering in $S5_n$

In this case study, we will use GLIF to answer yes/no questions based on a sequence of sentences. For example, given the input

> "*John knows that Mary or Eve knows that Ping has a dog.*
> *Mary doesn't know if Ping has a dog.*
> *Does Eve know if Ping has a dog?*"

the system should reply with "*yes*". For this case study, we will need a form of **intensional logic**, specifically the multi-agent epistemic logic $\boldsymbol{S5_n}$, which makes it the only example in this thesis that is not more-or-less based on a (subset) of first-order logic. Of course, it is possible to translate $S5_n$ formulae into first-order logic. Before looking into the GLIF implementation, we will provide the reader with a brief introduction to $S5_n$.

6.2.1 Epistemic Modal Logic and $S5_n$

There is no intuitive way to represent sentences like "*John knows that* φ" in first-order logic. One might be tempted to represent it as $know(john, \varphi)$ but first-order predicates cannot have propositions as arguments. To talk about propositions, we will introduce a modal operator $\Box_a$ for each agent a. Then we can represent "*John knows that* φ" as $\Box_{john}\varphi$. This gives us a **multi-modal logic**.

As the reader may know, there is a whole zoo of modal logics, and it is not obvious, which one we should pick. In this case study, we decided to use $S5_n$. $S5_n$ is a normal modal logic, which means that we have necessitation and distribution. In addition, the following three axiom schemata hold in $S5_n$:

- **T** : $\Box_a\varphi \Rightarrow \varphi$: if a knows φ then φ is true.
- **4** : $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$: if a knows φ then a also knows that it knows φ.
- **B** : $\varphi \Rightarrow \Box_a\Diamond_a\varphi$ if φ is true then a knows that it thinks that φ might hold, from which it follows (due to **T**) that a thinks φ might hold. $\Diamond_a$ is the dual operator to $\Box_a$ and is defined as $\Diamond_a\varphi := \neg\Box_a\neg\varphi$. Intuitively, $\Diamond_a\varphi$ means that a thinks φ could be true.

Since $S5_n$ is normal, we can use Kripke semantics. Each agent a gets their own accessibility relation R_a. That would allow us to translate $S5_n$ propositions into a (subset) of first-order logic. For example, $\Box_{john}\varphi$ would be translated to $\forall x.sR_{john}x \Rightarrow \varphi'_x$, where s is the world in which the proposition is evaluated and φ'_x is the translation of φ from world x. An advantage of $S5_n$ over (general) first-order logic is that logical entailment in $S5_n$ is decidable [HM92].

For a more detailed introduction to epistemic logic, the reader may be interested in [RS19].

6.2.2 GLIF Implementation

Implementing a grammar for our example sentences was straight-forward. The language was complex enough that it made sense to use the Resource Grammar Library for the concrete syntax.

In the semantics construction, both statements and questions are translated to propositions. For example,

"*Does Eve know if Ping has a dog?*"

corresponds to the proposition

$$(\Box_{Eve}hd(Ping)) \ \vee \ \Box_{Eve}\neg hd(Ping).$$

For the inference step we need an $S5_n$ prover. It was fairly easy to convert the Prolog prover from [Boe06] into a working ELPI prover that can be directly used in GLIF.

Let us say we have a sequence of sentences $P_1, \ldots, P_n$ followed by a question Q. The answer should be "*yes*" if $[\![P_1]\!], \ldots, [\![P_n]\!] \vdash [\![Q]\!]$. Recall that $[\![S]\!]$ stands for the meaning of sentence S (i.e. the result of the semantics construction). The answer should be "*no*" if $[\![P_1]\!], \ldots, [\![P_n]\!] \vdash \neg[\![Q]\!]$. Otherwise $[\![Q]\!]$ is contingent under the premises and the answer should be "*maybe*". If there were no decision procedure for our logic, the answer could be "*maybe*" if the prover fails to find a proof of $[\![Q]\!]$ or $\neg[\![Q]\!]$ within a certain time limit. Because of the principle of explosion, the system would answer "*yes*" whenever there is a contradiction in the premises. It would probably be more intuitive to check for contradictions ($[\![P_1]\!], \ldots, [\![P_n]\!] \vdash \bot$) and then instead respond with "*contradiction*".

6.3 Translating Family Relations

In this case study, our ultimate goal is not to obtain a semantic representation for input sentences but to provide accurate translations. Consider the sentence

"*Kim is Ahmed's cousin.*"

As "*Kim*" is a unisex name, there are two possible German translations:

```
theory calculus : ur:?LF =
  include ?logic ▮
  ded : o → type | # ⊢ 1 prec 5
                 | role Judgment ▮
  contra : type | # ↯ | role Judgment ▮
  findContra : {A} ⊢ A → ⊢ ¬A → ↯ ▮
  andEl : {A} {B} ⊢ A ∧ B → ⊢ A ▮
  andEr : {A} {B} ⊢ A ∧ B → ⊢ B ▮
▮
```

```
accumulate calculus.

check P :-
  ded/hyp _ P =>
    contra (idcert 7),
  !, fail.
check _.
```

Listing 6.3: The ELPI code for discarding contradictory readings (right) is based on a prover generated from a calculus specified in MMT (left).

(1) "*Kim ist Ahmeds Cousine.*"

(2) "*Kim ist Ahmeds Cousin.*"

Translation (1) uses the feminine translation of "*cousin*" and translation (2) the masculine one. If we knew the gender of "*Kim*", we could discard the wrong translation. For example, in the sentence

"*Kim is Ahmed's cousin and the father of Grace*"

it is obvious that "*Kim*" must be male (fathers are always male).

Grammar We could model "*cousin*" as a single word with two forms in German – masculine and feminine. In this case study, however, we take a slightly different approach and assume that there are two separate words for "*cousin*": one for male cousins (`cousin_male`) and one for female cousins (`cousin_female`), analogous to how "*father*" is a male parent and "*mother*" is a female parent. Since both the female and male word for cousin are identical in English, GF generates two ASTs for our example sentence.

Semantics Construction The only purpose of the semantic representation is to identify contradictory gender assignments. Therefore, the semantics construction will ignore any irrelevant information. For example, "*Kim is the mother of Grace*" will be translated to `female kim` as mothers are always female. For the sentence

"*Kim is Ahmed's cousin and the father of Grace*"

we will get one expression for `cousin_female` and one for `cousin_male`:

```
(female kim) ∧ (male kim)      // cousin_female ⇝ contradiction
(male kim) ∧ (male kim)        // cousin_male   ⇝ no contradiction
```

Defining `male` as `[x] ¬ (female x)`, it becomes obvious that the first reading is wrong.

Inference The logic for this example is extremely simple. A calculus for finding contradictions can be easily specified in MMT (Listing 6.3, left). MMT can generate a prover from the calculus that completely suffices for our application. In ELPI we can implement a predicate `check` (Listing 6.3, right) that uses the generated prover with a depth-limit to reject readings if a contradiction is found.

Conclusion Putting it all together, we can now translate sentences like

"*Kim is Ahmed's cousin and the father of Grace*"

using the following sequence of steps:

1. Parse the sentence to obtain multiple ASTs
2. Use the semantics construction to extract gender information
3. Discard contradictory ASTs
4. Linearize only the remaining ASTs in the target language

For the example sentence, we are only left with one AST, which gets linearized to

"*Kim ist Ahmeds Cousin und Graces Vater*"

At the time of writing, the example above gets translated correctly by Google Translate. However, the sentence

"*Kim is Ahmed's cousin and Grace's sister.*"

gets wrongly translated to

"*Kim ist Ahmeds Cousin und Graces Schwester.*"

while the GLIF implementation results in the correct translation.

It would be interesting to turn this case study into a much larger one to cover more phenomena and more complex reasoning about family relations. For example, the Swedish word for "*uncle*" is "*morbror*" or "*farbror*", depending on whether it is the mother's or the father's brother.

6.4 Tableaux Machine

> ***Disclaimer*** *This case study has already been presented in [Koh+20a]. My contribution was the implementation of the tableaux machine as well coming up with the example. [Koh+20a] also contains a variant of Figure 6.6, which was created by me.*

In this case study, we implement a tableaux machine. A **tableaux machine** [KK00] maintains a model of a discourse, which gets updated after each sentence. Like tableau provers, the tableaux machine is based on tableaux. A tableau prover proves a formula φ by marking it as false and closing all branches. If that fails, the

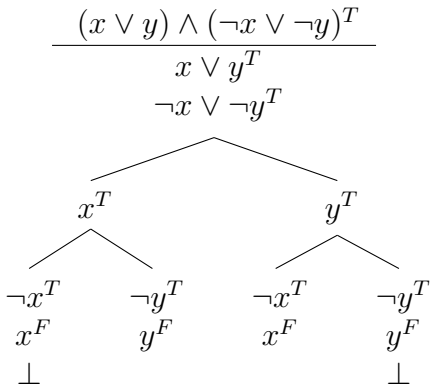


Figure 6.4: Two models: $\{x^T, y^F\}$, $\{x^F, y^T\}$

$$\frac{\forall x.P(x)^T \quad \mathcal{H} = \{a_1, \ldots, a_n\}}{\begin{array}{c} P(a_1)^T \\ \vdots \\ P(a_n)^T \end{array}} \forall^T \qquad \frac{\exists x.P(x)^T \quad \mathcal{H} = \{a_1, \ldots, a_n\} \quad c \text{ new}}{P(a_1)^T \mid \cdots \mid P(a_n)^T \mid P(c)^T} \exists^T$$

Figure 6.5: Quantifier rules in a tableaux machine. $\forall^F$ and $\exists^F$ are analogous.

open branches correspond to (Herbrand) models that make φ false. In a tableaux machine, the formulae are marked as true and open branches correspond to models that make the formulae true (Figure 6.4). We could theoretically generate all models of a discourse by simply adding all sentences into the tableau and collecting the open branches. However, the combinatorial explosion makes this impossible for any larger example. Arguably, this is also not what a human does when reading a text. Instead, the tableaux machine picks the most likely model for a sentence, according to some heuristic, and bases the processing of the next sentence on that model. If a contradiction is found later on, the tableaux machine backtracks and picks a different model.

