Computational Semantics

Aljoscha Burchardt
Alexander Koller
Stephan Walter

ESSLLI 2004, Nancy

Abstract

The most central fact about natural language is that it has meaning. Semantics is the study of meaning. In formal semantics, we conduct this study in a formal manner. In computational semantics, we're additionally interested in using the results of our study when we implement programs that process natural language. In this course, we show how formal semantic representations can be computed and used in simple inference systems.

This material is part of a one semester course that has been developed within the project "MiLCA" ([3]), funded by the German Ministry of Education and Research. The material was generated using the OzDoc-Tool ([4]). As it was originally designed to be disseminated in HTML format, some formatting of this black and white print version is not optimal. The full course material including exercises and Prolog code is available at [2].

Parts of this reader are loosely based on an early version of the course on computational semantics by Patrick Blackburn and Johan Bos [1] and materials contributed by Michael Kohlhase. All errors are our own.

Computerlinguistik, Universität des Saarlandes
Saarbrücken, Germany
August 2004

{albu|koller|stwa}@coli.uni-sb.de

Contents

1

Semantic Construction

In this chapter, the central question we're going to look at is the following: 'Given a sentence, how do we get to its meaning?' And, being programmers, the next thing that interests us is: 'How can we automate this process?' This is, we're going to look at the task of *meaning* or *semantic construction* . But one of the first things we're going to see is that *syntactic structure* plays a crucial role in meaning construction.

1.1 Introduction

Meaning Representations

Before looking at the details of semantics construction, there's one big question that we have to answer: Meaning as such is a very abstract concept. It's not at all easy to imagine what it is to 'get to the meaning' of a sentence, let alone to 'construct' it from that sentence. To study meaning, and especially to deal with it on a computer, we need a handle on it, something more concrete: We shall work with *meaning representation* s - strings of a formal language that has technical advantages over natural language. In this chapter, we will represent meanings using formulas of *first-order logic*.

For instance, we will say that the meaning of the sentence 'Every man walks' is represented by the first order formula $\forall x(\textsc{man}(x) \rightarrow \textsc{walk}(x))$, and that the formula $\textsc{love}(\textsc{john}, \textsc{mary})$ represents the meaning of the natural language sentence 'John loves Mary'.

So basically, this chapter will be concerned with finding a systematic way of translating natural language sentences into formulas of first order logic (and writing a program that automates this task). Here's what we will do:

1. We will start with a very short repetition of some central concepts of first order logic.
2. Then, we will show how to represent first order formulas - thus, our target representations - in Prolog.
3. Next, we will discuss theoretically some of the basic problems that arise in semantic construction, introduce λ-calculus, and show why it is our tool of choice for solving these problems.
4. Finally we will turn to our implementation: We give Prolog code for the basic functionalities of λ-calculus, and then show how to couple λ-based semantic construction with our first, DCG-based, parsing-predicates.

1.2 First-Order Logic: Basic Concepts

In order to talk about meanings, we need a way for representing them. In this chapter, we're going to use the language of first-order logic for this purpose. So, when we say that we construct a meaning representation for some sentence, that means that we construct a formula of first-order logic that we take to represent this meaning.

You may say: 'What's the point in that? You're not giving the meaning of that sentence, you're just translating it to some artificial language that nobody uses.'

Is the situation really like that? No! Using first-order logic as a meaning-representation language has many advantages. Here are two of them:

- First of all, first order logic isn't just some artificial language that nobody uses. There is the *truth-functional interpretation* (see Chapter 5) telling us unambigously under which conditions formulas hold true and what the symbols they're made of mean. In other words, we have a formally precise conception of how our first order meaning representations relate to certain situations in the world. And we can compare this to our intuitions about the truth-conditions of the natural language sentences under consideration. That way we can judge the adequacy of our first-order formulas as meaning representations.
- Second, first order logic is a formal language with desirable properties such as having a simple, well defined (and unambigous) syntax. This makes it fit for use with computer programs.

We assume that you've already heard about first order logic. In the following we'll only shortly review its syntax and postpone the discussion of semantic concepts like truth or models for formulas to Section 5.1. In the rest of this chapter it's enough to have some intuition about what the right first order formula is for a given sentence. So as regards semantics, all we will do is sometimes give rough natural language equivalents to first-order formulas and their building blocks, to help you get this intuition.

1.2.1 Vocabularies

Intuitively, a vocabulary tells us the language the 'first-order conversation' is going to be conducted in. It tells us *in what terms* we will be able to talk about things. Here is our first vocabulary:

$$(\{\textsc{mary}, \textsc{john}, \textsc{anna}, \textsc{peter}\}, \{(\textsc{love}, 2), (\textsc{therapist}, 1), (\textsc{moron}, 1)\})$$

Generally, a vocabulary is a pair of sets:

1. The first set tells us what symbols we can use to name certain entities of special interest. In the case of the vocabulary we have just established, we are informed that we will be using four symbols for this purpose (we call them *constant symbol* s or simply *name* s), namely $\textsc{mary}$, $\textsc{john}$, $\textsc{anna}$, and $\textsc{peter}$.

2. The second set tells us with what symbols we can speak about certain properties and relations (we call these symbols *relation symbol* s or *predicate symbol* s). With our example vocabulary, we have one predicate symbol LOVE of arity 2 (that is, a 2-place predicate symbol) for talking about one two-place relation, and two predicate symbols of arity 1 (THERAPIST and MORON) for talking about (at most) two properties.

As such, the vocabulary we've just seen doesn't yet tell us a lot about the kinds of situations we can describe. We only know that some entities, at most two properties, and one two-place relation will play a special role in them. But since we're interested in natural language, we will use our symbols 'suggestively'. For instance, we will only use the symbol LOVE for talking about a (one-sided) relation called loving, and the two symbols THERAPIST and MORON will serve us exclusively for talking about therapists and morons. With this additional convention, the vocabulary really shows us what kind of situations the conversation is going to be about (formally, it gives us all the information needed to define the class of models of interest - but we said that we won't go into this topic here). Syntactically, it helps us define the relevant first-order language (that means the kinds of formulas we can use). So let's next have a look at how a first order language is generated from a vocabulary.

1.2.2 First-Order Languages

A *first-order language* defines how we can use a vocabulary to form complex, sentence-like entities. Starting from a vocabulary, we then build the first-order language over that vocabulary out of the following ingredients:

The Ingredients

1. All of the symbols in the vocabulary. We call these symbols the *non-logical* symbols of the language.
2. A countably infinite collection of variables x, y, z, w and so on.
3. The Boolean connectives $\neg$ (negation), $\rightarrow$ (implication), $\vee$ (disjunction), and $\wedge$ (conjunction).
4. The quantifiers $\forall$ (the universal quantifier) and $\exists$ (the existential quantifier).
5. The round brackets) and (. (These are essentially punctuation marks; they are used to group symbols.)

Items 2-5 are called logical symbols. They are common to all first-order languages. Hence the only aspect that distinguishes first-order languages from one another is the choice of non-logical symbols (that is, of vocabulary).

1.2.3 Building Formulas

Terms

Let's suppose we've composed a certain vocabulary. How do we mix these ingredients together? That is, what is the *syntax* of first-order languages? First of all, we define

a first-order *term* τ to be any constant or any variable. Roughly speaking, terms are the noun phrases of first-order languages: constants can be thought of as first-order counterparts of proper names, and variables as first-order counterparts of pronouns.

Atomic Formulas

We can then combine our 'noun phrases' with our 'predicates' (meaning, the various relation symbols in the vocabulary) to form what we call *atomic formula* s:

If R is a relation symbol of arity n , and $\tau_1, \ldots, \tau_n$ are terms, then $R(\tau_1, \ldots, \tau_n)$ is an atomic formula.

Intuitively, an atomic formula is the first-order counterpart of a natural language sentence consisting of a single clause (that is, a simple sentence). So what does a formula like $R(\tau_1, \cdots, \tau_n)$ actually mean? As a rough translation, we could say that the entities that are named by the terms $\tau_1, \cdots, \tau_n$ stand in a relationship that is named by the symbol R. An example will clarify this point:

$$\text{LOVE}(\text{PETER}, \text{ANNA})$$

What's meant by this formula is that the entity named PETER stands in the relation denoted by LOVE to the entity named ANNA - or more simply, that Peter loves Anna.

Complex Formulas

Now that we know how to build atomic formulas, we can define more complex ones as well. The following inductive definition tells us exactly what kinds of *well-formed formula* s (or *wffs*, or simply *formulas*) we can form.

1. All atomic formulas are wffs.
2. If φ and ψ are wffs then so are $\neg\varphi$, $(\varphi \rightarrow \psi)$, $(\varphi \vee \psi)$, and $(\varphi \wedge \psi)$.
3. If φ is a wff, and x is a variable, then both $\exists x\varphi$ and $\forall x\varphi$ are wffs. (We call φ the *matrix* or *scope* of such wffs.)
4. Nothing else is a wff.

Roughly speaking, formulas built using $\wedge$, $\rightarrow$, $\vee$ and $\neg$ correspond to the natural language expressions '... and ...', 'if ... then ...', '... or ...', and 'it is not the case that ...', respectively. First-order formulas of the form $\exists x\varphi$ and $\forall x\varphi$ correspond to natural language expressions of the form 'some...' or 'all...'.

1.2.4 Bound and Free Variables

Free and Bound Variables

Let's now turn to a rather important topic: the distinction between *free variable* s and *bound variable* s.

Have a look at the following formula:

$$\neg(\text{THERAPIST}(x) \vee \forall x(\text{MORON}(x) \wedge \forall y\text{PERSON}(y)))$$

The first occurrence of x is *free*, whereas the second and third occurrences of x are *bound*, namely by the first occurrence of the quantifier $\forall$. The first and second occurrences of the variable y are also bound, namely by the second occurrence of the quantifier $\forall$.

Informally, the concept of a *bound variable* can be explained as follows: Recall that quantifications are generally of the form:

$$\forall x \varphi$$

or

$$\exists x \varphi$$

where x may be any variable. Generally, all occurences of this variable within the quantification are bound. But we have to distinguish two cases. Look at the following formula to see why:

$$\exists x(\textsc{man}(x) \wedge (\forall x \textsc{walks}(x)) \wedge \textsc{happy}(x))$$

1. x may occur within another, embedded, quantification $\forall x\psi$ or $\exists x\psi$, such as the x in $\textsc{walks}(x)$ in our example. Then we say that it is bound by the quantifier of this embedded quantification (and so on, if there's another embedded quantification over x within ψ).
2. Otherwise, we say that it is bound by the top-level quantifier (like all other occurences of x in our example).

Here's a full formal simultaneous definition of *free* and *bound*:

1. Any occurrence of any variable is free in any atomic formula.
2. No occurrence of any variable is bound in any atomic formula.
3. If an occurrence of any variable is free in φ or in ψ, then that same occurrence is free in $\neg\varphi$, $(\varphi \rightarrow \psi)$, $(\varphi \vee \psi)$, and $(\varphi \wedge \psi)$.
4. If an occurrence of any variable is bound in φ or in ψ, then that same occurrence is bound in $\neg\varphi$, $(\varphi \rightarrow \psi)$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$. Moreover, that same occurrence is bound in $\forall y\varphi$ and $\exists y\varphi$ as well, for any choice of variable y.
5. In any formula of the form $\forall y\varphi$ or $\exists y\varphi$ (where y can be any variable at all in this case) the occurrence of y that immediately follows the initial quantifier symbol is bound.
6. If an occurrence of a variable x is free in φ, then that same occurrence is free in $\forall y\varphi$ and $\exists y\varphi$, for any variable y distinct from x. On the other hand, all occurrences of x that are free in φ, are bound in $\forall x\varphi$ and in $\exists x\varphi$.

If a formula contains no occurrences of free variables we call it a *sentence* .

1.2.5 Notation

In what follows, we won't always be adhering to the official first-order syntax defined above. Instead, we'll generally try and use as few brackets as possible, as this tends to improve readability. For example, we would rather not write

Outer Brackets

$$(\textsc{therapist}(\textsc{john}) \wedge \textsc{moron}(\textsc{peter}))$$

which is the official syntax. Instead, we are (almost invariably) going to drop the outermost brackets and write

$$\textsc{therapist}(\textsc{john}) \wedge \textsc{moron}(\textsc{peter})$$

Precedence

To help further reduce the bracket count, we assume the following precedence conventions for the Boolean connectives: $\neg$ takes precedence over $\vee$ and $\wedge$, both of which take precedence over $\rightarrow$. What this means, for example, is that the formula

$$\forall x((\neg\textsc{therapist}(x)) \wedge \textsc{moron}(x) \rightarrow \textsc{moron}(x))$$

is shorthand for the following:

$$\forall x((\neg\textsc{therapist}(x) \wedge \textsc{moron}(x)) \rightarrow \textsc{moron}(x))$$

In addition, we'll use the square brackets] and [as well as the official round brackets, as this can make the intended grouping of symbols easier to grasp visually.

1.2.6 Representing formulas in Prolog

We would like to use first order logic as a semantic representation formalism, and we want to deal with it in Prolog. So the next thing we need is a way of writing down formulas of first-order logic in Prolog. In short, we will simply use Prolog terms for this purpose that resemble the formulas they stand for as closely as possible. This is what we deal with in this section.

Atomic Formulas

First, we must decide how to represent *constant symbols*, *predicate symbols*, and *variables*. We do so in the easiest way possible: a first-order constant c will be represented by the Prolog atom `c`, and a first-order predicate symbol P will be represented by the Prolog atom `p`. Variables will also be represented by Prolog atoms. Note that this choice of representation won't allow our programs to distinguish constants from variables. So it's our own responsibility to choose the atoms for constants distinct from those for variables when we write down formulas in Prolog.

Given these conventions, it is obvious how *atomic formulas* should be represented. For example, $\textsc{love}(\textsc{john},\textsc{mary})$ would be represented by the Prolog term `love(john,mary)`, and $\textsc{hate}(\textsc{peter},x)$ would be represented by `hate(peter,x)`.

Complex Formulas

Next for Boolean combinations of simple formulas. The symbols

```
&    v    >    ~
```

will be used to represent the connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ respectively.

The following Prolog code ensures that these connectives have their usual precedences:

```
:- op(900,yfx,>).        % implication
:- op(850,yfx,v).        % disjunction

:- op(800,yfx,&).        % conjunction

:- op(750, fy,~).        % negation
```

Have a look at *Learn Prolog Now!*[1] if you are unsure about what this code means.

Here are some examples of complex first-order formulas represented in Prolog. To test your understanding of the above operator definitions: How would the formulas look in fully bracketed version?

- `love(john, mary) & love(mary, john) > hate(peter,john)`
- `love(john, mary) & ~ love(mary, john) > hate(john.peter)`
- `~ love(mary, john) v love(peter,mary) & love(john, mary) > hate(john.peter)`

Quantifiers

Finally, we must decide how to represent quantifiers. Recall that the first order formula $\textsc{man}(x)$ has the Prolog-representation `man(x)`. Now $\forall x.\textsc{man}(x)$ will be represented as

```
forall(x,man(x))
```

and $\exists x.\textsc{man}(x)$ will be represented as

```
exists(x,man(x))
```

1.3 Building Meaning Representations

Now that we've learned something about first-order logic and how to work with it in Prolog, it is time to have a look at the major issue of this chapter, which is:

Given a sentence of English, how do we get to its meaning representation?