For our example, we will use pronouns as a source of ambiguity and use first-order logic for our semantics construction. The first-order tableaux machine uses the standard propositional tableau rules but the quantifier rules deviate from the first-order standard tableau and the commonly used free variable tableau. The $\forall^T$ rule (Figure 6.5) acts on a formula $\forall x.P(x)^T$ by applying P to all ground terms in the current branch. We will denote the set of ground terms in the current branch by $\mathcal{H}$. For a formula $\exists x.P(x)^T$, a new branch is opened for every $a \in \mathcal{H} \cup \{c\}$ and initialized with $P(a)^T$, where c is a new (Skolem-) constant.

To pick a model, we can associate a cost with each open branch and pick the best one. For that we could e.g. consider **salience**, which quantifies how available different constants are at what point in the discourse. Our example implementation uses a simpler heuristic: $\exists^T$ prefers the constants that were used most recently in

the branch.

At the core of our example discourse will be the sentence "*he loves her*" and the tableaux machine will have to resolve who "*he*" and "*her*" refer to. Therefore, we initialize the tableaux machine with some world knowledge about gender, such as that nothing is both feminine and masculine. Figure 6.6 illustrates an example run of the tableaux machine. The discourse starts with

"*John talks to Mary*"

which simply adds an atomic formula to the current branch. The next sentence is "*Sasha is sad*", which adds the formula $\text{sad}(s)$ to the branch and, since Sasha is not assigned a gender in the world knowledge, we get two branches: one with $\neg\text{fem}(s)$ and one with $\neg\text{masc}(s)$. Arbitrarily the branch with $\neg\text{fem}(s)$ is picked. For simplicity, this is not shown in Figure 6.6. Things get interesting when we add the sentence

"*He loves her*"

which the semantics construction translates to

$$\exists x.\text{masc}(x) \wedge \exists y.\text{fem}(y) \wedge \text{love}(x, y).$$

The tableaux machine now tries to instantiate x and y with the most recently used constant: s. As this fails (Sasha can't be both "*he*" and "*her*") it settles for $x = s$ and $y = m$, i.e. Sasha loves Mary. We can break this model with another sentence:

"*Sasha is a woman*"

After a lot of backtracking, the tableaux machine settles on a new model in which John loves Sasha. Note that "*Sasha is a woman*" specifies the biological gender and not the grammatical gender that we have been using throughout. Of course, the biological gender determines grammatical gender here. We've glossed over this in our implementation. The distinction becomes more relevant in other languages. For example, the German world for "*girl*" ("*Mädchen*") is neuter while clearly referring to someone female.

Implementation For this case study, the tableaux machine was first implemented in ELPI without using GLIF at all. The ELPI implementation required logical expressions as input and printed the current model whenever a new expression was entered. While this might be sufficient to show an idea, GLIF made it easy to support natural language input. The grammar and semantics construction were implemented in a Jupyter notebook. As the Jupyter interface does not really support discourse-level processing, everything was combined in a command line script.

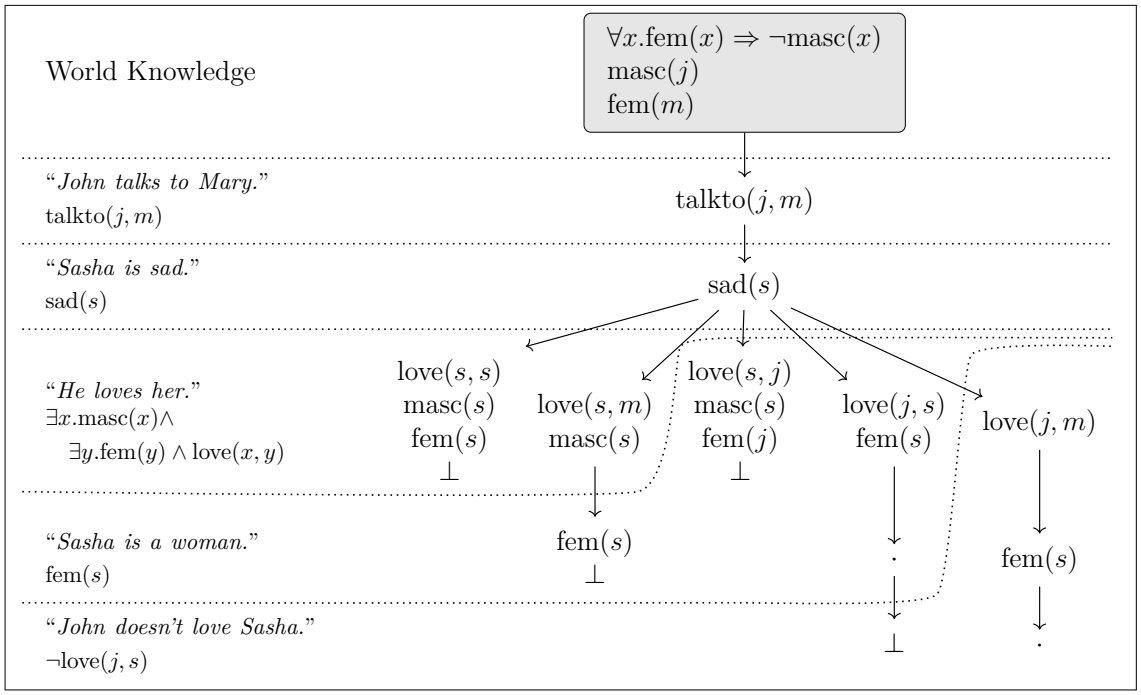


Figure 6.6: Simplified example run of the tableaux machine. Only positive atomic formulae are shown. A variant of this figure was pre-published in [Koh+20a].

CHAPTER 7

Evaluation

In this chapter we will attempt to evaluate GLIF as a tool for implementing NLU pipelines. While there are data sets for evaluating natural language inference systems (e.g. the Winograd Schema Challenge [LDM12] or the FraCaS test suite [Coo+]), there is no data set for frameworks for implementing such systems. Instead, we will compare GLF with other frameworks (Section 7.1) and then discuss my own experiences of using GLF/GLIF, including what I have learnt from the case studies.

7.1 Comparison with Existing Frameworks

The goal of GLF was to facilitate the implementation of Montague-style language-to-logic pipelines. In this section we will compare GLF with frameworks that have similar goals. Concretely, we will compare it with Prolog/RINL [BB05], Haskell/CSFP [EU10] and pure GF [Ran11]. Section 2.4 introduces these frameworks with an example implementation for sentences like

> "*John and Mary are happy and smart.*"

Appendix A contains the complete implementations in each framework.

7.1.1 Parsing

With GF/GLF the user can usually write a grammar without any considerations for how a parser is generated from the specification[1]. Let us take the rule for combining two adjective phrases (**<AP>**) as an example:

```
<AP> ::= <AP> "and" <AP>
```

In GF it can be written as

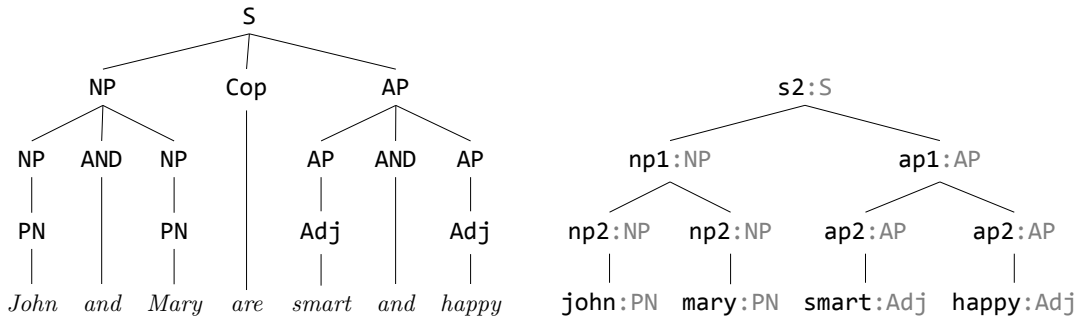
```
ap1     : AP -> AP -> AP;      -- abstract syntax
ap1 a b = a ++ "and" ++ b;     -- concrete syntax
```

or, using the Resource Grammar Library,

```
ap1 a b = mkAP and_Conj a b;
```

In Prolog/RINL and Haskell/CSFP, on the other hand, one has to actively prevent the parser from getting stuck in an infinite loop with left-recursive rules. For

[1] There can be some efficiency issues, which do require a deeper understanding (see e.g. [Ang11, pp. 67–72]).



(a) Parse tree. (b) Abstract syntax tree.