This question is of course far too general for what we can achieve in one chapter of this course. So let's rather ask a more specific one: 'Is there a systematic way of translating such simple sentences as 'John loves Mary' and 'A woman walks' into first-order logic?' The important point here is the demand of being systematic. Next, we will discuss why this is so important.

[1] `http://www.coli.uni-saarland.de/~kris/learn-prolog-now/html/node82.html#sec.l9.operators`

1.3.1 Being Systematic

Is there a *systematic* way of translating such simple sentences as 'John loves Mary' and 'A woman walks' into first-order logic?

The key to answering this question is to be more precise about what we mean by 'systematic'. When examining the sentence 'John loves Mary', we see that its semantic content is (at least partially) captured by the first-order formula LOVE(JOHN,MARY). Now this formula consists of the symbols LOVE, JOHN and MARY. Thus, the most basic observation we can make about systematicity is the following: the proper name 'John' contributes the constant symbol JOHN to the representation, the transitive verb 'loves' contributes the relation symbol LOVE, and the proper name 'Mary' contributes the constant symbol MARY.

More generally, it's the words of which a sentence consists that contribute the relation symbols and constants in its semantic representation. But (important as it may be) this observation doesn't tell us everything we need to know about systematicity. It only tells us where the *building blocks* of our meaning representations will come from - namely from words in the lexicon.

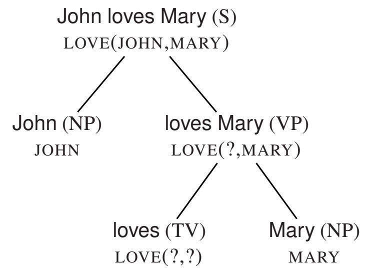
But it doesn't tell us how to *combine* these building blocks. For example we have to form the first-order formula LOVE(JOHN,MARY) from the symbols LOVE, JOHN and MARY. But from the same symbols we can also form LOVE(MARY,JOHN). So why do we choose to put MARY in the second argument slot of LOVE rather than in the first one? Is there a principle behind this decision? For this task, we haven't been specific yet about what we mean by working *in a systematic fashion*.

1.3.2 Being Systematic (II)

Syntactic Structure...

Our missing link here is the notion of *syntactic structure* . As we know well from the previous chapters, 'John loves Mary' isn't just a string of words: it has a hierarchical structure. In particular, 'John loves Mary' is an S (sentence) that is composed of the subject NP (noun phrase) 'John' and the VP (verb phrase) 'loves Mary'. This VP is in turn composed of the TV (transitive verb) 'loves' and the direct object NP 'Mary'. Given this hierarchy, it is easy to tell a conclusive story about - and indeed, to draw a convincing picture of - why we should get the representation LOVE(JOHN,MARY) as a result, and nothing else:

See movie in HTML version.



...and its use for Semantics

When we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP (in this case, MARY) in the *second* argument slot of the VP's semantic representation (in this case, LOVE(?,?)). Next, JOHN needs to be inserted into the *first* argument slot. Why? Simply because this is the slot reserved for the semantic representations of NPs that we combine with VPs to form an S.

In more general terms, given that we have some reasonable syntactic story about what the pieces of our sentences are, and which pieces combine with which other pieces, we can try to use this information to explain how the various semantic contributions have to be combined.

Summing up we are now in a position to give quite a good explication of 'systematicity': When we construct meaning representations systematically, we integrate information from *two different* sources:

1. The lexical items (i.e. the words) in a sentence give us the basic ingredients for our representation.
2. Syntactic structure tells us how the semantic contributions of the parts of a sentence are to be joined together.

1.3.3 Three Tasks

Let us have a look at the general picture that's emerging. How do we translate simple sentences such as 'John loves Mary' and 'A woman walks' into first-order logic? Although we still don't have a specific method at hand, we can formulate a plausible strategy for finding one. We need to fulfill three tasks:

Task 1 Specify a reasonable syntax for the natural language fragment of interest.

Task 2 Specify semantic representations for the lexical items.

Task 3 Specify the translation of complex expressions (i.e. phrases and sentences) *compositionally*. That is, we need to specify the translation of such expressions in terms of the translation of their parts, *parts* here referring to the substructure given to us by the syntax.

Of course all three tasks should be carried out in a way that naturally leads to computational implementation. Because this chapter is on *semantic* construction, tasks 2 and 3 are where our real interests lie, and most of our attention will be devoted to them. But we also need a way of handling task 1.

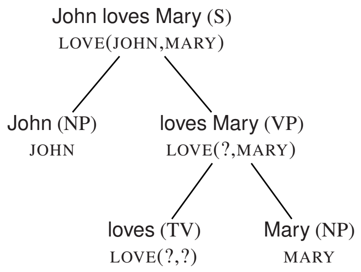
1.3.4 From Syntax to Semantics

Task 1 ✓

In order to approach Task 1, we will use a simple context free grammar. As ususal, the syntactic analysis of a sentence will be represented as a tree whose non-leaf nodes

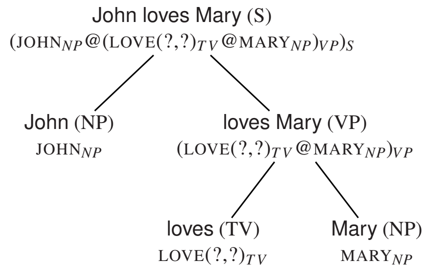
represent *complex syntactic categories* (such as S, NP and VP) and whose leaves represent *lexical items* (these are associated with *lexical categories* such as noun, transitive verb, determiner, proper name and intransitive verb). To enhance the readability of such trees, we will omit the non-branching steps and take for instance Mary (NP) as a leave node.

Let's have a second look at our semantically annotated syntax-tree for the sentence 'John loves Mary' (from Section 1.3.2).



We said that the systematic contribution of syntactic structure to semantic construction consists in guiding the semantic contributions of words and phrases to the right places in the final semantic representation. So far so good, but in constructing the formula LOVE(JOHN,MARY) along the above syntax tree, we made tacit use of a lot of knowledge about *how exactly* syntactic information should be used. Can we make this knowledge more explicit?

Let's take a step back. What's the simplest way of taking over syntactic information into our semantic representation? Surely, the following is a very undemanding first step:



We've simply taken the semantic contributions of words and phrases, uniformly joined them with an @-symbol, and encoded the tree structure in the bracketing structure of the result. Yet the result is still quite far from what we actually want to have. It definitely isn't a first order formula. In fact we've only postponed the question of how to exploit the syntactic structure for guiding arguments to their places. What we've got is a nice linear 'proto'-semantic representation, in which we still have all syntactic information at hand. But this representation still needs a lot of post-processing.

What we could now try to do is start giving post-procesing rules for our 'proto'-semantic representation, rules like the following:'If you find a transitive verb representation between two @-symbols, always take the item to its left as first argument, and the item to its right as second argument.'

Formulating such rules would soon become very complicated, and surely our use of terms like 'item to the left' indicates that we've not yet reached the right level of abstraction in our formulation. In the next section, we're going to look at λ-calculus, a formalism that gives us full flexibility in speaking about missing pieces in formulas, where they're missing, and when and from where they should be supplied. It provides the right level of generality for capturing the systematics behind the influence that syntactic structure has on meaning construction. Post-processing rules like the one just seen won't be necessary, their role is taken over by the uniform and very simple operation of β-reduction.

1.4 The Lambda Calculus

Towards the end of the last section we saw how to transfer as much information as possible about the syntactic structure of a sentence into a kind of proto-semantic representation. But we still completely lack a uniform way of combining the collected semantic material into well-formed first order formulas.

In this section we will dicuss a mechanism that fits perfectly for this task. It will allow us to explicitly mark gaps in first-order formulas and give them names. This way we can state precisely how to build a complete first order formula out of separate parts. The mechanism we're talking about is called λ-calculus. For present purposes we shall view it as a notational extension of first order logic that allows us to bind variables using a new variable binding operator λ. Here is a simple λ-expression:

$$\lambda x.\textsc{woman}(x)$$

The prefix $\lambda x.$ binds the occurrence of x in $\textsc{woman}(x)$. That way it gives us a handle on this variable, which we can use to state how and when other symbols should be inserted for it.

1.4.1 Lambda-Abstraction

λ-expressions are formed out of ordinary first order formulas using the λ-operator. We can prefix the λ-operator, followed by a variable, to any first order formula or λ-expression. We call expressions with such prefixes *λ-abstraction* s (or, more simply, *abstractions*). We say that the variable following a λ-operator is *abstracted over* . And we say that the variable abstracted over is *(λ-)bound* by its respective λ-operator within an abstraction, just as we say that a quantified variable is bound by its quantifier inside a quantification.

Abstractions

The following two are examples of λ-abstractions:

1. $\lambda x.\textsc{woman}(x)$

2. $\lambda u.\lambda v.\textsc{love}(u,v)$

In the first example, we have abstracted over x. Thus the x in the argument slot of WOMAN is bound by the λ in the prefix. In the second example, we have abstracted twice: Once over v and once over u. So the u in the first argument slot of LOVE is bound by the first λ, and the v is bound by the second one.

Missing Information

We will think of occurrences of variables bound by λ as placeholders for missing information: They serve us to mark *explicitly* where we should substitute the various bits and pieces obtained in the course of semantic construction. Let us look at our first example λ-expression again. Here the prefix $\lambda x.$ states that there is information missing in the formula following it (a one-place predication), and it gives this 'information gap' the name x. The same way in our second example, the two prefixes $\lambda u.$ and $\lambda v.$ give us separate handles on *each of the two* information gaps in the following two-place predication.

1.4.2 Reducing Complex Expressions

So the use of λ-bound variables allows us to mark places where information is missing in a partial first order formula. But how do we fill in the missing information when it becomes available? The answer is simple: We *substitute* it for the λ-bound variable. We can read a λ-prefix as a request to perform substitution for its bound variable.

Controlled substitution

In $\lambda x.\textsc{woman}(x)$, the binding of the free variable x in WOMAN(x) explicitly indicates that WOMAN has an argument slot where we may perform substitutions.

We will use concatenation (marked by an @-symbol) to indicate when we have to perform substitutions, and what to substitute. By concatentating a λ-expression with another expression, we trigger the substitution of the latter for the λ-bound variable. Consider the following expression (we use the special symbol @ to indicate concatenation):

$$\lambda x.\textsc{woman}(x)@\textsc{mary}$$

Functional Application, β-Reduction

This compound expression consists of the abstraction $\lambda x.\textsc{woman}(x)$ written immediately to the left of the expression MARY, both joined together by @. Such a concatenation is called *functional application* ; the left-hand expression is called the *functor*, and the right-hand expression the *argument*. The concatenation is an instruction to discard the $\lambda x.$ prefix of the functor, and to replace every occurrence of x that was bound by this prefix with the argument. We call this substitution process β-*reduction* (other common names include β-conversion and λ-conversion). Performing the β-reduction demanded in the previous example yields:

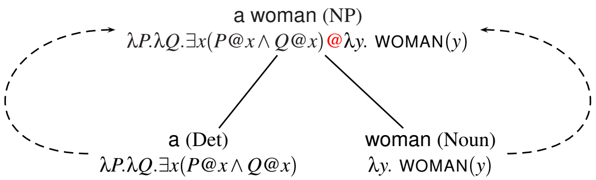
$$\textsc{woman}(\textsc{mary})$$

The purpose of λ-bound variables is thus to mark the slots where we want substitutions to be made, the purpose of λ-prefixes is to indicate at what point in the reduction process substitutions should be made, and the arguments of applications provide the material to be substituted. Abstraction, functional application, and β-reduction together will drive our first really systematic semantic construction mechanism. Next, let's see how it works in practice.

1.4.3 Using Lambdas

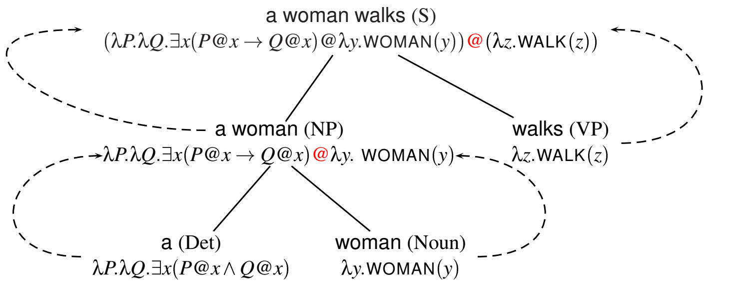
Let's return to the sentence 'A woman walks'. According to our grammar, a determiner and a common noun can combine to form a noun phrase. Our semantic analysis couldn't be simpler: we will associate the NP node with the functional application that has the determiner representation as functor and the noun representation as argument. Structurally, this is of course the same thing that we did in Section 1.3.4. Only this time the semantic contributions of constituents are generally λ-expressions, and we will simply read the @-symbols as application markers. In fact it will turn out that the combination of functional application and β-reduction is a method of such generality that we can even completely disregard the phrase-indices (such as NP and VP) that were part of our 'proto'-representations in Section 1.3.4.

Building a structured application...



As you can see from the picture, we use the λ-expression $\lambda P.\lambda Q.(\exists x(P@x \wedge Q@x))$ as our representation for the indefinite determiner 'a'. We'll take a closer look at this representation soon, after we've looked at how it does its job in the semantic construction process. But there's one thing that we have to remark already now. While the λ-bound variables in the examples we've seen so far were placeholders for missing *constant* symbols, P and Q in our determiner-representation stand for missing *predicates*. The version of λ-calculus introduced here does not distinguish variables that stand for different kinds of missing information. Nevertheless we will stick to a convention of using lower case letters for variables that stand for missing constant symbols, and capital letters otherwise.

But now let's carry on with the analysis of the sentence 'A woman walks'. We have to incorporate the intransitive verb 'walks'. We assign it the representation $\lambda.z\textsc{walk}(z)$. The following tree shows the final representation we obtain for the complete sentence:



The S node is associated with $(\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)@\lambda y.\textsc{woman}(y))@(\lambda z.\textsc{walk}(z))$. We obtain this representation by a procedure analogous to that performed at the NP node. We associate the S node with the application that has the NP representation just obtained as functor, and the VP representation as argument.

...and reducing it.

Now instead of hand-tailoring lots of specially dedicated post-processing rules, we will simply β-reduce the expression that we find at the S node as often as possible. We must follow its (bracketed) structure when we perform β-reduction. So we start with reducing the application $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)@\lambda y.\textsc{woman}(y)$. We have to replace P by $\lambda y.\textsc{woman}(y)$, and drop the λP prefix. The whole representation then looks as follows:

$$\lambda Q.\exists x(\lambda y.\textsc{woman}(y)@x \wedge Q@x)@\lambda z.\textsc{walk}(z)$$

See movie in HTML version.

Let's go on. This time we have two applications that can be reduced. We decide to get rid of the λQ first. Replacing Q by $\lambda z.\textsc{walk}(z)$ we get:

$$\exists x(\lambda y.\textsc{woman}(y)@x \wedge \lambda z.\textsc{walk}(z)@x)$$

Again we have the choice where to go on β-reducing – this time it should be obvious that our choice doesn't make any difference for the final result (in fact it never does. This property of λ-calculus is called *confluence*). Thus let's β-reduce twice. We have to replace both y and z by x. Doing so finally gives us the desired:

$$\exists x(\textsc{woman}(x) \wedge \textsc{walks}(x))$$

Determiner

Finally, let's have a closer look at the determiner-representation we've been using. Remember it was $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$. Why did we choose this expression? In a way, there isn't really an answer to this question, except simply: *Because it works.*

But then let's at least have a closer look at why it works. We know that a determiner must contribute a quantifier *and* the pattern of the quantification. Intuitively, indefinite determiners in natural language are used to indicate that there is something of a certain kind (epressed in the so-called *restriction* of the determiner), about which one is going to say that it also has some other property (expressed in the so-called *scope* of the determiner). In the sentence 'A woman walks', the 'a' indicates that there is something of a certain kind, namely 'woman', *and* that this something also has a certain property, namely 'walk'.

So for the case of an indefinite determiner, we know that the quantifier in its first-order formalization has to be existential, and that the main connective within the quantification is a conjunction symbol. This is the principle behind formalizing indefinite determiners in first-order logic.

Now clever use of λ-bound variables in our determiner representation allows us to leave unspecified all but just these two aspects. All that is already 'filled in' in the representation $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$ is the quantifier and a little bit about the internal structure of its scope, namely that the main connective is $\wedge$. The rest is 'left blank', and this is indicated using variables.

The second important thing about a λ-expression is the order of its prefixes. This is where the role of syntactic structure comes in - and where explanation really doesn't go any further in the case of our determiner: We had to choose $\lambda P.\lambda Q$ and not $\lambda Q.\lambda P$ for the simple reason that phrases and sentences containing determiners are built up syntactically as they are. This holds with all generality: When deciding about the order of λ-prefixes of a meaning representation, one has to think of the right generalizations over the syntactic use of its natural language counterpart.

1.4.4 Advanced Topics: Proper Names and Transitive Verbs

As we've just learnt, the way we use λs in our meaning representations reflects generalizations over the syntactic use of their natural language counterparts. Let's look at the representation of *proper names* and *transitive verbs* as further examples of this connection. We said before that the first-order counterparts of proper names are constant symbols, and that for example JOHN stands for 'John'. But while the semantic representation of a quantifying NP such as 'a woman' can be used as a functor, surely such a constant symbol will have to be used as an argument. Will this be a problem for our semantic construction mechanism?

Proper names

In fact, there's no problem at all - if we only look at things the right way. We *want* to use proper names as functors, because syntactic structure suggests to treat them the same way as quantified noun phrases. But then we just shouldn't translate them as constant symbols *directly*. Let's keep their intended use in mind when we design the semantic representations for proper names. It's all a matter of abstracting cleverly. Indeed the λ-calculus offers a delightfully simple functorial representation for proper names, as the following examples show:

'Mary': $\lambda P.P@\textsc{mary}$

'John': $\lambda Q.Q@\textsc{john}$

Role-Reversing

From outside (i.e. if we only look at the λ-prefix) these representations are exactly like the ones for quantified noun phrases. And - most importantly - they can be used in the same way: They are abstractions, thus they can be used as functors. However looking at the inside, note what such functors do. As always, they are essentially instructions to substitute their argument for the bound variable (i.e. P or Q). But this time, this means that the argument becomes itself applied, namely to the constant symbol that stnds for the bearer of the name! Because the λ-calculus offers us the means to specify such role-reversing functors, proper names can be used as functors just like quantified NPs.

Transitive verbs

As an example of these new representations in action, let us build a representation for 'John loves Mary'. But before we can do so, we have to meet another challenge: 'loves' is a transitive verb, it takes an object and forms a VP; we will want to apply it to its object-NP. And the resulting VP should be usable just like a standard intransitive verb; we want to be able to apply the subject NP to it. This is what we know in advance.

Given these requirements, a λ-expression like the simple $\lambda u.\lambda v.\textsc{love}(u,v)$ (which we've seen in Section 1.4.1) surely won't do. After all, the object NP combining with a transitive verb is itself a functor. It would be inserted for u in this λ-expression, but u isn't applied to anything anywhere. So the result could never be β-reduced to a well-formed first-order formula. How do we make our representation fit our needs this time? Let's try something like our role-reversing trick again; we'll assign 'loves' the following λ-expression:

$\lambda R.\lambda z.(R@\lambda x.\textsc{love}(z,x))$

An example

Thus prepared we're now ready to have a look at the semantic construction for 'John loves Mary'. We can build the following tree:

See movie in HTML version.

John loves Mary (S)
$(\lambda P.P@\textsc{john})@(\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))@(\lambda P.P@\textsc{mary}))$

John (NP)
$\lambda P.P@\textsc{john}$

loves Mary (VP)
$\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))@(\lambda P.P@\textsc{mary})$

loves (TV)
$\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))$

Mary (NP)
$\lambda P.P@\textsc{mary}$

How is this going to work? Let's look at the application at the S-node, and think through step by step what happens when we β-convert (page 12) it: Inside our complex application, the representation for the object NP is substituted for X. It ends up being applied to something looking like an intransitive verb ($\lambda x.\text{LOVE}(z,x)$), namely an 'intransitive verb with a free variable'. This application is going to be no problem - it's structurally the same we would get if our object NP was the subject of a true intransitive verb. So everything is fine here.

Now the remaining prefix λz makes the complete VP-representation also function like that of an intransitive verb (from outside). And indeed the subject NP semantic representation finally takes the VP semantic representation as argument, again as if it was the representation of a true intransitive verb. So everything is fine here, too.

Trace the semantic construction!

Make sure you understand what is going on here by β-reducing the expression at the S-node yourself!

1.4.5 The Moral

Our examples have shown that λ-calculus is ideal for semantic construction in two respects:

1. The process of combining two representations was perfectly uniform. We simply said which of the representations is the functor and which the argument, whereupon combination could be carried out by applying functor to argument and β-converting. We didn't have to make any complicated considerations here.
2. The load of semantic analysis was carried by the lexicon: We used the λ-calculus to make missing information stipulations when we gave the meanings of the *words* in our sentences. For this task, we had to think accurately. But we could make our stipulations declaratively, without hacking them into the combination process.

Our observations are indeed perfectly general. Doing semantic construction with the help of λ-calculus, most of the work is done before the actual combination process.

What we have to do...

When giving a λ-abstraction for a lexical item, we have to make two kinds of decisions:

1. We have to locate gaps to be abstracted over in the partial formula for our lexical item. In other words, we have to decide *where to put* the λ-bound variables inside our abstraction. For example when giving the representation $\lambda P.P@\text{MARY}$ for the proper name 'Mary' we decided to stipulate a missing functor. Thus we applied a λ-abstracted variable to MARY.
2. We have to decide *how to arrange* the λ-prefixes. This is how we control in which order the arguments have to be supplied so that they end up in the right places after β-reduction when our abstraction is applied. For example we chose the order $\lambda P.\lambda Q$ when we gave the representation $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$ for the indefinite determiner 'a'. This means that we will first have to supply it with the argument for the restriction of the determiner, and then with the one for the scope.

...and how

Of course we are not totally free in these decisions. What constrains us is that we want to be able to combine the representations for the words in a sentence so that they can be *fully* β*-reduced* to a well-formed first order formula. And not just some formula, but the one that captures the meaning of the sentence.

So when we design a λ-abstraction for a lexical item, we have to anticipate its potential use in semantic construction. We have to keep in mind *which final semantic representations* we want to build for sentences containing our lexical item, and *how* we want to build them. In order to decide what to abstract over, we must think about *which pieces* of semantic material will possibly be supplied from elsewhere during semantic construction. And in order to arrange our λ-prefixes, we must think about *when and from where* they will be supplied.

Summing up

The bottom line of all this is that devising lexical representations will be the tricky part when we give the semantics for a fragment of natural language using λ-calculus. But with some clever thinking, we can solve a lot of seemingly profound problems in a very streamlined manner.

1.4.6 What's next

What's next?

For the remainder of this lecture, the following version of the three tasks listed earlier (page 9) will be put into practise:

Task 1 Specify a DCG for the fragment of natural language of interest.

Task 2 Specify semantic representations for the lexical items with the help of the λ-calculus.

Task 3 Specify the translation $\mathcal{R}'$ of a syntactic item $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ with the help of functional application. That is, specify which of the subparts is to be thought of as functor (here it's $\mathcal{F}$), which as argument (here it's $\mathcal{A}$) and then define $\mathcal{R}'$ to be $\mathcal{F}'@\mathcal{A}'$, where $\mathcal{F}'$ is the translation of $\mathcal{F}$ and $\mathcal{A}'$ is the translation of $\mathcal{A}$. Finally, apply β-conversion as a post-processing step.

1.4.7 [Sidetrack:] Accidental Bindings

But before we can put λ-calculus to use in an implementation, we still have to deal with one rather technical point: Sometimes we have to pay a little bit of attention which variable names we use. Suppose that the expression $\mathcal{F}$ in $\lambda V.\mathcal{F}$ is a complex expression containing many λ operators. Now, it could happen that when we apply a functor $\lambda V.\mathcal{F}$ to an argument $\mathcal{A}$, some occurrence of a variable that is is free in $\mathcal{A}$ becomes bound when we substitute it into $\mathcal{F}$.

For example when we construct the semantic representation for the verb phrase 'loves a woman', syntactic analysis of the phrase could lead to the representation:

$$\lambda P.\lambda y.(P@\lambda x.\text{LOVE}(y,x))@(\lambda Q.\lambda R.(\exists y(Q@(y) \wedge R@y))@\lambda w.\text{WOMAN}(w))$$

β-reducing three times yields:

$$\lambda y.(\lambda R.(\exists y(\textsc{woman}(y) \wedge R@y))@\lambda x.\textsc{love}(y,x))$$

Notice that the variable y occurs λ-bound as well as existentially bound in this expression. In $\textsc{love}(y,x)$ it is bound by λy, while in $\textsc{woman}(y)$ and R it is bound by $\exists y$. So far, this has not become a problem. But look what happens when we β-convert once more:

$$\lambda y.(\exists y(\textsc{woman}(y) \wedge \lambda x.\textsc{love}(y,x)@y))$$

$\textsc{love}(y,x)$ has been moved inside the scope of $\exists y$. In result, the occurrence of y has been 'caught' by the existential quantifier, and λy doesn't bind any occurence of a variable at all any more. Now we β-convert one last time and get:

$$\lambda y.(\exists y(\textsc{woman}(y) \wedge \textsc{love}(y,y)))$$

We've got an empty λ-abstraction, made out of a formula that means something like 'A woman loves herself'. *That's not what we want to have*. Such accidental bindings (as they are usually called) defeat the purpose of working with the λ-calculus. The whole point of developing the λ- calculus was to gain control over the process of performing substitutions. We don't want to lose control by foolishly allowing unintended interactions.

1.4.8 [Sidetrack:] Alpha-Conversion

But such interactions need never happen. Obviously, our problem occured simply because we used *two* variables named y in our representation. But λ-bound variables are merely placeholders for substitution slots. The exact names of these placeholders do not play a role for their function. So, relabeling bound variables yields λ-expressions which lead to exactly the same substitutions in terms of 'slots in the formulas' (much like relabeling bound variables in quantified formulas doesn't change their truth values).

Let us look at an example. The λ-expressions $\lambda x.\textsc{man}(x)$, $\lambda y.\textsc{man}(y)$, and $\lambda z.\textsc{man}(z)$ are equivalent, as are the expressions $\lambda Q.\exists x(\textsc{woman}(x) \wedge Q@x)$ and $\lambda Y.\exists x(\textsc{woman}(x) \wedge Y@x)$. All these expressions are functors which when applied to an argument, replace the bound variable by the argument. No matter which argument $\mathcal{A}$ we choose, the result of applying any of the first three expressions to $\mathcal{A}$ and then β-converting is $\textsc{man}(\mathcal{A})$, and the result of applying either of the last two expressions to $\mathcal{A}$ is $\exists x(\textsc{woman}(x) \wedge \mathcal{A}@x)$.

α-Equivalence

Two λ-expressions are called α-*equivalent* if they only differ in the names of λ-bound variables. In what follows we often treat α-equivalent expressions as if they were identical. For example, we will sometimes say that the lexical entry for some word is a λ-expression $\mathcal{E}$, but when we actually work out some semantic construction, we might use an α-equivalent expression $\mathcal{E}'$ instead of $\mathcal{E}$ itself.

α-Conversion

The process of relabeling bound variables is called α-*conversion* . Since the result of α-converting an expression performs the same task as the initial expression, α-conversion is always permissible during semantic construction. But the reader needs to understand that it's not merely *permissible* to α-convert, it can be *vital* to do so if β-conversion is to work as intended.

Returning to our intial problem, if we can't use $\lambda V.\mathcal{F}$ as a functor, any α-equivalent formula will do instead. By suitably relabeling the bound variables in $\lambda V.\mathcal{F}$ we can always obtain an α-equivalent functor that doesn't bind any of the variables that occur free in $\mathcal{A}$, and accidental binding is prevented.

So, strictly speaking, it is not merely functional application coupled with β-conversion that drives our approach to semantic construction, but functional application and β-conversion coupled with (often tacit) use of α-conversion. Notice we only didn't encounter the problem of accidental binding earlier because we (tacitly) chose the names for the variables in the lexical representations cleverly. This means that we have been working with α-equivalent variants of lexical entries all along in our examples.

1.5 Implementing Lambda Calculus

Our decision to perform semantic construction with the aid of an abstract "glue" language (namely, λ-calculus) has pleasant consequences for grammar writing, so we would like to make the key combinatorial mechanisms (functional application and β-conversion), available as black boxes to the grammar writer. From a grammar engineering perspective, this is a sensible thing to do: when writing fragments we should be free to concentrate on linguistic issues.

In this section we build the required black boxes. With such a black boxes available, we will be able to use a small DCG for semantic construction. We will decorate it with extremely natural semantic construction code and start building representations.

1.5.1 Representations

First, we have to decide how to represent λ-expressions in Prolog. As in the case of representing first-order formulas, we will use Prolog terms that resemble the expressions they represent as closely as possible. For abstractions, something as simple as the following will do:

```
lambda(x,F)
```

Secondly, we have to decide how to represent application. Let's simply transplant our @-notation to Prolog by defining @ as an infix operator:

```
:- op(950,yfx,@).         % application
```

That is, we shall introduce a new Prolog operator `@` to explicitly mark where functional application is to take place: the notation `F@A` will mean 'apply function `F` to argument `A`'. We will build up our representations using these explicit markings, and then carry out β-conversion when all the required information is to hand.

1.5.2 Extending the DCG

Let's see how to use this notation in DCGs. We'll use a small DCG with e.g. an intransitive verb and a proper name as well as the necessary rules to use them in sentences. To use the resulting DCG for semantic construction, we have to specify the semantic representation for each phrasal and lexical item. We do this by giving additional arguments to the phrase markers of the DCG.

The resulting grammar is found in See file `semanticDCG.pl`.. Let's have a look at the phrasal rules first:

```
s(NP@VP) --> np(NP),vp(VP).

np(DET@N) --> det(DET),n(N).
np(PN) --> pn(PN).

vp(TV@NP) --> tv(TV),np(NP).
vp(IV) --> iv(IV).
```

The unary phrasal rules just percolate up their semantic representation (here coded as Prolog variables `NP`, `VP` and so on), while the binary phrasal rules use `@` to build a semantic representation out of their component representations. This is completely transparent: we simply apply function to argument to get the desired result.