Figure 7.1: Haskell/CSFP uses parse trees, while GF/GLF uses ASTs.

example, a boolean flag can be used to block further applications of that rule. In Prolog/RINL the corresponding code is

```
ap([coord:yes,sem:Sem]) -->
    ap([coord:no,sem:AP1]),
    [and],
    ap([coord:_,sem:AP2]),
    {combine(ap:Sem,[ap:AP1,ap:AP2])}.
```

where `coord:no` blocks the rule from being applied again. Similarly, the rule can be blocked in Haskell/CSFP by passing `False` to the `parseAP` function:

```
andAPRule :: PARSER Cat Cat
andAPRule = \xs ->
    [ (Branch (Cat "_" "AP" []) [p1,and,p2],as) |
      (p1,ys)  <- parseAP False xs,
      (and,zs) <- leafP "AND" ys,
      (p2,as)  <- parseAP True zs ]
```

The parser output differs among the systems as well: Prolog/RINL directly applies the semantics construction and generates logical expressions, GF generates abstract syntax trees, and Haskell/CSFP creates a parse tree, i.e. nodes are decorated with syntactic categories (`<S>`, `<NP>`, etc.) rather than the names of the rules that generated them (compare also with Sections 2.3.2 and 2.3.4). Figure 7.1 illustrates the resulting AST and parse tree for our example sentence.

The example also indicates that GF implementations are significantly shorter than implementations in Prolog/RINL or Haskell/CSFP. The Resource Grammar Library can help with the development of more complex grammars and (as far as we know) there is no equivalent library for Prolog or Haskell. Another advantage of GF is the separation of abstract and concrete syntax, which makes it easy to add further languages to an existing pipeline.

7.1.2 Lexicon

In GLF, adding a new word requires one entry in each, the abstract syntax, the discourse domain theory and the semantics construction view, as well as one entry for each concrete syntax. Prolog/RINL and Haskell/CSFP, on the other hand, require one entry for each word form. For example, the original Prolog/RINL grammar would introduce the word "*laugh*" with

```
lexEntry(iv,[symbol:laugh,syntax:[laugh],inf:inf,num:sg]).
lexEntry(iv,[symbol:laugh,syntax:[laughs],inf:fin,num:sg]).
lexEntry(iv,[symbol:laugh,syntax:[laugh],inf:fin,num:pl]).
```

The original Haskell/CSFP, which only uses the past form, has the corresponding entries

```
lexicon "laughed"   = [Cat "laughed"   "VP" [Tense] []]
lexicon "laugh"     = [Cat "laugh"     "VP" [Infl]  []]
```

The equivalent lines in GLF would be:

```
laugh : V;              -- abstract syntax
laugh = mkV "laugh";    -- concrete syntax
laugh : ι → o ❙         // discourse domain theory ❙
laugh = laugh ❙         // semantics construction ❙
```

Note that the lexicon format proposed in Section 8.1 would shorten this to the single line

```
v laugh
```

In GLF the symbol name for the new entry is explicitly introduced in the discourse domain theory. Prolog/RINL also explicitly introduces symbol names (`symbol:laugh`). However, Haskell/CSFP does not do that, which results in different symbols for different verb forms:

```
*P> P.process "Alice laughed"
[laughed[alice]]
*P> P.process "Alice didn't laugh"
[~laugh[alice]]
```

7.1.3 Target Logic

The (syntax of the) target logic can be represented easily in any framework – at least if it is something simple like first-order logic. However, Prolog does not natively support λ-functions or higher-order abstract syntax. Therefore, Prolog/RINL requires a custom implementation for substitution. An advantage of MMT is of course its foundation-independence, allowing GLF users to work with custom logical frameworks.

7.1.4 Semantics Construction

As an example, we will again look at the rule that combines two adjective phrases with the word "*and*". In GLF the semantics construction can be defined as

```
ap1 = [a,b] [x] (a x) ∧ (b x)
```

The Prolog/RINL implementation is similar in the sense that it also directly receives the semantic representations of the constituents:

```
combine(ap:lambda(X,and(apply(A,X),apply(B,X))), [ap:A,ap:B]).
```

Since Prolog does not natively support λ-functions, Prolog/RINL comes with custom code for α- and β-conversion. Another issue with Prolog is that it does not have a type-checker. Experience has shown that MMT's type-checking is very useful for detecting errors in the semantics construction early.

In Haskell/CSFP the semantics construction is based on pattern matching on the parse tree (Figure 7.1a):

```
transAP (Branch (Cat _ "AP" _) [p1,and,p2]) =
    \ t -> And (transAP p1 t) (transAP p2 t)
```

That is very similar to the implementation in pure GF, except that the GF semantics construction acts on an abstract syntax tree:

```
transAP (ap1 a b) = \x -> And (transAP a x) (transAP b x);
```

7.1.5 Conclusion

When it comes to grammar development, GF as a dedicated framework appears to be significantly more convenient than hand-writing parsers in Prolog/Haskell. The difference should be even more pronounced for more complex examples. Overall, the Prolog/RINL test implementation was surprisingly short. This may partly be because it is not typed and partly because the parsing directly results in a semantic representation and no intermediate representation (parse tree/AST) is generated. The advantage of typed implementations in other frameworks (including GLF) is that many mistakes can already be caught during compilation. Another issue with Prolog is that it does not natively support λ-functions and higher-order abstract syntax. Haskell is of course typed and supports λ-functions natively. However, the Haskell/CSFP implementation is by far the longest – a clear disadvantage for rapid prototyping.

Using pure GF results in a framework similar to GLF. Apart from having the same grammar development capabilities (after all, GLF uses GF exactly for that), the semantics construction is also relatively compact. The main selling point of GLF would be the foundational flexibility that MMT brings – and possibly the convenience of introducing notations in MMT's surface syntax.

A disadvantage of GLF is that it requires learning two different languages: GF and MMT. For GLIF, this increases to three languages. Simultaneously, the installation

is harder, though that has been partly mitigated through the use of Jupyter notebooks, which can be accessed online. Since GLF is not based on a general-purpose programming language, the user is restricted in what is possible to implement. For example, currently there is not much preprocessing functionality for strings. Ultimately, these disadvantages have to be weighed against the advantages of having a specialized framework that allows for the quick implementation of language-to-logic pipelines. Especially for prototypes, ease of implementation should be a priority.

7.2 Jupyter Interface

> ***Disclaimer*** *[SAK20] contains already an evaluation of the GLF Jupyter interface. It is a bit shorter and was largely written by me.*

The Jupyter interface (Chapter 5) is intended as the main way of using GLIF.

Technical Considerations Originally, the installation of GLF and the Jupyter interface was relatively tricky. The interface was used in a one-semester course on logic-based language processing [LBS]. In that time, several problems were resolved and the kernel now runs on Linux, Mac OS and Windows. It appears, however, that ELPI does not run on Windows (but it can be used in the Windows Subsystem for Linux). To further improve accessibility (the installation is still not that easy), there is now a docker image and an online demo. For a link to the online demo, see [GLIF]. The online demo makes it very easy to let curious readers try out GLIF and to share demos of NLU pipelines implemented in GLIF.

Implementing Pipelines in Notebooks For smaller projects, the notebooks turned out to be extremely useful for the implementation. The main reason is that everything is in one place. Without notebooks, separate files are needed for the abstract syntax, concrete syntax, MMT content and ELPI content – possibly in different editors/IDEs. For larger projects, on the other hand, the notebooks became too unwieldy and it made more sense to implement the pipeline elsewhere. But even for larger projects, Jupyter notebooks were the go-to solution for experimenting with specific challenges. The stub generation is extremely useful and significantly speeds up the development. Unfortunately, additional changes in the abstract syntax don't get propagated into the concrete syntaxes and semantics construction view.

Testing Pipelines in Notebooks Overall, the notebooks are very useful for testing pipelines implemented in GLIF. And with the online demo, it is even possible for other people to test GLIF pipelines without installing anything on their machine. There are, however, a few things that could be improved. First of all, the interface is still geared towards sentence-level processing, which makes case studies like the one presented in Section 6.2 a bit messier. Another problem is that the ASTs are not connected to the logical expressions obtained through the semantics construction.

That way, linguistic cues are lost for the semantic-pragmatic analysis. Furthermore, it means that case studies like the one presented in Section 6.3 (discarding contradictory readings for better translations) cannot be tested in the Jupyter interface. At last, it turns out that it is often desirable to test different examples with the same sequence of commands. For instance, a notebook that illustrates how GLForTheL (Section 6.1.2) processes statements has many copies of the following command with just the sentence replaced, which clutters the notebook:

```
parse -cat=Statement "there exists an empty set" |
    construct -v semantics/forthelUnsortedSem
```

Note that the GLIF kernel currently does not support multi-line commands. Most of these issues are addressed in the future work section (Chapter 8).

Teaching GLIF was originally created along a lecture on logic-based language processing [LBS] to make it easier to implement and try out the theoretical considerations. Early predecessors of GLIF were in the form of Scala scripts that required a lot of pipeline- and system-specific boilerplate code. Both sharing and implementing pipelines has gotten a lot easier since. The Jupyter notebooks have been used for interactive classroom work and homework assignments. The results of the classroom work were usually notebooks that implement a Montague fragment or analyse a specific issue. After the lecture, the notebooks could be cleaned up, extended with markdown descriptions and shared with the students. Sharing a single notebook file is significantly easier than sharing a collection of GF grammar modules and MMT files. For homework assignments, the notebooks can contain explanations as well as a skeleton of the implementation and commands for testing the implementation – all in the same document (Figure 7.2). The students were free to choose between using Jupyter notebooks and their own editors + command line tools for their homework. Most students preferred using Jupyter notebooks.

7.3 Limitations in MMT

In Section 7.1 we have already seen that GF is a good choice for grammar development and parsing. For MMT, we have only briefly mentioned that its foundation-independence and modularity make it a good fit. While this is certainly true, there are a number of issues with MMT that should be mentioned.