1.5.3 The Lexicon

The real work is done at the lexical level. Nevertheless, the lexical entries for nouns and intransitive verbs practically write themselves:

```
n(lambda(X, witch(X))) --> [witch], {vars2atoms(X)}.
n(lambda(X, wizard(X))) --> [wizard], {vars2atoms(X)}.
n(lambda(X, broomstick(X))) --> [broomstick], {vars2atoms(X)}.
n(lambda(X, man(X))) --> [man], {vars2atoms(X)}.
n(lambda(X, woman(X))) --> [woman], {vars2atoms(X)}.

iv(lambda(X, fly(X))) --> [flies], {vars2atoms(X)}.
```

If you do not remember the somewhat difficult representation of transitive verbs, look at Section 1.4.4 again. Here's the lexical rule for our only transitive verb form, 'curses':

```
tv(lambda(X, lambda(Y, X@lambda(Z, curse(Y,Z))))) --> [curses], {vars2atoms
tv(lambda(X, lambda(Y, X@lambda(Z, love(Y,Z))))) --> [loves], {vars2atoms(X
```

Recall that the λ-expressions for the determiners 'every' and 'a' are $\lambda P.\lambda Q.\forall x.(P@x \rightarrow Q@x)$ and $\lambda P.\lambda Q.\exists x.(P@x \wedge Q@x)$. We express these in Prolog as follows:

```
det(lambda(P, lambda(Q, exists(X, ((P@X) & (Q@X)))))) --> [a], {vars2atoms(
det(lambda(P, lambda(Q, forall(X, ((P@X) > (Q@X)))))) --> [every], {vars2at
```

Finally, the 'role-reversing' (Section 1.4.4) representation for our only proper name:

```
pn(lambda(P, P@harry)) --> [harry], {vars2atoms(P)}.
pn(lambda(P, P@john)) --> [john], {vars2atoms(P)}.
pn(lambda(P, P@mary)) --> [mary], {vars2atoms(P)}.
```


Prolog Variables?

Note that we break our convention (page 6) of representing variables by constants in these lexical rules. All the λ-bound variables are written as Prolog variables instead of atoms. This is the reason why we have to add the calls to `vars2atoms/1` in some of our phrasal rules (included in curly brackets - curly brackets allow us to include further Prolog calls with DCG-rules). Whenever a lexical entry is retrieved, `vars2atoms/1` replaces all Prolog variables in it by new atoms. Distinct variables are replaced by distinct atoms. We won't go into how exactly this happens - if you're interested, have a look at the code of the predicate. After this call, the retrieved lexical entry is in accord with our representational conventions again.

This sounds complicated - so why do we do it? If you have read the sidetracks in the previous section (Section 1.4.7 and Section 1.4.8), you've heard about the possibility of accidental binding and the need for α-conversion during the semantic construction process. Now by using Prolog variables in lexical entries and replacing them by atoms on retrieval, we make sure that no two meaning representations taken from the lexicon ever contain the same λ-bound variables. In addition, the atoms substituted by `vars2atoms/1` are distinct from the ones that we use for quantified variables. Finally, no other rules in our grammar ever introduce any variables or double any semantic material. In result accidental bindings just cannot happen. So using Prolog variables in the lexicon may be a bit of a hack, but that way we get away without implementing α-conversion.

1.5.4 A First Run

Semantic construction during parsing is now extremely easy. Here is an example query:

?- `s(Sem,[harry,flies],[]).`

```
Sem = Sem=lambda(v1, v1@harry)@lambda(v2, fly(v2))
```

Or generate the semantics for 'Harry curses a witch.': `s(Sem,[harry,curses,a,witch],[]).`

The variables `v1,v2` etc. in the output come from the calls to `vars2atoms` during lexical retrieval. The predicate generates variable names by concatenating the letter `v` to a new number each time it is called.

So now we can construct λ-terms for natural language sentences. But of course we need to do more work *after* parsing, for we certainly want to reduce these complicated λ-expressions into readable first-order formulas by carrying out β-conversion. For this purpose we will now implement the predicate `betaConvert/2`.

1.5.5 Beta-Conversion

The first argument of `betaConvert/2` is the expression to be reduced and the second argument will be the result after reduction. Let's look at the two clauses of the predicate in detail. You find them in the file See file `betaConversion.pl`..

```
betaConvert(Functor@Arg,Result):-
       betaConvert(Functor,lambda(X,Formula)),
```

```
        !,
        substitute(Arg,X,Formula,BetaConverted),
        betaConvert(BetaConverted,Result).
```

The first clause of betaConvert/2 is for the cases where 'real' β-conversion is done, i.e. where a λ is thrown away and all occurences of the respective variable are replaced by the given argument. In such cases

1. The input expression must be of the form Functor@Arg,
2. The functor must be (recursively!) reducible to the form lambda(X,Formula) (and is indeed reduced to that form before going on).

If these three conditions are met, the required substitution is made and the result can be further β-converted recursively.

This clause of betaConvert/2 makes use of a predicate substitute/4 (originally implemented by Sterling and Shapiro) that we won't look at in any more detail. It is called like this:

```
substitute(Substitute,For,In, Result).
```

Substitute is substituted for For in In. The result is returned in Result.

1.5.6 Beta-Conversion Continued

Second, there is a clause of betaConvert/2 that deals with those expressions that do not match the first clause. Note that the first clause contains a cut. So, the second clause will deal with all *and only* those expressions whose functor is *not* (reducible to) a λ-abstraction. The only well-formed expressions of that kind are formulas like walk(john) & (lambda(X,talk(X))@john) and atomic formulas with arguments that are possibly still reducible. Apart from that, this clause also applies to predicate symbols, constants and variables (remember that they are all represented as Prolog atoms). It simply returns them unchanged.

```
betaConvert(Formula,Result):-
        compose(Formula,Functor,Formulas),
        betaConvertList(Formulas,ResultFormulas),
        compose(Result,Functor,ResultFormulas).
```

The clause breaks down Formula using the predicate compose/3. This predicate decomposes complex Prolog terms into the functor and a list of its arguments (thus in our case, either the subformulas of a complex formula or the arguments of a predication). For atoms (thus in particular for our representations of predicate symbols, constants and variables), the atom is returned as Functor and the list of arguments is empty.

If the input is not an atom, the arguments or subformulas on the list are recursively reduced themselves. This is done with the help of:


```
betaConvertList([],[]).
betaConvertList([Formula|Others],[Result|ResultOthers]):-
        betaConvert(Formula,Result),
        betaConvertList(Others,ResultOthers).
```

After that, the functor and the reduced arguments/subformulas are put together again using compose/3 the other way round. Finally, the fully reduced formula is returned as Result.

If the input is an atom, the calls to betaConvertList/2 and compose/3 trivially succeed and the atom is returned as Result.

Here is an example query with β-conversion:

```
?- s(Sem,[harry,flies],[]), betaConvert(Sem,Reduced).

Sem = lambda(A,A@mary)@lambda(B,walk(B)), Reduced = fly(harry)
```

Try it for 'Harry curses a witch.': s(Sem,[harry,curses,a,witch],[]), betaConvert(Sem,Res).

?- Question!

Above, we said that complex formulas like fly(harry) & (lambda(x,fly(x))@harry) are split up into their subformulas (which are then in turn β-converted) by the last clause of betaConvert/2. Explain how this is achieved at the example of this particular formula!

1.5.7 Running the Program

We've already seen a first run of our semantically annotated DCG, and we've now implemented a module for β-conversion. So let's plug them together in a driver predicate go/0 to get our first real semantic construction system:

```
go :-
        readLine(Sentence),
        resetVars,
        s(Formula,Sentence,[]),
        nl, print(Formula),
        betaConvert(Formula,Converted),
        nl, print(Converted).
```

This predicate first converts the keyboard input into a list of Prolog atoms. Next, it does some cleaning up that is needed to manage the creation of variable names during lexicon retrieval (see Section 1.5.3). Then it uses the semantically annotated DCG from See file semanticDCG.pl. and tries to parse a sentence.

Next, it prints the unreduced λ-expression produced by the DCG. Finally, the λ-expression is β-converted by our predicate betaConvert/2 and the resulting formula is printed out, too.

In order to run the program, consult runningLambda.pl at a Prolog prompt:

```
1 ?- [runningLambda].
%   comsemOperators compiled into comsemLib 0.00 sec, 520 bytes
%  comsemLib compiled into comsemLib 0.01 sec, 7,612 bytes
%   comsemOperators compiled into betaConversion 0.00 sec, 216 bytes
%  betaConversion compiled into betaConversion 0.00 sec, 1,604 bytes
%   comsemOperators compiled into runningLambda 0.00 sec, 216 bytes
%  semanticDCG compiled into runningLambda 0.01 sec, 4,336 bytes
%  comsemOperators compiled 0.00 sec, 136 bytes
% runningLambda compiled into runningLambda 0.02 sec, 14,848 bytes

Yes
2 ?- go.

> harry flies.

lambda(v1, v1@harry)@lambda(v2, fly(v2))
fly(harry)

Yes
```

Code For This Chapter

Here's a listing of the files needed:

See file `semanticDCG.pl`.	The semantically annotated DCG.
See file `runningLambda.pl`.	The driver predicate.
See file `betaConversion.pl`.	β-conversion.
See file `comsemOperators.pl`.	Definitions of operators used in semantic representations
See file `comsemLib.pl`.	Auxiliary predicates.

Further Reading

The approach we discussed here is a simplified implementation of Richard Montague's ideas (see [11]).

2

Towards a Modular Architecture

In this chapter, we will re-package our grammar and lexicon such that we arrive at a modular framework of semantics construction. Later on, we will be experimenting with a semantic construction techniques differing from λ-calculus. Incorporating these changes, and keeping track of what is going on, requires a disciplined approach towards grammar design. So we will take the opportunity and get to know some basic principles of software engineering on the way. That is, we will re-structure our program such that it is:

Modular: Each component should have a clear role to play and a clean interface with the other components.

Extensible: The grammar should be easy to upgrade, should the need arise.

Reusable: We should be able to reuse a significant portion of the grammar, even when we change the underlying representation language.

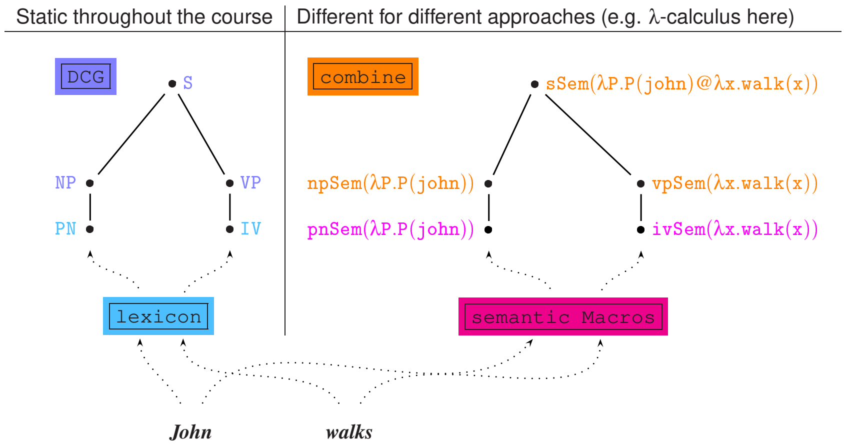
Sticking to these simple principles, we will build an overall framework that will serve us unchanged throughout the rest of this course.

2.1 Architecture of our Grammar

This section gives an overview of the general architecture we've adopted for our flexible and modular implementation of a semantics construction system.

We have adopted a fairly simple grammar architecture that has four (2x2) parts: a *lexicon, DCG-rules, semantic macros,* and *(semantic) combination rules*. Before we go through each of these components in detail, we will give an overview over what exactly their tasks are.

The figure below shows how our system analyses the sentence 'John walks':



Syntax...

Look at the left-hand side of the figure. This side shows the syntactic parts of the grammar. The syntactic analysis consists of two steps: First, a lexicon look-up tells us that 'John' is a proper name (`PN`) and that 'walks' is an intrantitive verb (`IV`). Second, the two non-branching DCG-Rules `NP` → `PN` and `VP` → `IV` tell us that 'John' is also a noun phrase (`NP`) and that 'walks' is also a verb phrase (`VP`). Finally, the DCG-rule `S` → `NP VP` tells us that 'John walks' in fact is a sentence. A grammar that's as simple as that on the side of syntax will do for our purposes.

...and Semantics.

Now look at the right-hand side of the figure. Here, you see the semantic parts of the grammar. The semantic macros for proper names (`pnSem(...)`) and for intransitive verbs (`ivSem(...)`) provide us with the semantic representations of the lexical items, in this case: $\lambda P.P(\mathrm{JOHN})$ and $\lambda x.\mathrm{WALK}(x)$. Then, two so-called `combine`-rules (there is one for each DCG-rule) tell us how to obtain the semantic representations `npSem(...)` and `vpSem(...)` out of the semantic representations `pnSem(...)` and `ivSem(...)`, respectively. Finally, there is a `combine`-rule that tells us how to combine these semantic representations to end up with the semantic representation of the sentence. At present, we're using functional application for this task and get: $\lambda P.P(\mathrm{JOHN})@\lambda x.\mathrm{WALK}(x)$.

Finally a postprocessing step is neccessary in order to get $\mathrm{WALK}(\mathrm{JOHN})$, namely doing β-reduction.

The division of our grammar into syntactic and semantic parts will make life easier for us as semanticists, because once we have specified the lexical entries for the words belonging to the syntactic categories of interest, and once we have formulated the DCG-rules that license buliding all complex phrases we want to deal with, we need not bother with syntax any more. Instead, we can concentrate on the semantic macros that give meaning to the lexical entries, and we can design the combine rules so that they adequately compute the meaning of larger phrases out of the meaning of their parts.

2.2 The Syntax Rules

In this section we turn to the core DCG rules that we are going to use in our semantics construction system.

DCG

But we first have to deal with syntax. So we now turn to the core DCG rules that we are going to use. Let's look at our diagram (page 27): We're in the blue part in the upper half of the left (blue) side. So what we're talking about now will remain fixed throughout the rest of the course. This basically means that for what's to come later, we will consider the problem of syntax as solved.

We will first discuss what syntax rules we *would like* to use. We will immediately see why we can't - and then solve the problem.

2.2.1 Ideal Syntax Rules

DCG

Here are some DCG rules that license a number of semantically important constructions: Proper names, determiners, pronouns, relative clauses, the copula construction, and coordination. In addition, the first two rules allow us to form discourses by stringing together sentences.

The DCG we would like to use

```
s--> [if], s, [then], s.
s--> np, vp.

np--> np, coord, np.
np--> det, noun.
np--> pn.
np--> whnp.
np--> whdet, noun.

noun--> noun, coord, noun.
noun--> noun.
noun--> adj, noun.
noun--> noun, pp.
noun--> noun, rc.
noun--> noun, coord, noun.

vp--> vp, coord, vp.
vp--> v(fin).
vp--> v(fin).
vp--> mod, v(inf).

v(I)--> v(I), coord, v(I).
v(I)--> tv(I), np.
```



```
v(I)--> iv(I).
v(fin)--> cop, np.
v(fin)--> cop, neg, np.

pp--> prep, np.
rc--> relpro, vp.
```

However these are not quite the rules we're actually going to use, for the following reason: The coordination rules are left-recursive, hence the standard Prolog DCG interpreter will loop when given this grammar. As we *do* want to give coordination examples while *not* implementing any parser that deals with left-recursive rules, we're going to adopt adopt an easy fix for this problem.

2.2.2 The Syntax Rules we will use

DCG

Luckily, there's a simple trick that will make a limited form of coordination available to us. We'll add an auxiliary set of categories named `np2`, `np1`, `v2`, `v1`, etc. These auxiliary categories allow us to specify left-recursive rules to a certain depth of recursion. For example, the rules which have something to say about NPs will be replaced by the following:

A DCG allowing only limited recursion.

```
np2--> np1.
np2--> np1, coord, np1.
np1--> det, noun2.
np1--> pn.
```

Similarly, the rules controlling nouns will become:

```
noun2--> noun1.
noun2--> noun1, coord, noun1.
noun1--> noun.
noun1--> noun, pp.
noun1--> noun, rc.
```

While this is a rather blunt way of dealing with the problem of left recursion in a grammar, it enables us to parse the examples we want without having to implement a more sophisticated parser.

Another shortcoming of these rules should be mentioned. As you might have noticed by now, we've imposed limits on inflectional morphology—all our examples are relentlessly third-person present-tense. This is a shame, since tense and its interaction with temporal reference is a particularly rich source of semantic examples. Nonetheless, we shall not be short of interesting things to do.

***See file* `englishGrammar.pl`.**

But for all its shortcomings, this small set of rules (to be found in `englishGrammar.pl`) assigns tree structures to an interesting range of English sentences:

'Mary loves every owner of a siamese cat.'

'John or Mary smokes.'

'Every man that loves a woman visits a therapist.'

'John does not love a therapist or woman that smokes.'

'If a therapist talks then a man works.'

2.3 The Semantic Side

Now we're going to look at the semantic construction copmonent of our new implementation. As we've stated, we plan to use our framework with different semantic formalisms. So the semantic construction part we're going to implement *can't* stay fixed throughout the course. Rather, we'll explicitely want to change it from time to time. And of course we want to be able to do so with as few complications as possible. We'll add a call to an interface predicate (named `combine/3`) to each of our syntax rules. And when we want to upgrade to a different method of semantic construction, we'll often be able to do so by simply re-implementing `combine/3`.

combine

Now we're going to look at the upper right (the reddish) side of our diagram (page 27): Semantic construction. As we've stated above, we plan to use our framework with different semantic formalisms. So the semantic construction part we're going to implement *can't* stay fixed throughout the course. Rather, we'll explicitely want to change it from time to time. And of course we want to be able to do so with as few complications as possible.

Let's recall an observation we made some time ago: Any systematic method of semantic construction has to use the information provided by syntactic structure. So one thing is for sure: As different as they may be, any of the formalisms for semantic construction we possibly come to use will have to communicate with our syntax component. We'll incorporate this insight into our framework as follows: We'll add a call to an interface predicate (named `combine/3`) to each of our syntax rules. And when we want to upgrade to a different method of semantic construction, we'll often be able to do so by simply re-implementing `combine/3`.

2.3.1 The Semantically Annotated Syntax Rules

Here we go providing the clean interface between syntax and semantics construction that we've just promised. Recall that so far, we've simply been using concatenation (indicated by the `@`-operator) to combine semantic representations while parsing a sentence, then β-converting the result in a subsequent post-processing step.

In our examples before, concatenation using the `@`-operator was encoded directly in the DCG. For instance, the `s`-rule looked like this:

```
s(NP@VP) --> np(NP),vp(VP).
```


Each DCG rule is paralleled by a combine-rule.

In view of our grammar engineering principles, this is not a good practice. The crucial keywords here are *modularity* and *reusability*. We shouldn't code one particular mode of *semantic* construction in the *syntactic* rules. Instead, we will encapsulate the particular method in use into a generic predicate `combine/2`. Each DCG rule will include a call to this predicate (to call a predicate with a DCG rule, we have to put it in curly brackets. The predicate is then called whenever the respective rule is applied). The following examples show what our DCG rules now look like:

DCG combine

```
s1(S1)--> np2(NP2), vp2(VP2), {combine(s1:S1,[np2:NP2,vp2:VP2])}.

np1(NP1)--> det(Det), noun2(N2), {combine(np1:NP1,[det:Det,n2:N2])}.

np1(NP1)--> pn(PN), {combine(np1:NP1,[pn:PN])}.
```

The first argument of `combine/2` is always the semantic representation passed on to the superordinate phrase. Now let's look at the way we specify the second argument. We use a little Prolog trick here: In order to uniformly have a binary `combine/2`, no matter how many daughters the syntax rule at hand licenses, we make the second argument of `combine/2` a list. On this list, we put the semantic representations of the daughters, each one tagged with its syntactic category. So there's one item on this list if we are in a unary syntax rule, and two if we are in a binary one.

As a result of our encapsulation strategy, changing the mode of semantic construction is now solely a matter of changing the implementation of `combine/2`, whereas the DCG itself will always remain as shown above. Additionally, we will often need to provide some postprocessing capabilities. These may of course also have to be implemented differently for different semantic construction methods. But given our modular architecture, they can simply be plugged in and out behind the modules we're looking at right now (we will see below (page 37) how all of this is done at the example of β-reduction).

***See file* `englishGrammar.pl`.**

The complete set of annotated DCG rules we will use in this course can be found in `englishGrammar.pl`.

2.3.2 Implementing `combine/2` for Functional Application

combine

Using the predicate `combine/2`, we have factored the task of combining semantic representations out of the syntax rules. As a case study, we will implement the combination technique we've got used to by now: Functional application. So the first task is simply building the obvious 'apply the function to the argument statements' expressed with the help of `@`. We do this in the clauses for the `combine/2`-predicate. Take a look at the `s`-rule of our grammar:

See file **`englishGrammar.pl`.**

```
s1(S1)--> np2(NP2), vp2(VP2), {combine(s1:S1,[np2:NP2,vp2:VP2])}.
```

Look at the call to `combine/2` in the curly brackets. The first argument contains the semantics of the sentence (tagged `s1`), the second argument contains a list with the semantics of the NP and the VP (tagged `np1` and `vp1` respectively).

The purpose of the tags contained within these arguments is to select an appropriate clause of the predicate `combine/2`. We define the `combine/2` predicate for lambda calculus in `lambda.pl`. So let us start by looking at the first clause of `combine/2` that goes with the syntax rule given above. It unifies `S` with `NP@VP`. This looks as follows:

```
combine(s1:(NP@VP),[np2:NP,vp2:VP]).
```

The clause of `combine/2` that goes with the `np` rule

```
np1(NP1)--> det(Det), noun2(N2), {combine(np1:NP1,[det:Det,n2:N2])}.
```

is similarily straightforward:

```
combine(np1:(DET@N),[det:DET,n2:N]).
```

The unary rules, of course, are even simpler, for they merely pass the input representation up to the mother node. For example:

```
combine(np2:X,[np1:X]).
```

goes with

```
np2(NP2)--> np1(NP1), {combine(np2:NP2,[np1:NP1])}.
```

For all combine rules see *See file* `lambda.pl`.

Note again the advantage of using a *list* of tagged semantic representations as the second argument (i.e. the input) of `combine/2`: We've been able to uniformly give clauses for one and the same (binary) predicate for combining the *two* input representations in the binary `s` and `np`-rules, as well as for passing on the *single* input representation in the unary `np`-rule. Guided by matching the tags and lengths of the input lists, Prolog will always select the right clause for us automatically.

2.4 Looking Up the Lexicon

`lexicon` `Semantic Macros`

Up to now, we've been focussing on the upper half of our diagram (page 27), the part where words (and meanings) are combined. Let's now focus on the lower half, where words and meanings come from: Let's look at the lexicon. In a lexical lookup, the syntax component needs to find the syntactic category for a given input token, while the semantic component has to be provided with the associated meaning representation. Our strategy will again be to factor out as much as possible of the semantic side, as this side is what will change with different formalisms.

2.4.1 Lexical Rules

`lexicon`

The combinatorial part of our grammar is connected to the lexicon via the so-called lexical rules. These are the grammar rules that apply to terminal symbols, the actual strings in the input of the parser. They need to call the lexicon to check if a string belongs to the syntactic category searched for, and retrieve its semantic representation.

```
noun(N)--> {lexicon(noun,Sym,Word,_),nounSem(Sym,N)}, Word.
```

`Semantic Macros`

The code that goes with each lexical DCG rule consists of two calls: One to a `lexicon/4` predicate, and one to a binary so-called semantic macro (`nounSem/2` in the example). The `lexicon/4`-call does the actual lexical lookup: If it finds `Phrase` (a list of atoms coming from the input sentence. Most of the time this list will of course contain only one item.) in the category given as first argument, it returns a core semantic representation in `Sym`. As we shall see below (page 34), such a core representation is nothing else than a predicate or constant symbol (hence the variable name `Sym`). This symbol is then further processed by the semantic macro. In our example, the semantic macro `nounSem/2` is used to construct the actual semantic representation for a noun.

Each lexical category is associated with one semantic macro. Using such macros, we can set up the lexicon as well as the lexical rules totally independent from the semantic theory. Note that we're now simply re-doing on the *lexical* level what we did when we introduced the `combine/2`-calls to our *combinatorial* rules: We're factoring out the specific types of structure required by various semantic formalisms into a (set of) interface predicates. This way, we encapsulate this structure and separate it from all other, more or less static information. To change the semantic formalism we will simply re-implement our interface predicates (i.e. the semantic macros) - and that's it.

See file **`englishGrammar.pl`.**

The lexical DCG rules can be found at the bottom of `englishGrammar.pl`.

2.4.2 The Lexicon

`lexicon`

Our lexicon declaratively lists information about the words belonging to most syntactic categories in a very basic form. Technically, it consists of a lot of `lexicon/4`-facts. Thus the general format of a lexical entry is:

```
lexicon(Cat,Sem,Phrase,Misc).
```

Here `Cat` is the syntactic category, `Sem` the core semantic information introduced by the phrase (normally a relation symbol or a constant), `Phrase` the string of words that span the phrase, and `Misc` miscellaneous information depending on the type of entry. In particular, `Misc` may list gender information for nouns, proper names and inflectional information for verbs, etc.

Typical entries for intransitive verbs are:

```
lexicon(pro,nonhuman,[it],[]).

lexicon(pro,male,[him],[]).

lexicon(iv,purr,[purr],inf).

lexicon(iv,smoke,[smokes],fin).
```

Nouns are listed in the following format:

```
lexicon(noun,woman,[woman],[]).

lexicon(noun,siamesecat,[siamese,cat],[]).
```

***See file* `englishLexicon.pl`.**

A complete list of our lexical rules can be found in the file `englishLexicon.pl`. All these rules will work for us unchanged throughout the course.

2.4.3 'Special' Words

lexicon Semantic Macros

There are two classes of words that get a special treatment in our framework:

1. First, look at the following `lexicon/4` facts for determiners:

   ```
   lexicon(det,_,[every],uni).

   lexicon(det,_,[a],indef).
   ```

   Note that these entries contain no semantic information whatsoever. This is because the semantic contribution of determiners is not simply a constant or predicate symbol, but rather a relatively complex expression that is expressed differently in different formalisms. Hence we shall specify the semantics of these categories in the semantic macros alone.

2. Secondly, a small number of important words - in particular, copula and the verb phrase modifier construct 'does not' - are *not* listed in the lexicon at all. This is because they are not associated with either a relation symbol or a constant, and there's no additional information we would like to list for them. For such words, a `lexicon/4` fact would simply list the word form as `Phrase` entry. Instead, we will check their word form directly in our lexical rules (or as one also says: we treat them *syncategorematically*). For example, the following rule handles verb phrase negation:

   ```
   neg(Neg)--> [not], {modSem(neg,Neg)}.
   ```

   Thus in these cases the semantic macros will be the sole source of semantic information.

2.4.4 Semantic Macros for Lambda-Calculus

Semantic Macros

We saw in the last chapter (page 1) that using functional application and β-conversion more or less reduces the process of combining semantic representations to an elegant triviality, while it shifts most of the semantic load to the lexical component. We've got the framework for constructing functional applications and for calling the lexicon. But the only semantic information that our `lexicon/4`-facts supply are the relevant constant and relation symbols. So where do the λs come from? The semantic macros are where the real work will be done. Basically, they will specify the templates for the abstraction patterns associated with different lexical categories. Let's now implement the semantic macros needed for λ-calculus. Here are some examples:

Nouns and proper names

```
nounSem(Sym,lambda(X,Formula)):-
   compose(Formula,Sym,[X]).
```

The first macro, `nounSem/2`, builds a semantic representation for a noun given its predicate symbol `Sym`, by turning this symbol into a formula λ-abstracted with respect to a single variable. For example, given the predicate symbol `man`, it will return the λ-abstraction `lambda(X,man(X))`. The scope of the abstraction is built using `compose/3` to incorporate the given predicate symbol into a well-formed open formula. The semantic macro for proper names (`pnSem/2`) is still simpler: It constructs the kind of λ-expression discussed above (page 15) and doesn't even need to call `compose/3` for this purpose.

Verbs

Let's have a look at the macros for verbs next. The one for intransitive verbs is very straightforward:

```
ivSem(Sym,lambda(X,Formula)):-
   compose(Formula,Sym,[X]).
```

On a closer look, the macro does exactly the same as `nounSem/2`. This is not surprising at all - after all the λ-expressions we saw (page 13) for intransitive verbs are also exactly like the ones for nouns.

Finding the right abstraction pattern for transitive verbs turned out (page 16) to be a little more involved. Nevertheless now we've got it, even this pattern translates into a semantic macro without a glimpse:

```
tvSem(Sym,lambda(K,lambda(Y,K @ lambda(X,Formula)))):-
   compose(Formula,Sym,[Y,X]).
```

This macro is again similar to that for nouns, except that it handles two variables rather than just one. Additionally, it resembles the macro for proper names in the way it incorporates our well known role-reversing trick.

Special words

As we've already mentioned, our grammar also deals with some 'special' words, words that do *not* have a value for a predicate or constant symbol specified in the lexicon. Determiners are such words - and here are the macros for the indefinite and the universal one. They're basically just the old-style 'lexical entries' we used in Chapter 1:

```
detSem(uni,lambda(P,lambda(Q,forall(X,(P@X)>(Q@X))))).
```

```
detSem(indef,lambda(P,lambda(Q,exists(X,(P@X)&(Q@X))))).
```

These macros are self-contained in that they provide a complete semantic representation starting from no input. The first argument is only a tag that helps Prolog select the right clause. It does not occur in the output representation.

All semantic macros can be found in *See file* `lambda.pl`.

For a complete listing of the macros we have been discussing, see the file `lambda.pl`. We shall introduce another semantic construction method in the next chapter - no problem for our new architecture. From now on, we will always use the lexicon and the rules listed above. The *primary locus of change will be the semantic macros and the implementation of* `combine/2`.

2.5 Lambda at Work

Beta-Reduce Afterwards

By now, we have seen all ingredients of our core system in detail, and we've provided almost all of the neccessary plug-ins for λ-based semantic construction. Almost all: Because remember that we still need to post-process the semantic representations our grammar produces, meaning we have to β-reduce them. Luckily, we already have the predicate `betaConvert/2` from the last chapter at our avail for this purpose.

Plugging Together

What's next? Let's plug it all together and provide a user-interface! The predicate `lambda/0` will be our driver predicate:

```
lambda :-
        readLine(Sentence),
        parse(Sentence,Formula),
        betaConvert(Formula,Converted),
        resetVars,vars2atoms(Formula),
        printRepresentations([Converted]).
```

```
parse(Sentence,Formula):-
        s2(Formula,Sentence,[]).
```


First, `readLine(Sentence)` reads in the input. Next, `parse(Sentence,Formula)` tests whether this input is accepted by our grammar as a sentence (the predicate simply calls our DCG to parse a phrase of category `s2`). If the input is a sentence, the λ-expression representing its meaning is returned in `Formula`. For example, for the input 'Tweety smokes.', we get the output `lambda(A, A@tweety)@lambda(B, smoke(B))`. This expression is then β-converted, and finally all remaining Prolog variables in it are replaced by atoms.