Types for Complex Semantics Construction During the GLForTheL project (Section 6.1.2) it became obvious that it sometimes is really difficult to implement the semantics construction in MMT. [Sch20] discusses the problem in great length. We will just briefly hint at the problems here. To handle sentences like

"*A, B and C are pairwise disjoint groups*",

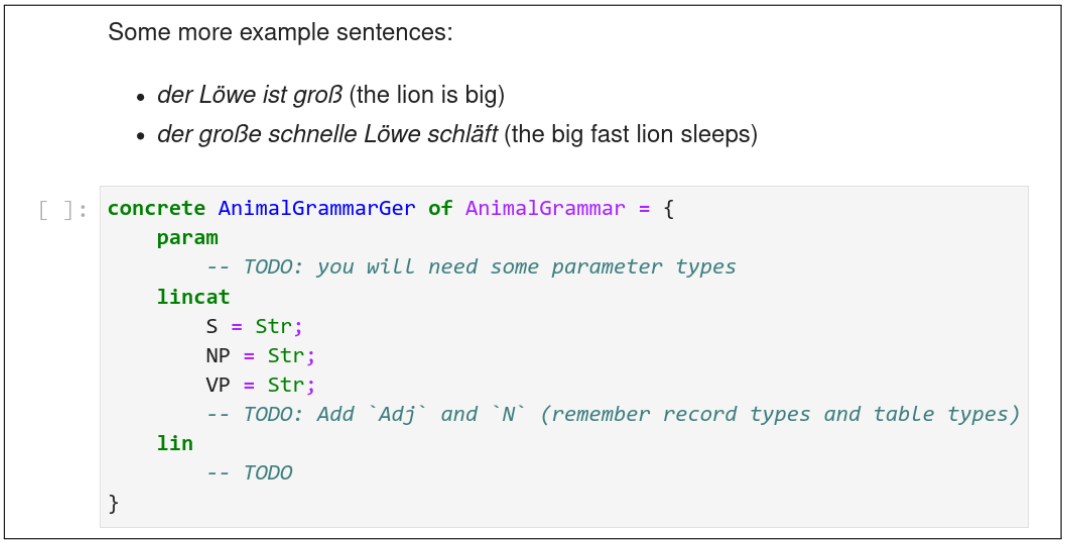


FIGURE 7.2: Example notebook for homework assignment.

the semantics construction needed to work with lists and pairs. The Church encoding of a pair $\langle a, b\rangle$ is $\lambda f.f\,a\,b$. Let us say $a : \alpha$ and $b : \beta$. Then

$$\langle a, b\rangle \;:\; \Pi_{\gamma:type}(\alpha \to \beta \to \gamma) \to \gamma,$$

which is a problem, because the "type" of a pair is now *kind*. Things get even more complicated if we want to have e.g. lists of pairs. While it is possible to make these things work, it gets quite tricky and requires extensions of LF, both for the language theory and the target logic. An alternative approach is to only generate a preliminary expression in the semantics construction and then create the final expression using ELPI. [Sch20] contains more details.

Discourse Representation Theory While MMT can be used to implement many different logics, an important logic is missing: **discourse representation theory** (**DRT**) [KR93]. DRT (and variants of it) address, among other things, scoping problems that arise from pronouns. For example, when translating

"*A man runs. He is fast.*"

compositionally, we would obtain an expression like

$$(\exists x.\mathrm{man}(x) \wedge \mathrm{run}(x)) \quad \wedge \quad (\mathrm{fast}(x))$$

where the last occurrence of x is outside the scope of the existential quantifier. Given how established DRT is, it would be great if it can be easily used in GLIF. Unfortunately, that is currently not possible and more work is needed to make it work.

Semantics Construction Must Result in Well-Formed Expressions The way the semantics construction is setup in GLIF, every AST is mapped to a well-formed logical expression. However, that is not always desirable. We can illustrate

that with an idea that came up in the GLForTheL project (Section 6.1.2): we briefly considered to use many-sorted first-order logic instead of single-sorted first-order logic (see also [Sch20]). For example, the sentence "*there is an empty set*" would result in the expression $\exists X^{set}.empty(X)$, i.e. X is of sort *set* and *empty* expects an argument of sort *set*. The problem are sentences like "*there is an even set*" that are grammatical but cannot be mapped to well-formed logical expressions (*even* might require an argument of sort *integer*).

We can actually implement the example by introducing dependent types to the grammar:

```
cat S; Sort; Adj Sort;
fun
    set, integer : Sort;
    empty : Adj set;
    even : Adj integer;
    exists : (s : Sort) -> Adjective s -> S;
```

GF would then discard sentence like "*there is an even set*" that don't type-check. Since both the abstract syntax and the (equivalent) language theory can only represent ASTs that correspond to well-formed logical expressions, we can now express the semantics construction as a view.

While the use of dependent types solves the problem for this example, it does not solve the problem in general. Furthermore, the grammar should not have to be designed with the semantics construction in mind.

7.4 Semantic/Pragmatic Analysis

The semantic/pragmatic analysis (unlike parsing and semantics construction) does not follow a particular recipe – which was the reason to use a general programming language like ELPI. Maybe such a standard recipe will emerge from future research but that is pure speculation at this point. A common theme of the semantic/pragmatic analysis is the need for theorem proving, which is, of course, facilitated by the automated generation of ELPI theorem provers. However, the prover generation is still at an early stage. Nevertheless, one of the case studies (Section 6.3) actually used a generated prover.

The GLForTheL case study (Section 6.1.2, [Sch20]) has shown that sometimes the line between the semantics construction and processing in ELPI blurs when certain aspects of the semantics construction are offloaded to ELPI because it would be too difficult to express them in the semantics construction view.

The case studies have also shown that the semantic/pragmatic analysis can benefit from syntactic information. For the tableaux machine (Section 6.4), anaphor resolution required information on how recently entities where referenced. A more advanced implementation would also need information on e.g. parallel sentence structures. Furthermore, we need information on the grammatical gender (and the agreement more general) for anaphor resolution. In the case studies we got along

by adding the grammatical gender to the logical expressions – and by not properly distinguishing between biological and grammatical gender. In general, however, the relevant syntactic information is highly language-specific and much information is already lost in the AST. In summary, it is unclear how syntactic information should be provided to the semantic/pragmatic analysis.

CHAPTER 8

Future Work

In this chapter we will discuss a few ideas how GLIF should be improved. They have come up in my practical work on the case studies.

8.1 Lexicon Generation

Often, GLIF projects start out with a minimal vocabulary that is used for testing. Once everything works, more and more words need to be added to increase the number of meaningful sentences that are covered. Adding a new word is always a bit of work since four new entries are required, as illustrated here with the word "*lion*":

- Abstract syntax: `lion : Animal;`
- Concrete syntax: `lion = mkAnimal "lion";`
- Discourse domain theory: `lion : Animal`
- Semantics construction: `lion = lion`

Based on this example, it looks as if the relevant entries could be generated automatically from a lexicon entry

```
Animal lion
```

A first attempt in this direction was sketched in [Sch20], however, it can be simplified and improved. For example, by requiring that syntactic categories, constructors in the concrete syntax and types in the discourse domain theory are named consistently, we can drop the need for a pipeline-specific specification file.

So far, we have assumed that the constructor simply requires one argument: the symbol name as a string (`mkAnimal "lion"`). That is a good default, but in general we want to be able to pass custom arguments:

```
Animal mountainGorilla "mountain gorilla"
Animal tiger "tiger" "tigress"
```

GF handles different argument combinations via constructor overloading (as is done in the Resource Grammar Library). If desirable, this lexicon format could even be extended to multiple languages:

```
Animal wolf -eng "wolf" "wolves"
            -ger "Wolf" "Wölfe" masculine
            -rus "волк"
```

From such specifications it should be possible to generate an abstract syntax, concrete syntaxes, a discourse domain theory and a semantics construction view, which can be imported into the remaining pipeline. It would also be interesting to see if such a specification could be generated automatically from existing lexical resources.

8.2 GLIF Commands

GLIF commands can be used for parsing sentences, visualizing parse trees, applying the semantics construction, generating ELPI code, etc. Originally, the Jupyter kernel simply supported GF shell commands, which where passed to the GF shell and the output was displayed in the notebook. Over time, more commands where added, which are handled by the kernel directly. It appears that a re-design is necessary to allow for multi-sentence processing and to have logical representations associated with the ASTs and natural language representations. This association is e.g. needed to test the translation pipeline described in Section 6.3. In Section 8.2.1 we will see how using structured data could solve that problem. Afterwards, in Section 8.2.2, we will briefly discuss how user-defined commands could be introduced.

8.2.1 Using Structured Data

To illustrate the problem, we will use the following example command

```
parse -Lang=Eng "John works at a bank"
                "John is a professional goose watcher" |
    construct | elpi ... | linearize -Lang=Ger
```

that currently wouldn't work in GLIF. The intention behind the command is to translate the sentences to German, which requires disambiguation of the word "*bank*". The **parse** command returns three ASTs:

```
work_at john bank_institute
work_at john bank_river
has_job john goose_watcher
```

Since GLIF commands only use plain strings, it is completely lost that the first two ASTs correspond to the same string. The semantics construction can still be applied but the inference step definitely needs that information to discard one reading. Even if that can be solved, there is another problem: the result of the inference step is a sequence of logical expressions – not ASTs, which would be needed for the linearization.

One solution would be to replace the plain string representation by structured data. The output of a command would then be a list of entries, each of which has

- a reference to the natural language string (if applicable),
- an AST (if applicable),

- a logical expression (if applicable),
- a default representation (corresponding to the string representation in the current implementation).

If that output gets piped into another command, the new command can pick which representation is most useful. For example, **linearize** would always pick the AST representation.

The problem that multiple trees can correspond to one sentence could also be solved by introducing a new representation for the parser output that contains all possible readings. For example, the sentence "*John works at a bank*" could be represented as

```
work_at john (bank_institute | bank_river)
```

8.2.2 User-Defined Commands

It often makes sense to run a particular sequence of commands on different inputs. For example, testing the pipeline implemented in Section 6.3 meant running the command

```
parse -cat=QSeq "..." | construct -elpi | elpi -no-tc ...
```

on many different examples. Instead of having to copy the commands for every example, it would be much more convenient if the user could define a custom command:

```
define testQA[1] =          # requires 1 argument (?0)
    parse -cat=QSeq ?0 | construct -elpi | elpi -no-tc ...
```

This is similar to the GF shell command **define_command**.

User-defined commands come with a challenge: currently, the content of Jupyter notebook cells is identified via pattern matching. Not just for executing the cells but also for syntax highlighting – and we probably don't want the syntax highlighter to keep track of what commands have been introduced.

One possibility is to mark user-defined commands with a special character when they are called (e.g. `?testQA`). Another option would be to assume that any code cell that cannot be identified contains commands.

Ideally, the distinction of different cell types (GF/MMT/ELPI/GLIF command) would happen at the Jupyter level and not on the kernel level. Currently, Jupyter does not seem to support different types of code cells.

8.3 Further Ideas

In this section we will briefly discuss a number of other ideas for future work.

GLIF Usability Without Jupyter Especially for larger projects, GLIF is often used outside of Jupyter notebooks. Unfortunately, using GLIF directly is currently not very convenient. Therefore, it would make sense to create GLIF library (Python-based) which provides all the necessary functionality (command processing, stub generation, etc.). Then the kernel would be based on that library and GLIF could also be used from the command line or in custom Python scripts.

Connecting to Theorem Provers As interesting as the possibility to generate provers from MMT (Section 4.2) is, for many applications it would be much easier to use an off-the-shelf optimized theorem prover, e.g. for first-order theorems. For example, the $S5_n$ formulae used in the case study presented in Section 6.2 could be easily translated to first-order logic. Having that option can probably make the prototyping of many ideas much faster – even if we lose decidability in the case of $S5_n$.

MMT Extensions As discussed already in the evaluation (Section 7.3), one of the main short-comings of GLIF is that it does not properly support discourse representation theory. Adding support would probably be a significant step towards establishing GLIF in its role as an NLU framework. [Sch20] also brought up the need for handling sequences in MMT. While a corresponding LF extension exists (LFS, see [Hor14]), it is rather brittle. Therefore, a stable reimplementation is desirable.

GF String Continuation Suggestions GF can use incremental parsing to suggest words that could continue a string [GFT10]. Incorporating that into GLIF could be very beneficial as it can help with entering sentences that will be accepted by the grammar. When GLIF is used for the implementation of controlled natural languages (see also Section 6.1), it would to some extent alleviate the problem that users have to get an intuition about the supported fragment.

Propagate Changes in Abstract Syntax to Generated Stubs Generating stubs for the concrete syntaxes and semantics construction (Section 5.2.3) turned out to be a very useful feature. The stubs are generated from the abstract syntax. Unfortunately, the stubs currently can't be updated if the abstract syntax changes. As it is common to start with a minimal abstract syntax and then extend it when everything works, this limits the usefulness of stub generation and a mechanism to update generated (and filled out) stubs would certainly be useful.

CHAPTER 9

Conclusion

We have seen how GF, MMT and ELPI can be combined into a framework for prototyping NLU pipelines. We have evaluated the framework in a number of case studies. Let us review what that means for the research questions posed in the introduction.

Question. *Can we combine existing tools into a framework for prototyping the translation from natural language into a semantic representation?*

Answer. Yes. GLF is an example for that.

Discussion. GLF combines GF and MMT into a framework for implementing compositional language-to-logic translation in a Montagovian fashion. GLF ($\subset$ GLIF) was successfully used in all the case studies. However, I claim that further processing in the form of a semantic/pragmatic analysis is necessary for true natural language understanding, which brings us to the next research question.

Question. *Can we extend it to allow for semantic/pragmatic analysis?*

Answer. Yes. GLIF (= GLF+ELPI) allows to also prototype the semantic/pragmatic analysis.

Discussion. There is no "standard recipe" for the semantic/pragmatic analysis and therefore the flexibility of a programming language like ELPI seems appropriate (as opposed to the more specialized/restricted parsing and semantics construction steps). A full-fledged semantic/pragmatic analysis presumably has to perform many different tasks: enriching the semantic representation (e.g. by resolving anaphora), pruning wrong readings, etc. In the case studies, we used the semantic/pragmatic analysis to maintain a discourse model, which involved pronoun resolution (Section 6.4), and to discard contradictory readings (Section 6.3).

Question. *Can the implementation of the semantic/pragmatic analysis be (partly) automated to facilitate rapid prototyping?*

Answer. To some extent, yes.

Discussion. As the semantic/pragmatic analysis often requires inferential reasoning, it benefits from automated theorem provers. In Section 2.6 we have extensively discussed the possibility of automatically generating ELPI theorem provers from calculi specified in MMT. While this is certainly an exciting idea, there are a few caveats. The prover generation has not moved much beyond the proof-of-concept stage and it has not been tested on logics other than (subsets of) first-order logic. Another issue is the need for a natural deduction–style calculus specification in MMT, which is difficult for some logics. Nevertheless, one of the case studies (Sec-

tion 6.3) actually uses an automatically generated prover as that was easier than manually implementing one.

Final Remarks Even though GLIF can be improved in many ways (Chapter 8), it has already been used successfully in a number of case studies as well as in a class on logic-based natural language semantics. Just trying to establish a GLF implementation has uncovered problems in a published result that had gone unnoticed.

The GLIF Jupyter kernel, including a link to an online demo can be found at [GLIF].

Bibliography

[AG09] Pietro Abate and Rajeev Goré. "The tableau workbench". In: *Electronic Notes in Theoretical Computer Science* 231 (2009), pp. 55–67.

[Ang11] Krasimir Angelov. "The Mechanics of the Grammatical Framework". PhD thesis. Chalmers University of Technology, Gothenburg, Sweden, 2011. URL: http://www.cse.chalmers.se/~krasimir/phd-thesis.pdf.

[BB05] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, 2005.

[Boe06] Mathijs de Boer. "Praktische bewijzen in public announcement logica (Practical proofs in public announcement logic)". Master's Thesis. Department of Artificial Intelligence, University of Groningen, 2006.

[Bre+82] Joan Bresnan, Ronald M Kaplan, Stanley Peters, and Annie Zaenen. "Cross-serial dependencies in Dutch". In: *The formal complexity of natural language*. Springer, 1982, pp. 286–319.

[CMR13] Z. Chihani, D. Miller, and F. Renaud. "Checking Foundational Proof Certificates for First-Order Logic". In: *Proof Exchange for Theorem Proving*. Ed. by J. Blanchette and J. Urban. 2013, pp. 58–66.

[Cod+11] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, and Florian Rabe. "Project Abstract: Logic Atlas and Integrator (LATIN)". In: *Intelligent Computer Mathematics*. Ed. by James Davenport, William Farmer, Florian Rabe, and Josef Urban. LNAI 6824. Springer Verlag, 2011, pp. 289–291. URL: https://kwarc.info/people/frabe/Research/CHKMR_latinabs_11.pdf.

[Coo+] Robin Cooper, R Crouch, Jan Van Eijck, Chris Fox, Josef Van Genabith, Jan Jaspars, Hans Kamp, David Milward, Manfred Pinkal, Massimo Poesio, et al. *Using the framework. The FraCaS Consortium (1996)*. Tech. rep. FraCaS deliverable D-16. URL: http://chris.foxearth.org/papers/fracas/del16.pdf.

[Cre82] MJ Cresswell. *The Autonomy of Semantics*. 1982.

[Dav67] Donald Davidson. "Truth and Meaning". In: *Synthese* 17 (1967).

[Dun+15] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "ELPI: Fast, Embeddable, λProlog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2015, pp. 460–468.

[Elp] *ELPI Playground*. URL: https://gl.mathhub.info/elpi/elpi-playground (visited on 10/19/2020).

[EU10] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.

[FSS98] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. "Attempto Controlled English - Not Just Another Logic Specification Language". In: *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation*. LOPSTR '98. Springer-Verlag, 1998, pp. 1–20.

[GF] *Grammatical Framework – A programming language for multilingual grammar applications*. URL: https://www.grammaticalframework.org/ (visited on 10/31/2020).

[GFT10] Aarne Ranta. *Grammatical Framework Tutorial*. Dec. 2010. URL: https://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html (visited on 12/01/2019).

[GLIF] *GLIF Kernel*. URL: https://github.com/KWARC/GLIF (visited on 03/22/2020).

[GSCT19] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing type theory in higher order constraint logic programming". In: *Mathematical Structures in Computer Science* 29.8 (2019), pp. 1125–1150.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HM92] Joseph Y Halpern and Yoram Moses. "A guide to completeness and complexity for modal logics of knowledge and belief". In: *Artificial Intelligence* 54.3 (1992), pp. 319–379.

[Hor14] Fulya Horozal. "A Framework for Defining Declarative Languages". PhD thesis. Jacobs University Bremen, Nov. 2014. URL: https://gl.kwarc.info/supervision/PhD-archive/horozal_fulya/phd.pdf.

[KK00] Michael Kohlhase and Alexander Koller. "Towards A Tableaux Machine for Language Understanding". In: *Proceedings of Inference in Computational Semantics ICoS-2*. Ed. by Johan Bos and Michael Kohlhase. Computational Linguistics, Saarland University, 2000, pp. 57–88.

[Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Koh+20a] Michael Kohlhase, Florian Rabe, Claudio Sacerdoti Coen, and Jan Frederik Schaefer. "Logic-Independent Proof Search in Logical Frameworks (extended report)". 2020. URL: https://kwarc.info/kohlhase/submit/mmtelpi.pdf.

[Koh+20b] Michael Kohlhase, Florian Rabe, Claudio Sacerdoti Coen, and Jan Frederik Schaefer. "Logic-Independent Proof Search in Logical Frameworks (short paper)". In: *10th International Joint Conference on Automated Reasoning*. Springer Verlag, 2020.

[KR93] Hans Kamp and Uwe Reyle. *From Discourse to Logic: Introduction to Model-Theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer, 1993.

[KS] Michael Kohlhase and Jan Frederik Schaefer. "GF + MMT = GLF – From Language to Semantics through LF". In: *LFMTP 2019, Proceedings*. Electronic Proceedings in Theoretical Computer Science (EPTCS). URL: https://kwarc.info/kohlhase/submit/lfmtp-19.pdf.

[LATIN] *LATIN2 – Logic Atlas Version 2*. URL: https://gl.mathhub.info/MMT/LATIN2 (visited on 06/02/2017).

[LBS] Michael Kohlhase. *Logic-Based Natural Language Processing – Lecture Notes*. URL: https://kwarc.info/teaching/LBS/notes-WS1819.pdf (visited on 12/26/2019).

[LDM12] Hector Levesque, Ernest Davis, and Leora Morgenstern. "The Winograd Schema Challenge". In: *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2012.

[Lju04] Peter Ljunglöf. "Expressivity and complexity of the Grammatical Framework". PhD thesis. 2004.

[Mit] *MitM/Foundation*. URL: https://gl.mathhub.info/MitM/Foundation (visited on 10/28/2020).

[MKR20] Richard Marcus, Michael Kohlhase, and Florian Rabe. "TGView3D: a System for 3-Dimensional Visualization of Theory Graphs". In: *Intelligent Computer Mathematics (CICM) 2020*. Conferences on Intelligent Computer Mathematics. LNAI. Springer, 2020, pp. 290–296. URL: https://kwarc.info/kohlhase/papers/cicm20-tgview3d.pdf.

[MMTa] *MMT Repository on GitHub*. URL: https://github.com/UniFormal/MMT (visited on 10/31/2020).

[MMTb] *The MMT Language and System*. URL: https://uniformal.github.io/ (visited on 10/31/2020).

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.

[Mon70] R. Montague. "English as a Formal Language". In: Reprinted in [Tho74], 188–221. Edizioni di Communita, Milan, 1970. Chap. Linguaggi nella Societa e nella Tecnica, B. Visentini et al eds, pp. 189–224.

[MR19] Dennis Müller and Florian Rabe. "Rapid Prototyping Formal Systems in MMT: Case Studies". In: *LFMTP 2019*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2019. URL: https://kwarc.info/people/frabe/Research/MR_prototyping_19.pdf.

[Pas07a] Andrei Paskevich. *The syntax and semantics of the ForTheL language*. English excerpt from [Pas07b]. 2007.

[Pas07b] Andriy Paskevych. "Méthodes de formalisation des connaissances et des raisonnements mathématiques: aspects appliqués et théoriques". PhD thesis. Université Paris 12, 2007.

[Ran11] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). Stanford: CSLI Publications, 2011.

[Ran94] Aarne Ranta. *Type-Theoretical Grammar*. Clarendon Press, 1994.

[RGL] *GF Resource Grammar Library: Synopsis*. URL: https://www.grammaticalframework.org/lib/doc/synopsis/index.html (visited on 11/29/2019).

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[RS19] Rasmus Rendsvig and John Symons. "Epistemic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2019. Metaphysics Research Lab, Stanford University, 2019.

[Sad] *SAD repository*. URL: https://github.com/tertium/SAD (visited on 07/23/2020).

[Sag] *SageMath*. URL: https://www.sagemath.org/ (visited on 10/08/2020).

[SAK20] Jan Frederik Schaefer, Kai Amann, and Michael Kohlhase. "Prototyping Controlled Mathematical Languages in Jupyter Notebooks". In: *Mathematical Software – ICMS 2020. 7th international conference*. Ed. by Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff. Vol. 12097. LNCS. Springer, 2020, pp. 406–415. URL: https://kwarc.info/kohlhase/papers/icms20-glf-jupyter.pdf.

[Sch20] Jan Frederik Schaefer. *Implementing ForTheL in GLIF: A Case Study*. Master's Project Report. 2020. URL: https://gl.kwarc.info/supervision/projectarchive/-/blob/master/2020/Schaefer_Frederik.pdf.

[SCT15] Claudio Sacerdoti Coen and Enrico Tassi. *The ELPI system*. 2015. URL: https://github.com/LPCIC/elpi.

[Sha92] S.C. Shapiro. *Encyclopedia of Artificial Intelligence*. Wiley-Interscience, 1992.

[Shi85] Stuart M Shieber. "Evidence against the context-freeness of natural language". In: *Philosophy, Language, and Artificial Intelligence*. Springer, 1985, pp. 79–89.

[SK20] Jan Frederik Schaefer and Michael Kohlhase. "GLIF: A Declarative Framework for Symbolic Natural Language Understanding". In: *Proceedings of the 6th Workshop on Formal and Cognitive Reasoning*. Ed. by Christoph Beierle, Marco Ragni, Frieder Stolzenburg, and Matthias Thimm. 2020, pp. 4–11. URL: http://ceur-ws.org/Vol-2680/paper1.pdf.

[Tho74] R. Thomason, ed. *Formal Philosophy: selected Papers of Richard Montague*. Yale University Press, New Haven, CT, 1974.

[VLP07] Konstantin Verchinine, Alexander Lyaletski, and Andrei Paskevich. "System for Automated Deduction (SAD): a tool for proof verification". In: *Automated Deduction, 21st International Conference, CADE-21*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Bremen, Germany: Springer, July 2007, pp. 398–403.

APPENDIX A

Example in Different Frameworks

This appendix contains the complete example implementations used for comparing different frameworks for natural language semantics in Section 2.4. For brevity, we have omitted lexing and start right away with a token sequence. Section 7.1 evalutes GLF based on this comparison.

A.1 Prolog/RINL

The Prolog/RINL approach is explained in Section 2.4.1. You can test the example implementation with queries like

```
?- run([john,and,mary,are,happy,and,smart],X).
```

Since we don't have β-reduction, that example query results in a rather complex expression with many lambdas. Here is the actual code:

```
% GRAMMAR

% <S> ::= <S> "and" <S>
% coord avoids infinite loops (rule is left-recursive)
s([coord:yes,sem:Sem]) -->
    s([coord:no,sem:S1]),
    s([coord:_,sem:S2]),
    {combine(s:Sem,[s:S1,s:S2])}.

% <S> ::= <NP> <COPULA> <AP>
% <COPULA> is "is"/"are"
s([coord:_,sem:Sem]) -->
    np([num:Num,sem:NP]),
    copula([num:Num]),  % Num ensures that the number matches
    ap([coord:_,sem:AP]),
    {combine(s:Sem,[np:NP,ap:AP])}.

% <NP> ::= <NP> "and" <NP>
np([num:pl,sem:Sem]) -->
    np([num:sg,sem:NP1]),
    [and],
    np([num:_,sem:NP2]),
    {combine(np:Sem,[np:NP1,np:NP2])}.

% <NP> ::= <PN>
% where PN is a lexicon entry
np([num:sg,sem:Sem]) -->
    {lexEntry(pn,[symbol:Sym,syntax:Word])},
    Word,
    {semLex(pn,[symbol:Sym,sem:Sem])}.

% <AP> ::= <AP> "and" <AP>
ap([coord:yes,sem:Sem]) -->
    ap([coord:no,sem:AP1]),
    [and],
```

```
    ap([coord:_,sem:AP2]),
    {combine(ap:Sem,[ap:AP1,ap:AP2])}.

% <AP> ::= <ADJ>
% were adj is a lexicon entry
ap([coord:_,sem:Sem]) -->
    {lexEntry(adj,[symbol:Sym,syntax:Word])},
    Word,
    {semLex(adj,[symbol:Sym,sem:Sem])}.

% <COPULA> ::= <COP>
% where COP is a lexicon entry ("is"/"are")
% For the sake of conciseness, copulae don't have a meaning here.
copula([num:Num]) -->
   {lexEntry(cop,[num:Num,syntax:Word])},
   Word.

% LEXICON

% Note: The symbol distinction becomes relevant,
% if we have e.g. different verb forms.
% E.g. "run" and "runs" should have the same symbol.
lexEntry(pn,[symbol:john,syntax:[john]]).
lexEntry(pn,[symbol:mary,syntax:[mary]]).
lexEntry(adj,[symbol:happy,syntax:[happy]]).
lexEntry(adj,[symbol:sad,syntax:[sad]]).
lexEntry(adj,[symbol:smart,syntax:[smart]]).
lexEntry(cop,[num:sg,syntax:[is]]).
lexEntry(cop,[num:pl,syntax:[are]]).

% SEMANTICS CONSTRUCTION

combine(s:and(A,B),[s:A,s:B]).
combine(s:apply(NP,AP),[np:NP,ap:AP]).
combine(np:lambda(P,and(apply(A, P), apply(B, P))),[np:A,np:B]).
combine(ap:lambda(X,and(apply(A,X),apply(B,X))),
        [ap:A,ap:B]).

% Meaning of John is lambda P.P(john)
semLex(pn,[symbol:S,sem:lambda(P,apply(P,S))]).
semLex(adj,[symbol:S,sem:S]).

% CALLER PREDICATE

run(L, S) :- s([coord:_,sem:S],L,[]).
```