Check it out!

Our driver predicate `lambda/0` is contained in the module `lambda`, which is the main module for λ-calculus. Below, we give a listing of all modules that are used by `lambda`. But before that, here is an example call for the sentence 'A therapist loves a siamese cat':

```
lambda([a,therapist,loves,a,siamese,cat],Sem),write(Sem).
```

All modules used by `lambda.pl`

See file `lambda.pl`.	The driver predicate; definition of the `combine`-rules and the le
See file `comsemOperators.pl`.	Operator definitions.
See file `englishGrammar.pl`.	The DCG-rules and the lexicon (using module `englishLex`
See file `englishLexicon.pl`.	The lexical entries for a small fragment of English.
See file `betaConversion.pl`.	β-conversion.
See file `comsemLib.pl`.	Auxiliaries.
See file `signature.pl`.	Generating new variables
See file `readLine.pl`.	Reading the input from stdin.

Further Reading

A textbook introduction of the analysis of language and meaning by methods of formal logic is [8].

3

Scope and Underspecification

In the previous lecture, we have seen how we can compute semantic representations for simple sentences. The formal tool we have used for this purpose was λ-calculus, and the linguistic theory we have followed was Montague Semantics.

Now of course Montague Semantics does not cover all semantic phenomena there are; otherwise semanticists would be out of jobs by now. The good news, however, is that the insights into the structure of semantic representations that Montague gained are so fundamental that many modern semantic theories still uphold Montague Semantics when dealing with simple sentences. Such theories typically come with extensions to the formal framework to allow them the necessary flexibility.

In this lecture, we will investigate the classical case in which Montague Semantics fails to compute the correct meaning(s): *scope ambiguities*, a certain kind of semantic ambiguity. In the first part, we talk about what scope ambiguities are and why the mechanisms we know so far aren't powerful enough to compute them. In the second part of the lecture we learn about a rather modern approach to dealing with scope, based on *underspecification* with *dominance constraints* [5]. Finally, we look at the computational problems connected to this approach and show how to solve them - first in an abstract way and then (in the next Chapter) by means of a Prolog implementation.

3.1 Scope Ambiguities

In this part of the lecture we will learn what *scope ambiguities* are and why such ambiguities constitute a problem for Montague style semantic construction. We will then see some ways of dealing with these problems (more or less satisfactorily) by extending Montague's framework a little.

3.1.1 What Are Scope Ambiguities?

What are scope ambiguities?

A *scope ambiguity* is an ambiguity that occurs when two quantifiers or similar expressions can take scope over each other in different ways in the meaning of a sentence. Here are some examples.

1. 'Every man loves a woman.'

2. 'Every student did not pass the exam.'

Let's look at the first sentence to see the ambiguity. The more prominent meaning of this sentence is that for every man, there is a woman, and it's possible that each man loves a different woman. But the sentence also has a second possible meaning, which says that there is one particular woman who is loved by every man. This reading becomes clearer if we continue the example by "..., namely Brigitte Bardot."

To further underline the difference, have a look at the two readings represented in first-order logic.

1. $\forall x.\textsc{man}(x) \rightarrow (\exists y.\textsc{woman}(y) \land \textsc{love}(x,y))$
2. $\exists y.\textsc{woman}(y) \land (\forall x.\textsc{man}(x) \rightarrow \textsc{love}(x,y))$

They are genuinely semantic...

We see that the sentence has two different meanings: it is *ambiguous*. Moreover, there is no good reason to assume that the ambiguity should be syntactic. So we can say that scope ambiguities are genuine *semantic ambiguities*. It is important to observe here that both readings are made up of the same material (the semantic representations of the quantified NPs 'every man' and 'a woman', and the *nuclear scope* 'love'). The only difference is the way in which the material is put together. We will come back to this later.

...and omnipresent!

The second example shows that not only quantifiers can give rise to scope ambiguities (if you find this particular sentence a little odd, you can play the same game with the German 'Jeder Student hat nicht bestanden.'). In this sentence, it is the relative scope of the quantifier and the negation that is ambiguous. The two readings mean that either every single student failed, or, respectively, that not everyone of the students passed. In formulae:

1. $\forall x.\textsc{student}(x) \rightarrow \neg\textsc{pass}(x)$
2. $\neg\forall x.(\textsc{student}(x) \rightarrow \textsc{pass}(x))$

3.1.2 Scope Ambiguities and Montague Semantics

Using our Implementation

Now let's see what Montague Semantics has to say about sentences containing scope ambiguities like 'Every man loves a woman'. We have just seen that this sentence has two readings, but our implemented system only gets one of them:

```
?- lambda.

> every man loves a woman.

1 forall(A,man(A)>exists(B,woman(B)& love(A,B)))

yes
```

If you like to, reproduce this result at your computer:

```
lambda([every,man,loves,a,woman],Sem).
```

This is a correct representation of one of the possible meanings of the sentence - namely the one where the quantifier of the object-NP occurs inside the scope of the quantifier of the subject-NP. We say that the quantifier of the object-NP has *narrow* scope while the quantifier of the subject-NP has *wide* scope. But the other reading is not generated here! This means our algorithm doesn't represent the linguistic reality correctly.

What's the problem?

This is because our approach so far constructs the semantics *deterministically* from the syntactic analysis. Our implementation simply isn't yet able to compute *two different* meanings for a syntactically unambiguous sentence. The reason why we only get the reading with wide scope for the subject is because in the semantic construction process, the verb semantics is first combined with the object semantics, then with that of the subject. And given the order of the λ-prefixes in our semantic representations, this eventually transports the object semantics inside the subject's scope.

A Closer Look

To understand why our algorithm produces the reading it does (and not the other alternative), let us have a look at the order of applications in the semantic representation as it is before we start β-reducing. To be able to see the order of applications more clearly, we abbreviate the representations for the determiners. E.g. we write Every instead of $\lambda P \lambda Q \forall x(P(x) \to Q(x))$. We will of course have to expand those abbreviations at some point when we want to perform β-reduction.

$$(\mathsf{Every}@\lambda \mathsf{v}.\textsc{man}(\mathsf{v}))@((\lambda \mathsf{P}\lambda \mathsf{x}\mathsf{P}@(\lambda \mathsf{y}.\textsc{love}(\mathsf{x},\mathsf{y})))@(\mathsf{A}@\lambda \mathsf{w}.\textsc{woman}(\mathsf{w})))$$

After β-reducing the VP once, things look a little nicer:

i. $(\mathsf{Every}@\lambda \mathsf{v}.\textsc{man}(\mathsf{v}))@(\lambda \mathsf{x}(\mathsf{A}@\lambda \mathsf{w}.\textsc{woman}(\mathsf{w}))@(\lambda \mathsf{y}.\textsc{love}(\mathsf{x},\mathsf{y})))$

The resulting expression is an application. The universal quantifier occurs in the functor (the translation of the subject NP), and the existential quantifier occurs in the argument (corresponding to the VP). The scope relations in the β-reduced result reflect the structure in this application.

An Idea for a Solution

With some imagination we can already guess what an algorithm would have to do in order to produce the second reading we've seen above (where the subject-NP has narrow scope): It would somehow have to move the $\mathsf{A}@\lambda \mathsf{y}\textsc{woman}(\mathsf{y})$ part in front of the Every. Something like the following expression would do:

ii. $(\mathsf{A}@\lambda \mathsf{w}.\textsc{woman}(\mathsf{w}))@(\lambda \mathsf{y}.(\mathsf{Every}@\lambda \mathsf{v}.\textsc{man}(\mathsf{v}))@(\lambda \mathsf{x}.\textsc{love}(\mathsf{x},\mathsf{y})))$

3.1.3 A More Complex Example

So by sticking to the principles of classical Montague grammar, our implementation can only construct *one* reading per sentence. You may think: But hey, at least we get half of what we want. If this is what you think, the following example should convince you that things are really worse:

'Every owner of a siamese cat loves a therapist.'

We have a scope ambiguity between three quantifiers in the example sentence. Below are four possible readings that are pretty easy to get for most speakers of English, and a fifth one which we will examine a little closer soon:

1. $\forall x.(\textsc{owner}(x) \wedge \exists y.\textsc{siamesecat}(y) \wedge \textsc{of}(y,x)) \to \exists z.\textsc{therapist}(z) \wedge \textsc{love}(x,z)$
   $(\mathsf{Every}@\lambda \mathsf{x}.(\textsc{owner}(\mathsf{x}) \wedge ((\mathsf{A}@\mathsf{s\_cat})@\lambda \mathsf{y}.\textsc{of}(\mathsf{y},\mathsf{x}))))@(\lambda \mathsf{x}(\mathsf{A}@\mathsf{therapist})@\lambda \mathsf{z}.\textsc{love}(\mathsf{x},\mathsf{z}))$

2. $\exists y.\textsc{siamesecat}(y) \wedge (\forall x.(\textsc{owner}(x) \wedge \textsc{of}(y,x)) \to \exists z.\textsc{therapist}(z) \wedge \textsc{love}(x,z))$
   $(\mathsf{A}@\mathsf{s\_cat})@(\lambda \mathsf{y}.[(\mathsf{Every}@\lambda \mathsf{x}.[\textsc{owner}(\mathsf{x}) \wedge \textsc{of}(\mathsf{y},\mathsf{x})])@\lambda \mathsf{z}.[(\mathsf{A}@\mathsf{therapist})@\lambda \mathsf{z}\textsc{love}(\mathsf{x},\mathsf{z})]])$

3. $\exists z.\textsc{therapist}(z) \wedge \forall x.(\textsc{owner}(x) \wedge \exists y.\textsc{siamesecat}(y) \wedge \textsc{of}(y,x)) \to \textsc{love}(x,z)$
   $(\mathsf{A}@\mathsf{therapist})@\lambda \mathsf{z}.[(\mathsf{Every}@\lambda \mathsf{x}.[\textsc{owner}(\mathsf{x}) \wedge (\mathsf{A}@\mathsf{s\_cat})@\lambda \mathsf{y}.\mathsf{of}(\mathsf{y},\mathsf{x})])@\lambda \mathsf{x}.\textsc{love}(\mathsf{x},\mathsf{z})]$

4. $\exists y.(\textsc{siamesecat}(y) \wedge \exists z.\textsc{therapist}(z) \wedge \forall x.((\textsc{owner}(x) \wedge \textsc{of}(y,x)) \to \textsc{love}(x,z)))$
   $(\mathsf{A}@\mathsf{s\_cat})\lambda \mathsf{y}.[(\mathsf{A}@\mathsf{therapist})@\lambda \mathsf{z}.[(\mathsf{Every}@\lambda \mathsf{x}.[\textsc{owner}(\mathsf{x}) \wedge \textsc{of}(\mathsf{y},\mathsf{x})])@\lambda \mathsf{x}.\textsc{love}(\mathsf{x},\mathsf{z})]]$

5. $\exists z.\textsc{therapist}(z) \wedge \exists y.\textsc{siamesecat}(y) \wedge \forall x.((\textsc{owner}(x) \wedge \textsc{of}(y,x)) \to \textsc{love}(x,z))$
   $(\mathsf{A}@\mathsf{therapist})@\lambda \mathsf{z}.[(\mathsf{A}@\mathsf{s\_cat})\lambda \mathsf{y}.[(\mathsf{Every}@\lambda \mathsf{x}.[\textsc{owner}(\mathsf{x}) \wedge \textsc{of}(\mathsf{y},\mathsf{x})])@\lambda \mathsf{x}.\textsc{love}(\mathsf{x},\mathsf{z})]]$

We have also given an equivalent expression for each of the readings that uses abbreviations for the determiners, and additionally abbreviates some of the less complex λ-expressions (if you like to, see for yourself by expanding and β-reducing). This should give you an intuition of how the differences in meaning between the readings actually go back to different ways of ordering the determiners.

So far only the first reading can be produced by our implementation. Again the order of the quantifiers in this reading quite directly reflects the relations between the corresponding NPs in the syntax tree. For instance 'a therapist' is a constituent of the VP 'loves a therapist'. Thus its quantifier is in the scope of the universal quantifier of the subject NP. The same goes for the existential quantifier of 'a siamese cat', because the phrase is a constituent of the subject NP.

3.1.4 The Fifth Reading

Two readings may be equivalent...

If you have already looked closely at all the readings we have listed for the complex example, you will have noticed that the fourth and fifth readings are logically equivalent.

...but not necessarily

The reason why we have listed readings four and five separately in spite of this is that there are structurally identical examples (which just use other determiners) in which the two readings do mean different things. Consider the sentence 'Every researcher of a company saw most samples.' Because of the determiner "most", the readings of this sentence can't be represented in first-order logic, but we can use Most as the analogue of Every and A in the λ terms. We are then able to write the semantic representations of the fourth (1.) and fifth (2.) reading of the previous example as follows:

1. $(\mathsf{A@company})\lambda x.[(\mathsf{Most@sample})@\lambda y.[(\mathsf{Every}@\lambda z.[\text{RESEARCHER}(z) \wedge \text{OF}(z,x)])@\lambda z.\text{SEE}($
2. $(\mathsf{Most@sample})@\lambda y.[(\mathsf{A@company})\lambda x.[(\mathsf{Every}@\lambda z.[\text{RESEARCHER}(z) \wedge \text{OF}(z,x)])@\lambda z.\text{SEE}($

3.1.5 Montague's Approach to the Scope Problem

Of course, linguists soon became well aware of the fact that Montague Grammar had to do something about scope. Montague himself extended his formalism with an operation called *quantifying in* to remedy the problem.

Basically, his idea was to postulate *two* alternative *syntactic* analyses of sentences like 'Every man loves a woman':

1. The sentence is taken to consist of the NP 'Every man' and the VP 'loves a woman'. This is the analysis we're used to. We already know that this analysis gives us the formula $\forall x.(\text{MAN}(x) \rightarrow (\exists y.\text{WOMAN}(y) \wedge \text{LOVE}(x,y)))$, where 'Every man' has scope over 'a woman'.
2. Alternatively, the sentence is analysed in a way that may be paraphrased as 'A woman - every man loves her.'. (Of course 'her' in this paraphrase refers to the woman introduced by the NP 'a woman'). For semantic construction, this means that the representation for the whole sentence is built by applying the translation of 'a woman' to the translation of 'every man loves her'. This analysis yields the reading where 'a woman' outscopes 'every man'.

To make the second analysis work, one has to think of a representation for the pronoun, and one must provide for linking the pronoun to its antecedent 'a woman' later in the semantic construction process. Intuitively, the pronoun itself is semantically empty. Now Montague's idea essentially was to choose a new variable to represent the pronoun. Additionally, he had to secure that this variable ends up in the right place after β-reduction.

3.1.6 Quantifying In: An Example

Let's look at our example from before (page 43): Suppose we chose the variable v_1 for the pronoun 'her'. But we want to be able to use this pronoun like a quantified NP that would usually stand in the same place. Eventually, it should end up in the second argument slot of WOMAN. So we will wrap it in a λ-expression as follows: $\lambda P.P(v_1)$. This should ring a bell - we did the same thing for proper names, for example when translating 'John' as $\lambda P.P(\text{JOHN})$.

Introduce a pleaceholder...