A.2 Haskell/CSFP

Haskell/CSFP is described in Section 2.4.2. You can test it using the `run` function:

```
> run ["John","and","Mary","are","smart","and","happy"]
```

And here is the actual code:

```
import Data.List

-- GENERIC PARSER STUFF

-- a: terminal type; b: tree type
type Parser a b = [a] -> [(b,[a])]

-- parse combinator: fail if first argument is false
disable :: Bool -> Parser a b -> Parser a b
disable True p  = p
disable False _ = \xs -> []
```

```
-- parser combinator: alternatives
infixr 4 <|>
(<|>) :: Parser a b -> Parser a b -> Parser a b
(p1 <|> p2) xs = p1 xs ++ p2 xs

data ParseTree a b = Ep | Leaf a | Branch b [ParseTree a b] deriving Eq
type PARSER a b = Parser a (ParseTree a b)

-- CATEGORIES AND FEATURES

data Feature = Masc | Fem | Sg | Pl -- and anything else one might need
    deriving (Eq,Show,Ord)
type Agreement = [Feature]

gender, number :: Agreement -> Agreement
gender = filter (`elem` [Masc,Fem])
number = filter (`elem` [Sg,Pl])

type CatLabel = String
data Cat = Cat String CatLabel Agreement deriving Eq

getAgr :: Cat -> Agreement
getAgr (Cat _ _ agr) = agr

-- extract category of tree
t2c :: ParseTree Cat Cat -> Cat
t2c (Leaf c) = c
t2c (Branch c _) = c

-- combines feature lists; [] on failure (i.e. contradicting features)
combine :: Cat -> Cat -> [Agreement]
combine cat1 cat2 =
    [ feats | length (gender feats) <= 1, length (number feats) <= 1 ]
    where feats = (nub . sort) (getAgr cat1 ++ getAgr cat2)

-- two parse trees agree wrt their features if they can be combined
agree :: ParseTree Cat Cat -> ParseTree Cat Cat -> Bool
agree a b = not (null (combine (t2c a) (t2c b)))

-- LEXICON

lexicon :: String -> [Cat]
lexicon "John"  = [Cat "John" "PN" [Masc,Sg]]
lexicon "Mary"  = [Cat "Mary" "PN" [Fem,Sg]]
lexicon "happy" = [Cat "happy" "Adj" []]
lexicon "smart" = [Cat "smart" "Adj" []]
lexicon "is"    = [Cat "is" "Cop" [Sg]]
lexicon "are"   = [Cat "are" "Cop" [Pl]]
-- every acceptable token has to be in the lexicon
lexicon "and"   = [Cat "and" "AND" []]

lookupWord :: (String -> [Cat]) -> String -> [Cat]
lookupWord db w = [ c | c <- db w ]

-- PRE-PROCESSING

-- transforms word list into cat list using lexicon
collectCats :: (String -> [Cat]) -> [String] -> [[Cat]]
collectCats db words =
    let listing = map (\ x -> (x,lookupWord db x)) words
        unknown = map fst  (filter (null.snd) listing)
    in
        if unknown /= [] then error ("unkown words: " ++ show unknown)
        else initCats (map snd listing)

-- multiply out ambiguities
initCats :: [[Cat]] -> [[Cat]]
initCats [] = [[]]
initCats (cs:rests) = [ c:rest | c <- cs, rest <- initCats rests]
```

```
-- PARSING

-- parser for leaf nodes (lexical entries)
leafP :: CatLabel -> PARSER Cat Cat
leafP label [] = []
leafP label (c@(Cat _ label2 _):cs) = [ (Leaf c,cs) | label2 == label ]

sRule :: PARSER Cat Cat
sRule = \ xs ->
    [ (Branch (Cat "_" "S" []) [np,cop,ap],as) |
      (np,ys)  <- parseNP True xs,
      (cop,zs) <- parseCop ys,
      (ap,as) <- parseAP True zs,
      agree np cop ]

andSRule :: PARSER Cat Cat
andSRule = \xs ->
    [ (Branch (Cat "_" "S" []) [s1,and,s2],as) |
      (s1,ys) <- parseS False xs,
      (and,zs)   <- leafP "AND" ys,
      (s2,as) <- parseS True zs ]

parseS :: Bool -> PARSER Cat Cat
parseS b = sRule <|> disable b andSRule

npRule :: PARSER Cat Cat
npRule = \xs ->
    [ (Branch (Cat "_" "NP" (number (getAgr (t2c pn)))) [pn],ys) |
      (pn,ys) <- leafP "PN" xs ]

andNPRule :: PARSER Cat Cat
andNPRule = \xs ->
    [ (Branch (Cat "_" "NP" [Pl]) [np1,and,np2],as) |
      (np1,ys) <- parseNP False xs,
      (and,zs) <- leafP "AND" ys,
      (np2,as) <- parseNP True zs ]

parseNP :: Bool -> PARSER Cat Cat
parseNP b = npRule <|> disable b andNPRule

parseCop :: PARSER Cat Cat
parseCop = leafP "Cop"

apRule :: PARSER Cat Cat
apRule = \xs ->
    [ (Branch (Cat "_" "AP" []) [adj],ys) |
      (adj,ys) <- leafP "Adj" xs ]

andAPRule :: PARSER Cat Cat
andAPRule = \xs ->
    [ (Branch (Cat "_" "AP" []) [p1,and,p2],as) |
      (p1,ys)  <- parseAP False xs,
      (and,zs) <- leafP "AND" ys,
      (p2,as)  <- parseAP True zs ]

parseAP :: Bool -> PARSER Cat Cat
parseAP b = apRule <|> disable b andAPRule

-- SEMANTICS CONSTRUCTION

type Term = String
data Proposition = Neg Proposition | And Proposition Proposition |
                   ApplyUnPred String Term deriving (Eq,Ord,Show)

transS :: ParseTree Cat Cat -> Proposition
transS (Branch (Cat _ "S" _) [s1, Leaf (Cat _ "AND" _), s2]) =
    And (transS s1) (transS s2)
transS (Branch (Cat _ "S" _) [np, Leaf (Cat _ "Cop" _), ap]) =
    (transNP np) (transAP ap)

transNP :: ParseTree Cat Cat -> (Term -> Proposition) -> Proposition
```

```
transNP (Branch (Cat _ "NP" _) [np1, Leaf (Cat _ "AND" _), np2]) =
    \ p -> And (transNP np1 p) (transNP np2 p)
transNP (Branch (Cat _ "NP" _) [pn]) =
    \ p -> p (transPN pn)

transPN :: ParseTree Cat Cat -> Term
transPN (Leaf (Cat s "PN" _)) = s

transAP :: ParseTree Cat Cat -> Term -> Proposition
transAP (Branch (Cat _ "AP" _) [p1,and,p2]) =
    \ t -> And (transAP p1 t) (transAP p2 t)
transAP (Branch (Cat _ "AP" _) [adj]) =
    transAdj adj

transAdj :: ParseTree Cat Cat -> Term -> Proposition
transAdj (Leaf (Cat s "Adj" _)) =
    \ t -> ApplyUnPred s t

run :: [String] -> [Proposition]
run ws = [ transS ps | catlist <- collectCats lexicon ws,
                       (ps,[]) <- parseS True catlist ]
```

A.3 GF

GF is explained in Section 2.4.3. The concrete syntax is based on the Resource Grammar Library. You can test it by calling:

```
p -lang=Rgl -cat=Prop "John and Mary are happy and smart" |
    pt -compute
```

The resulting logical expression is `And (And (h j) (s j)) (And (h m) (s m))`. The implementation is split into two sections, one for parsing and one for the semantics construction, because the parsing part is also used for the GLF implementation (Appendix A.4).

A.3.1 Parsing

```
abstract Rules = {
  cat
    S; NP; AP; A; PN;
  data
    s1  : S  -> S  -> S;
    s2  : NP -> AP -> S;
    np1 : NP -> NP -> NP;
    np2 : PN -> NP;
    ap1 : AP -> AP -> AP;
    ap2 : A  -> AP;
}

abstract Lexicon = Rules ** {
  data
    john, mary : PN;
    happy, smart : A;
}

abstract People = Rules, Lexicon
```

```
concrete RulesRgl of Rules = open SyntaxEng in {
  lincat
    S = S; NP = NP; AP = AP; A = A; PN = PN;
  lin
    s1 a b   = mkS and_Conj a b;
    s2 np ap = mkS (mkCl np (mkVP ap));
    np1 a b  = mkNP and_Conj a b;
    np2 pn   = mkNP pn;
    ap1 a b  = mkAP and_Conj a b;
    ap2 adj  = mkAP adj;
}

concrete LexiconRgl of Lexicon = RulesRgl ** open ParadigmsEng in {
  lin
    john  = mkPN "John";
    mary  = mkPN "Mary";
    happy = mkA "happy";
    smart = mkA "smart";
}

concrete PeopleRgl of People = RulesRgl, LexiconRgl
```

A.3.2 Semantics Construction

```
abstract Logic = {
  cat
    Prop; Term;
  fun
    Not : Prop -> Prop;
    And : Prop -> Prop -> Prop;
}

abstract DiscourseDomainTheory = Logic ** {
  fun
    j,m : Term;
    h,s : Term -> Prop;
}

abstract Semantics = People, DiscourseDomainTheory ** {
  fun
    transS  : S -> Prop;
    transNP : NP -> (Term -> Prop) -> Prop;
    transAP : AP -> Term -> Prop;
    transPN : PN -> Term;
    transA  : A -> Term -> Prop;
  def
    transS (s1 a b) = And (transS a) (transS b);
    transS (s2 np ap) = (transNP np) (transAP ap);
    transNP (np1 a b) = \p -> And (transNP a p) (transNP b p);
    transNP (np2 pn) = \p -> p (transPN pn);
    transAP (ap1 a b) = \x -> And (transAP a x) (transAP b x);
    transAP (ap2 a) = transA a;
    transPN john = j;
    transPN mary = m;
    transA happy = h;
    transA smart = s;
}

concrete SemanticsRgl of Semantics = PeopleRgl ** {
  -- hack: if parsing as Prop, transS gets wrapped around
  -- parse tree. This is a work-around since the -transfer
  -- option in the put_tree command isn't supported anymore.
  lincat Prop = S; lin transS s = s;
}
```

A.4 GLF

For the GLF implementation, we can simply use the GF parsing code (Appendix A.3.1) and extend it with MMT code for the semantics construction. The implementation can be tested with the GLIF command

```
parse "John and Mary are happy and smart" | construct
```

which results in the expression

```
((happy' john')∧(smart' john'))∧((happy' mary')∧(smart' mary'))
```

Here is the MMT code:

```
theory proplog : ur:?LF =
  proposition : type | # o |
  not : o⟶o    | # ¬ 1 prec 80   |
  and : o⟶o⟶o | # 1 ∧ 2 prec 60 |
  or  : o⟶o⟶o | # 1 ∨ 2 prec 50 |
      = [a,b] ¬ (¬ a ∧ ¬ b) |

theory plnq : ur:?LF =
  include ?proplog |
  individuals : type | # ι |

theory DDT : ?plnq =
  john  : ι    | # john'  |
  mary  : ι    | # mary'  |
  smart : ι⟶o | # smart' |
  happy : ι⟶o | # happy' |

view rulesSem : jupyter/Rules.gf?Rules -> ?plnq =
  S   = o                          |
  NP  = (ι⟶o)⟶o                  |
  AP  = ι⟶o                       |
  A   = ι⟶o                       |
  PN  = ι                          |
  s1  = [a,b] a ∧ b               |
  s2  = [np,ap] np ap             |
  np1 = [a,b] [p] (a p) ∧ (b p) |
  np2 = [pn] [p] p pn             |
  ap1 = [a,b] [x] (a x) ∧ (b x) |
  ap2 = [a] a                      |

view lexiconSem : jupyter/Lexicon.gf?Lexicon -> ?DDT =
  include ?rulesSem |
  john  = john'  |
  mary  = mary'  |
  smart = smart' |
  happy = happy' |

view peopleSem : jupyter/People.gf?People -> ?DDT =
  include ?rulesSem |
  include ?lexiconSem |
```

APPENDIX B

ND and Tableau Rules in MMT

This appendix contains some of the MMT code used for the ELPI prover generation discussed in Chapter 4:

```
theory proplog : ur:?LF =
  proposition : type | # o |
  not : o⟶o | # ¬ 1 prec 80 |
  and : o⟶o⟶o | # 1 ∧ 2 prec 60 |

theory nd : ur:?LF =
  include ?proplog |
  ded : o ⟶ type | # ⊢ 1 prec 10 | role Judgment |
  falsum : type | # ↯ | role Judgment |

  notI : {A} (⊢A ⟶ ↯) ⟶ ⊢¬A | # noti 2 |
  notE : {A} ⊢¬¬A ⟶ ⊢A | # note 2 |

  falsumE : {A} ↯ ⟶ ⊢A | # falsume 1 2 |
  falsumI : {A} ⊢A ⟶ ⊢¬A ⟶ ↯ | # falsumi 2 3 |

  andI  : {A,B} ⊢A ⟶ ⊢B ⟶ ⊢A∧B | # andi 3 4 |
  andEl : {A,B} ⊢A∧B ⟶ ⊢A | # andel 3 |
  andEr : {A,B} ⊢A∧B ⟶ ⊢B | # ander 3 |

theory tab : ur:?LF =
  include ?proplog |
  marktrue     : o ⟶ type | # 1 ^T prec -5 | role TabMarker |
  markfalse    : o ⟶ type | # 1 ^F prec -5 | role TabMarker |
  closedbranch : type     | # ⊥           | role TabClosed |

  closebranch : {A} A^T ⟶ A^F ⟶ ⊥ | # close 2 3 |
  proofstart : {A} (A^F ⟶ ⊥) ⟶ A^T | # start 2 |

  notF : {A} ¬A^F ⟶ (A^T ⟶ ⊥) ⟶ ⊥ | # notf 2 3 |
  notT : {A} ¬A^T ⟶ (A^F ⟶ ⊥) ⟶ ⊥ | # nott 2 3 |

  andF : {A,B} A∧B^F ⟶ (A^F ⟶ ⊥) ⟶ (B^F ⟶ ⊥) ⟶ ⊥ | # andf 3 4 5 |
  andT : {A,B} A∧B^T ⟶ (A^T ⟶ B^T ⟶ ⊥) ⟶ ⊥ | # andt 3 4 |

view proplogID : ?proplog -> ?proplog =
  proposition = proposition |
  not = not |
  and = and |

view nd2tab : ?nd -> ?tab =
  include ?proplogID |
  ded = [p] p^T |
  falsum = closedbranch |

  notI = [A, ar] start ([naf] (notf naf ar)) |
  notE = [A, nna] start ([af] nott nna ([naf] notf naf ([a] close a af))) |

  falsumE = [A, bot] start ([af] bot) |
  falsumI = [A, a, na] nott na ([af] close a af) |

  andI = [A, B, a, b]
```

```
        start ([abf] andf abf ([af] close a af) ([bf] close b bf))
  andEl = [A, B, ab] start ([af] andt ab ([a,b] close a af))
  andEr = [A, B, ab] start ([bf] andt ab ([a,b] close b bf))

view tab2nd : ?tab -> ?nd =
  include ?proplogID
  marktrue     = [A] ⊢A
  markfalse    = [A] ⊢¬A
  closedbranch = ↯
  closebranch  = falsumI
  proofstart   = [A,nar] note (noti nar)

  notF = [A,nna,ar] ar (note nna)
  notT = [A, na, nar] (nar na)
  andF = [A, B, nab, nar, nbr]
      falsumi (andi (note (noti nar)) (note (noti nbr))) nab
  andT = [A, B, ab, abr] abr (andel ab) (ander ab)
```

APPENDIX C

Notation Guide

This thesis uses notational conventions from different fields and systems, which may be confusing. Here is a brief overview of important notations used throughout the thesis.

C.1 LF

GLIF is based on the type-theoretical compatibility of several systems that, unfortunately, don't use the same notations:

	General	GF	MMT	ELPI
Dependent types	$\prod_{x:A}$ or $\Pi_{x:A}$	`(x : A)`	`{x:A}`	
Arrow types	$A \to B$	`A -> B`	`A → B`	*`A -> B`*
λ-abstraction	$\lambda x : A.M$	`\x -> M`	`[x:A] M`	*`x \ M`*
Function application	$f\ x$ or $f(x)$	`f x`	`f x`	`f x`

Note that the type annotations for dependent types and λ-abstraction may be omitted if the type can be inferred or, in the case of λ-abstraction, if used in an untyped setting. As a short-hand notation, multiple values can be introduced by a binder (e.g. $\lambda x, y.M$ instead of $\lambda x.\lambda y.M$).

C.2 Logical Operators

$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence
$\forall x.P(x)$	universal quantification
$\forall \lambda x.P\ x$	universal quantification (higher-order abstract syntax)
$\exists x.P(x)$	existential quantification
$\exists \lambda x.P\ x$	existential quantification (higher-order abstract syntax)

Index

λProlog, 23

abstract syntax, 15
abstract syntax tree, 11
assumption (LF rule), 36
AST, 11
atomic judgment, 36

category of theories and theory morphisms, 18
code cell, 46
complete, 7
compositional, 9
conclusion (LF rule), 36
concrete syntax, 15
controlled natural language, 50

DDT, 30
declaration (MMT), 19
definite clause grammar, 13
discourse domain theory, 18, **30**
discourse representation theory, 67
DRT, 67

ELPI, 23

fake lambda, 53
formal language, 6
formal semantics, 6
ForTheL, 52
foundation independent, 19
fragment, 7

GF, 15
GLF, 29
GLForTheL, 52
GLIF, 34
GLIF command, 48
GLIF kernel, 46
grammar, 8
Grammatical Framework, 15
Grammatical Logical Framework, 29
Grammatical Logical Inference Framework, 34

Haskell/CSFP, 14
helper predicate, 24
higher-order abstract syntax, 20
hypothesis (LF rule), 37

intensional logic, 54
iterative deepening certificate, 25

judgments-as-types, 21
Jupyter, 46

language theory, 29
LATIN, 23
LF, 19
linearization, 15

markdown cell, 46
meta theory, 19
method of fragments, 7
MMT, 18
MMT surface syntax, 19
most certain principle, 4
multi-modal logic, 54

natural language, 4, 6
natural language understanding, 1
NLU, 1
NLU pipeline, 1, **7**

parameter (LF rule), 36
parse tree, 8
parser, 8
parser combinator, 14
product certificate, 26
Prolog/RINL, 12
proof certificate, 25
proof term certificate, 25

referent, 4
Resource Grammar Library, 17

$S5_n$, 54
SageMath, 51
salience, 58
semantic/pragmatic analysis, 1, **12**

semantics construction, 8
sense, 5
sound, 7
stub generation, 49
syntactic category, 8

tableaux machine, 57
textual entailment, 6
theory, 18
theory graph, 22
theory morphism, 18
thesis (LF rule), 37
truth condition, 5
type-raising, 10

urtheory, 19

view, 20

world knowledge, 7