In effect, the sentence 'Every man loves her' yields the representation

$$(\lambda P \lambda Q \forall x.(P(x) \rightarrow Q(x))@\lambda y.\text{MAN}(y))@(\lambda R \lambda x R@\lambda y.\text{LOVE}(x,y)@\lambda P.P(v_1))$$

which can be β-reduced to:

$$\forall x(\text{MAN}(x) \rightarrow \text{LOVE}(x, v_1))$$

So what remains to be done? We still have to process the antecedent for our pronoun, namely the phrase 'a woman', translated as $\lambda Q \exists y.(\text{WOMAN}(y) \wedge Q(y))$. And of course, our pronoun variable should be connected to this antecedent: We eventually want the second argument position of $\text{LOVE}(x, v_1)$ to be bound by the existential 'woman-quantifier'. This is achieved by λ-abstracting over the pronoun variable v_1 and then applying the translation of 'a woman' to the resulting abstraction:

...and eliminate it again.

$$\lambda Q \exists y.(\text{WOMAN}(y) \wedge Q(y))@(\lambda v_1.\forall x(\text{MAN}(x) \rightarrow \text{LOVE}(x, v_1)))$$

This reduces to:

$$\exists y.(\text{WOMAN}(y) \wedge \forall x(\text{MAN}(x) \rightarrow \text{LOVE}(x,y)))$$

We've finally got the reading where 'a woman' has scope over 'every man'. The basic trick was to find a way to *delay* processing the NP 'a woman' until we have processed 'every man', thus lifting the existential quantifier above the universal. Implementing this trick of course required quite a piece of sophisticated λ-programming.

3.1.7 Other Traditional Solutions

So we have managed to construct the second reading for our sentence. At a price, though: in order to solve a semantic problem, we had to postulate an alternative syntactic analysis for no obvious syntactic reason - and a rather unintuitive and strange one at that; one that employs a pronoun that doesn't surface in the sentence itself. The fundamental problem that each syntactic analysis still can have only one possible meaning remains.

A more Elegant Solution

In 1975, Robin Cooper proposed a much more elegant mechanism to solve this problem. It became known as *Cooper Storage* . This mechanism took up Montagues idea of lifting quantifiers by using ‘placeholders’ (like pronoun variables) as arguments instead of quantified NPs, and accessing these placeholders later at different points during the semantic construction process. Cooper started with a syntax tree whose leaves had been annotated with the λ-terms representing the semantics of the words. Then he performed bottom-up semantic composition as we have seen it above, but whenever he had to combine an NP and a verb or VP, he could not only immediately apply the NP semantics to the verb semantics, but alternatively use a placeholder and put the NP semantics into a *quantifier store* . This way, he could potentially collect a lot of quantifiers whose application he wanted to delay on his way up in the tree. Whenever he hit a sentence node, his algorithm could pick some or all of the quantifiers and apply them to the current semantics, in any order, thus generating all possible permutations of quantifiers.

A Pseudo-reading

Cooper’s algorithm was a big step forward, but it suffered from an overgeneration problem. For example, it generated a sixth reading for the three-quantifier sentence we’ve seen above. The problem with Cooper’s approach was that it liberally assumed that you can obtain readings by simply permuting the quantifiers, and that each formula obtained that way would represent a possible reading as well. In this respect it did not differ too much from Montague’s technique of quantifying in. However, this assumption is not true. Look at the following formula. It is another permutation of the three quantifiers in our siamese-cat-example, but it is not a possible reading.

$$*\forall x.\mathrm{OWNER}(x) \wedge \mathrm{OF}(y,x) \wedge \exists y.\mathrm{SIAMESECAT}(y) \rightarrow \exists z.\mathrm{THERAPIST}(z) \wedge \mathrm{LOVE}(x,z)$$

Advanced Solutions

In 1988, Keller managed to fix this overgeneration problem in a modified Cooper Storage mechanism called *Nested Cooper Storage* (or simply *Keller Storage* , see [10]). By the mid-Eighties, algorithms like this one or Hobbs and Shieber’s (1987) scoping algorithm allowed to enumerate the readings of a scope ambiguity reasonably well.

3.1.8 The Problem with the Traditional Approaches

By the time most linguists were satisfied with having algorithms that computed the readings of a scope ambiguity in a reasonably elegant way, the more computationally minded researchers started to become a bit unhappy. Their problem was that they tried to build practical language-processing systems, and it turned out that ambiguities (including, but not limited to scope) were a major efficiency problem.

Combinatorial Explosion

The problem is one of *combinatorial explosion* . We’ve already seen above that a scopally ambiguous sentence with two quantifiers has two readings, and one with three quantifiers has five readings. The number of readings for similar sentences increases as follows:

number of quantifiers	readings
4	14
5	42
6	132
7	429
8	1430

As you can see, the number of readings grows exponentially with the number of quantifiers in the sentence. Now imagine that you wanted to do something interesting with the possible meanings of your sentence - for example, feed them to a theorem prover for inferences, as we will learn to do later in this course. Such operations are expensive even on a single reading, but they become completely unfeasible for 1430 readings. This is particularly annoying because the vast majority of these readings may be theoretically possible, and thus *must* be predicted by the theory. Still most readings will not be intended by the speaker in the particular situation. Thus an NLP system spends a lot of time on expensive computations, most of which are probably irrelevant.

The problem is serious.

At this point, you might argue that sentences that contain so many quantifiers are very rare, but in the words of Jerry Hobbs, ‘Many people feel that most sentences exhibit too few quantifier scope ambiguities for much effort to be devoted to this problem, but a casual inspection of several sentences from any text should convince almost everyone otherwise.’ Besides, you should bear in mind that not only NPs, but also negation, some verbs (e.g. ‘believe’) and adverbs (‘possibly, sometimes, always’) take scope, the basic combinatoric principles applying to these as well. Finally, scope is of course only one source of ambiguity, and the numbers of readings for each type of ambiguity multiply. The bottom line is that ambiguity in general is one of the big challenges for efficient natural-language processing today; scope ambiguities are just one of many culprits in this respect.

3.2 Underspecification

In the rest of this lecture, we will explore algorithms that do *not* enumerate *all* readings from a syntactic analysis, but instead derive just *one*, *underspecified description* of them all. It will still be possible to efficiently extract all readings from the description, but we want to delay this enumeration step for as long as possible. At the same time, the description itself will be very compact (not much bigger than one single reading), and we will be able to compute a description from a syntactic analysis efficiently.

3.2.1 Introduction

A Clean and Declarative Approach

So basically, we are going to *separate semantic construction from the enumeration* of readings of ambiguities. We thus divide the problem into two independent parts, which we can in turn solve independently. This means we can stick to our original setup, where we derive *one* representation from *one* syntactic analysis, only now this representation is the *description* of a whole set of readings. It also means we can take a more *declarative perspective* on scope ambiguity: First of all, we specify what readings a sentence should have, and in a second step we can think about how to actually compute them. We call this step of enumerating the single readings *solving*. Our algorithm for this task will turn out to be quite an elegant one, constituting a great step forward from traditional Cooper or Hobbs style algorithms, which not only had to think about the structure of the semantics, but also about syntactic considerations.

Let us now sum up our discussion so far, using a few pictures. Then we illustrate how our new underspecification based approach relates to the Montague style semantic construction system from the last chapters, and to its extensions that we discussed in the first part of this chapter.

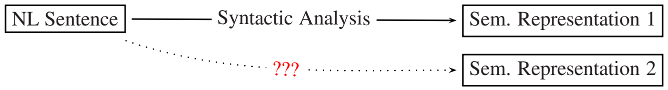
Here's a schema of how we get from a sentence to its semantic representation in the standard case that our Montague style system covers: Unambiguous sentences like 'John loves Mary'.

Standard Montague

We've discussed a much more detailed version of this picture in the last chapter- the semantic representation of the sentence is constructed via and along with its syntactic analysis. One syntactic analysis can only yield one semantic representation. Now since we've assumed that our input sentence is unambiguous, that's fine. There is in fact only one semantic representation for it.

But as we have seen in the first sections of this chapter, there are sentences that contain genuinely semantic ambiguities. The paradigmatic case we've looked at is that of quantifier scope ambiguities as in 'Every man loves a woman'. The following graphic depicts the situation when we feed that sentence to our Montague style semantic construction system:

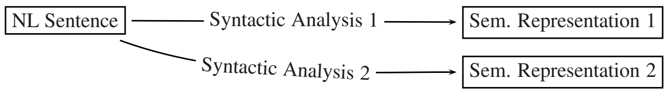
The Problem



There are two semantic representations that should be associated with our input sentence, due to the scope ambiguity in it. But our system can only construct one of them. That's because there's only one syntactic analysis for the sentence, and as we've just mentioned, one syntactic analysis can only yield one semantic representation.

So if we don't want to change anything substantial in the approach we've implemented, there seems to be only one way to get to the second reading. That is to allow a second syntactic analysis.

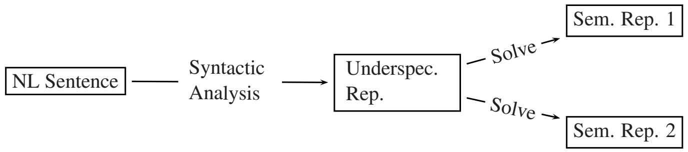
Montague with Quantifying In



Now we would be able to construct the second semantic representation together with this second syntactic analysis. As we've said (in Section 3.1.5), this is the solution that Montague himself adopted. But we've also discussed that there's one strong and obvious argument against this solution: Scope ambiguities simply are not syntactic. According to our intuitions, our example sentence is syntactically unambiguous, and so we should not for purely technical reasons claim the opposite.

Underspecification allows for a more satifactory solution to our problem:

Underspecification



We have split the 'semantic side' of our picture in two levels. On one level we have underspecified descriptions, and on the other one the semantic representations we're used to (i.e. λ-expressions and - at the end of the day - first order formulae). With this two-leveled architecture we can again construct *one* underspecified description along with only *one* syntactic analysis. But this one underspecified description sometimes describes *many* readings on the level of λ-expressions. This means that we have now captured the semantic ambiguity in truly semantic terms. Our first-level semantic representation (the underspecified description) remains ambiguous between multiple second-level semantic representations (λ-expressions) in the same way as the original sentence.

Terminology

Before we go on, let us sort out our terminology a bit. Up to now, we've used the term 'semantic construction' to denote the whole business of getting from natural language sentences to first order formulas. From now on, we will often have to differentiate a bit more. We will then use 'semantic construction' in a more narrow sense, only for the way from natural language sentences to underspecified descriptions. We will call the step from underspecified descriptions to λ-expressions *solving*.

As regards the term 'semantic representation', we'll sometimes use it as an umbrella term for underspecified descriptions as well as λ-expressions and first order formulas. But whenever it is important, we will carefully distinguish between the three.

3.2.2 Computational Advantages

Outlook

From a computational perspective the central hope connected with underspecification is that we will be able to overcome the problems arising out of the combinatorial explosion. We don't have the time here to go into it, but people have shown how to lift β-reduction and even some first-order deduction to underspecified descriptions. More than anything, however, underspecification may be an ideal platform when it comes to *incorporating external information* that excludes irrelevant readings. We have seen above that the theoretically possible number of readings of a sentence may be much higher than the number of readings that are actually possible in a given context. People have preferences for certain readings (e.g. going back to the word order), or they may judge some readings implausible. Underspecification may make it possible to exclude impossible or dispreferred readings *without ever seeing them*. But this is ongoing research and beyond the scope of this introduction.

3.2.3 Underspecified Descriptions

The first thing we need to do now is to render the notion of underspecified description more precise. To see how we can describe all readings of an ambiguous sentence, let's go back to our favourite example, 'Every man loves a woman.' We've said that the two readings of the sentence are these:

1. $\forall x.\textsc{man}(x) \to (\exists y.\textsc{woman}(y) \land \textsc{loves}(x,y))$
2. $\exists y.\mathsf{woman}(y) \land (\forall x.\mathsf{man}(x) \to \textsc{love}(x,y))$

Assessing the material...

Now the important observation is that both readings consist of the same material: the representations of the *two quantified NPs* and the *nuclear scope*. The difference is in the way that these three fragments are put together. Both quantifiers must have scope over $\textsc{love}(x,y)$, but they can still have scope over each other in either way.

...and describing its combinations.

If you have a closer look at what we've just said, you'll notice that this is a *description* of the two possible readings - in an informal way, of course. Underspecification formalisms are all about making such descriptions more formal: They specify what *material* the readings of a sentence consist of (in our example, the three formula fragments), and what *structural constraints* one must obey when configuring them into complete formulas. What is left underspecified is which of these readings is the "right" reading of a specific utterance of the sentence.

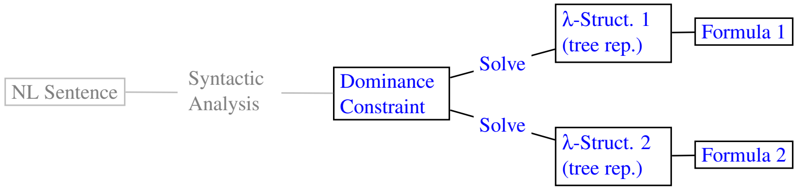
3.2.4 The Masterplan

In the rest of this chapter and in the next one, we will go into the details of underspecification. What exactly are we going to do? We will first give you an intuition of what the formalism to be presented does, and then make this intuition more formal. Here's how we will proceed:

1. In order to give underspecified descriptions of possible readings, the first thing we need is a way of talking about the structure of formulas and of λ-expressions. We will represent formulas and λ-expressions as *trees*. So to begin with (Section 3.2.5), we'll explain how to do this.

2. Then (in Section 3.2.6) we will introduce a formalism that allows us to *describe* trees (and thereby formulas and λ-expressions). We will use the language of *normal dominance constraints* for our formalism, in the form of *constraint graphs*. As a concrete example, we will look at the *two* λ-expressions (written as trees!) for our running example 'Every man loves a woman' (Section 3.2.7) and see how we can represent them using only *one* underspecified description from our new formalism. We will learn how we can construct this description from the two λ-expressions.
3. Once we know how an underspecified description describes (one or more) tree representations of λ-expressions, we turn to the question that's most important when we build semantic representations for a sentence: How do we solve underspecified descriptions? That is, given an underspecified description, how can we compute the formulae it describes? This involves a process called *constraint solving* . In the rest of this chapter we will give you a first intuition of what the problem is that we have to deal with (Section 3.2.8). In the next chapter, we will then continue our discussion by formulating an algorithm that incorporates this intuition. Section 4.1 introduces the basic concepts used in that algorithm. In Section 4.2 we consider one by one each of its subtasks.

Below you see again the general picture of underspecification-based semantic construction that you know from Section 3.2.1. But this time we've marked in blue what we will have dealt with when we're through with the three points just mentioned. Additionally we've filled in the boxes with the types of representation we're actually going to use:



Now you probably wonder: Isn't there something missing? What about the grey part of the picture above? We plan to discuss at length how underspecified descriptions relate to formulas, and even give an algorithm that constructs the latter from the former. But we seem to keep secret how to get from natural language sentences to underspecified descriptions...

You are right with this observation! At that stage we will not yet know how to construct e.g the one underspecified description of the two readings of 'Every man loves a woman' from this sentence. And of course we have to know how to do this. Yet we will not bother about this task until the very end of the next chapter (Section 4.4), when we actually implement semantic construction based on our underspecification formalism.

The reason for this postponement is that the actual construction of underspecified descriptions from sentences is by far the easiest step in our new semantic construction system. It's less complicated than the subsequent step of constraint solving, and it's even less complicated than the direct construction of λ-expressions that we're used to from our Montague based approach.

3.2.5 Formulas are trees!

Tree Notation

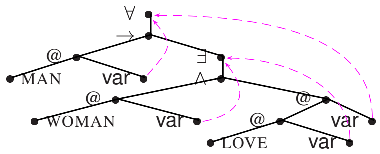
We've just said that we're going to develop a formalism that allows us to describe *trees*. But why trees? Shouldn't we talk about formulas? The answer is that formulas are trees - if you look at them the right way. Representing formulas as trees is simple. You know that every formula of first order logic has a *main connective*. For instance, a conjunction $\varphi \wedge \psi$ has the main connective $\wedge$ and the subformulas φ and ψ. So if we know how to represent the two subformulas as trees, we can represent the whole formula as a tree whose root node is labeled with the symbol $\wedge$ and the two trees for φ and ψ as children. This works similarly with the other connectives. The leaves of the tree are predicate symbols, constants, and variables.

New Representation of Atomic Formulae!

Finally, on the level of atomic formulas, we shall from now on write application of predicates to arguments with the binary symbol @. (We have already seen this: We indicate applications in λ-calculus the same way). Here's one of our standard example formulae in this notation (if you're interested in the motivation behind our decision to use this new notation, read the sidetrack (page 56) at the end of this chapter):

1. $\forall x.(\textsc{man}@x) \rightarrow (\exists y.(\textsc{woman}@y) \wedge ((\textsc{love}@x)@y))$
2. $\exists y.(\textsc{woman}@y) \wedge (\forall x.(\textsc{man}@x) \rightarrow ((\textsc{love}@x)@y))$

Now have a look at the following tree representation of this formula:



?- Question!

In one respect, the tree above differs from the formula it represents. Do you see where?

Binding Edges

The answer to this question is that variables are represented differently in the tree representation. We could have used variable names as we always have. But we will see later that this would have lead us into problems when writing underspecified descriptions. That's why we explicitly link bound variables to their binders via *binding edge* s. These are depicted as purple arrows in the picture above. Since these binding edges tell us all there is to know about which variables are bound by which binders, we can do away with variable names altogether, and it is sufficient to label variable nodes with the symbol var and quantifier nodes with the symbols $\exists$ and $\forall$. We will see in the implementation section that this way of handling variable binding will even simplify our implementation.

3.2.6 Describing Lambda-Structures

There is of course no reason to restrict our new tree notation to formulas of first-order logic. We can just as easily represent λ-expressions. All we have to do is to generally represent application as a tree with the root symbol @ and subtrees for the functor and the argument, and λ-abstraction as a tree with the root symbol lam. Again, we use binding edges to represent variable binding, and thus don't have to give a name to the variable bound by the λ. We call a tree with binding edges for variable binding a *λ-structure* . We can always convert a λ-expression (or a formula) into a unique λ-structure. At the same time, every λ-structure represents a λ-expression (but not uniquely): All we have to do is invent a new variable name whenever we hit a binder, and then use this name for all bound variables.

Let's look again at the λ-expressions that lead to the two first order formulae for 'Every man loves a woman'. We have seen these λ-expressions in Section 3.1.2, and repeat them here:

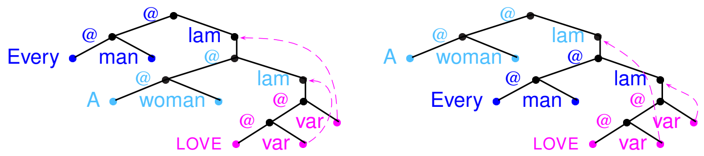
i. $(\mathsf{Every}@\lambda v.(\textsc{man}@v))@(\lambda x(\mathsf{A}@\lambda w.(\textsc{woman}@w))@(\lambda y.((\textsc{love}@y)@x)))$

ii. $(\mathsf{A}@\lambda w.(\textsc{woman}@w))@(\lambda y.(\mathsf{Every}@\lambda v.(\textsc{man}@v))@(\lambda x.((\textsc{love}@y)@x)))$

We have switched again to representations where we abbreviate determiners such as 'every': Every stands for $\lambda P\lambda Q\forall x(P@x \rightarrow Q@x)$. In the tree representations that we look at now, we will continue to use such abbreviations, and also abbreviate the simple λ-expressions and λ-structures for common nouns. Hence, from now on Every abbreviates the λ-structure for $\lambda P\lambda Q\forall x(P@x \rightarrow Q@x)$, and e.g. woman stands for the λ-structure for $\lambda w.\textsc{woman}@w$.

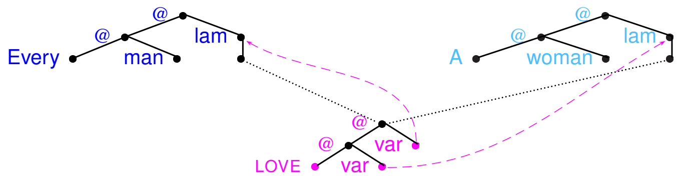
Two lambda-structures...

If we represent the two readings of our example as λ-structures, we can identify the three formula fragments relevant for the scope ambiguity we're interested in as three tree fragments. We have given them different colours in the picture below.



...but only one constraint graph

Now we can represent the information that is common to both readings in the following graph:



We call a graph as in this picture a *constraint graph* . A constraint graph is a directed graph that has node labels and three kinds of edges: ordinary solid edges, dotted *dominance edge* s, and purple arrow *binding edge* s. It consists of several little tree fragments which are internally connected with solid edges, and connected to other trees with dominance edges. Binding edges generally go from variable nodes to binder nodes.

3.2.7 From Lambda-Expressions to an Underspecified Description

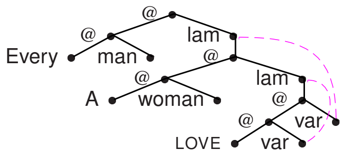
Let us look at our example once more and go through *step by step* how we have constructed the constraint graph describing our two λ-expressions. We had to take four steps:

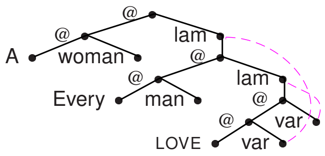
1. We wrote down all (two) readings of the sentence, as λ-expressions:

   (a) $(\mathsf{Every}@\lambda x.\mathrm{MAN}(x))@(\lambda x.(\mathsf{A}@\lambda y.\mathrm{WOMAN}(y))@\lambda y.\mathrm{LOVE}(x,y))$

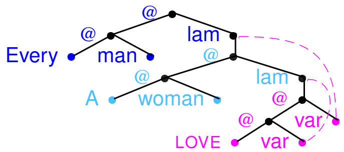
   (b) $(\mathsf{A}@\lambda y.\mathrm{WOMAN}(y))@\lambda y.(\mathsf{Every}@\lambda x.\mathrm{MAN}(x))@(\lambda x.\mathrm{LOVE}(x,y))$

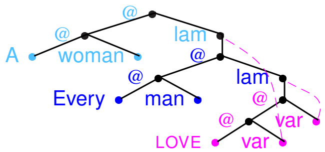
2. We converted the readings into λ-structures:



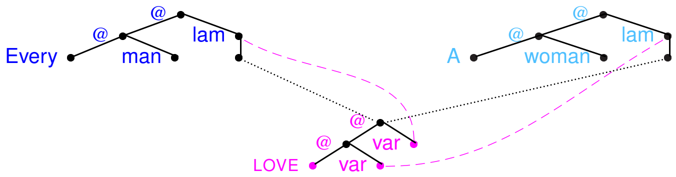


3. We identified the common material in both λ-structures. Generally, each block of common material must be contiguous (linked internally with only solid edges). It may be a complete subtree (like the purple part), or it may be just a tree fragment (like the other two parts).





4. We built an underspecified description that expresses what material the readings contain, and what structural constraints we must obey when putting that material together.



What we have just done, namely going from a natural language sentence via all its readings to an underspecified description, does not correspond to any part of our system architecture (page 49). We started off from fully specified λ-structures. But once we hold all λ-structures for a sentence in our hands, there is of course no point in constructing an underspecified description any more. Yet we hope that our discussion has given you a better idea of how this whole underspecification business works. Our explanations should enable you to solve the following exercise.

3.2.8 Relating Constraint Graphs and Lambda-Structures

We've just seen pictures that gave us an intuitive idea of how λ-structures relate to constraint graphs. Let's now frame our intuition into a more formal definition. We can say that a constraint graph *describe* s a λ-structure if it's possible to *embed* the tree fragments into the λ-structure. (In this case we also say that the λ-structure is a *solution* of the constraint graph.) That is, we must be able to map the nodes of the constraint graph to nodes of the λ-structure in a way that satisfies the following conditions:

1. Any node that has a label in the graph must have the same label in the λ-structure.
2. No two nodes that have a label in the graph must be mapped to the same node in the λ-structure.
3. Any two nodes connected with a solid edge or a binding edge in the graph must be connected in the same way in the λ-structure.
4. Whenever there is a dominance edge from a node X to a node Y in the graph, there must be a path from X to Y using only solid edges in the λ-structure.

Intuitively again, embedding a constraint graph into a λ-structure is a bit of a jigsaw puzzle: Overlay parts of the λ-structure with matching tree fragments so that no two fragments overlap and all the dominances are respected. If you start with a constraint graph and want to construct λ-structures that it describes, the puzzle character comes out even more strongly, as you basically have to configure the tree fragments into a valid λ-structure.

In the example, it is clearly possible to embed the fragments in the graph into each of the two λ-structures; the embeddings indicated by the colouring also respect the dominance requirements. Note that while different fragments do overlap at the borders, there never are any two labeled nodes that are mapped to the same node in the structure.

3.2.9 Sidetrack: Constraint Graphs - The True Story

In the example, we have been able to cover the complete λ-structure with the fragments in the constraint graph. This need not be the case in general: As the fragments only have to be *embedded* into the λ-structure, it is possible that the latter contains some material not mentioned in the graph.

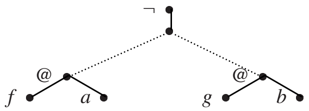
More Flexibility

This makes sense from a computational point of view, considering that constraint graphs are provably harder to deal with if solutions are not to contain additional material. From a linguistic point of view, one can take the idea behind underspecification even further, using the same formalism to deal not only with scope ambiguities, but also with cases where, for example, a speech recognizer has failed to recognize certain parts of the input. In such cases, we *want* flexibility to add more material to a solution - in a controlled way, of course.

Embedding vs. Configuring

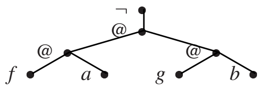
We will ignore this point here and assume that we'll never need to invent any additional material to solve the constraint graphs we get for scope ambiguities. So we can think of the process of solving a constraint as *configuring* the fragments into a bigger tree. It can be shown empirically that the distinction between configuring and embedding makes no difference in practice [7].

The difference between embedding and configuring may become clearer with an example given. Consider the following constraint graph:



Additional material...

This constraint graph trivially has a solution. It starts with the topmost fragment. Then we put an arbitrary label (like @) at the leaf of this fragment and say that this node should have two children. The left child should be the root of the lower left fragment, while the other child should be the root of the lower right one:



...or not?

But if we insist that we have to configure the fragments, without adding any new material, the answer is not so clear. Only if one of the two lower fragments had an unlabeled leaf, it would indeed be possible to configure them by attaching the other fragment to this leaf. Otherwise, it is impossible to configure them; this is the case in the example. Of course, it might take us a long time to figure out that we have plugged fragments together in the wrong order if the graph is larger. That's why computation becomes harder when we restrict ourselves to configuration instead of embedding.

Although we have presented constraint graphs a bit informally here, they can be given a very precise meaning as a shorthand notation for logical formulas, which then are called *normal dominance constraints* . In fact, if you look at the literature on dominance constraints [5], you'll find that the logical formulas are always the first concept to be defined, and the constraint graphs are then derived from them.

3.2.10 Sidetrack: Predicates versus Functions

When we introduced our tree notation for formulas (in Section 3.2.5) we also said that we use the application symbol @ in atomic formulas of first order logic and their tree representations. So we write for instance the application WALK@MARY instead of WALK(MARY) and the two nested applications (LOVE@MARY)@JOHN instead of LOVE(JOHN, MARY). We've introduced this as a simple change of notation, but in fact there's a somewhat deeper motivation behind it.

Using function symbols

Formally, the semantics of λ-expressions is generally defined in terms of functions, and the symbol @ is understood as *functional* application of the functor (to the left of the @) to the argument (to the right). Thus the semantics of an application $\mathcal{A}@\mathcal{B}$ is the result of applying the function denoted by $\mathcal{A}$ to whatever is denoted by $\mathcal{B}$. Now with our new notation (which is quite common for λ-based formalisms), this 'function-and-application perspective' on the syntax and semantics is extended to atomic formulas.

What exactly does this mean?

1. Syntactically, what used to be predicate and relation symbols are treated alike, as one-place function symbols that are combined with other symbols using the application symbol @. n-ary predications are written as n nested applications.
2. This means that the semantics of our (former) predicate and relation symbols has to be given in terms of unary functions if we want to interpret the @-symbol as functional application consistently. In short, we have to re-define models such that they interpret predicate symbols (which are by definition unary) as the characteristic function of the set that we used to assign to the respective symbol. The characteristic function of a set is the function that assigns TRUE to all entities in that set, and FALSE to all other entities. Unary predications can then be stated equivalently as application of such a function to the argument of the predication. On this basis, n-ary predicates are interpreted as complex functions, allowing us to express n-ary predications as a series of nested functional applications. This series has to end with the application of the characteristic function of some set, resulting in a truth value.

Examples: Expressing predicates by functions

This was a bit abstract, so here are two examples. First let's look at the predicate symbol MAN. Here, the situation is easy: A predicate symbol used to be interpreted as the set of things in the extension of the respective predicate in the model under consideration. In our example that's the set of all men, (assuming that we're looking at a model that really interprets the symbol MAN as counterpart of the word 'man'). Now, we will simply use the characteristic function of that same set instead, hence in our example the function that yields TRUE if its argument is a man (and thus would have been in the extension of MAN in our old model), and false otherwise.

But what can we do for relation symbols? How do we give an interpretation in terms of a *unary* function that corresponds in the right way to an *n-ary* relation? The solution is to use functions that yield functions as a result. We can do this in such a way that we finally arrive at the characteristic function of some set. Let us look at the example of the (former) relation symbol LOVE. We used to interpret this symbol as the set of ordered pairs such that the first element loves the second. Instead we will now interpret it as a unary function that takes each entity to the characteristic function of the set of all things that love that entity.

Let's see how we interpret

$$\text{LOVE}(\text{MARY},\text{JOHN})$$

versus

$$(\text{LOVE}@\text{JOHN})@\text{MARY}$$

We shall assume that we are looking at a model where MARY is assigned Mary and JOHN is assigned John. The first formula is true if the pair $< Mary, John >$ is in the set assigned to LOVE by our model. For the second formula, we will proceed in two steps. First we apply the function that our model assigns to the symbol LOVE to John. This yields the characteristic function of the set of 'john-lovers', which we apply in turn to Mary. This final application gives us the result TRUE in case Mary loves John in our model, and FALSE otherwise.

All other n-ary relations are treated analogously to our example: As functions that yield the characteristic function of some set after n-1 applications. Clearly we can characterize situations just as well using this functional way of speaking as we could with our familiar relational approach.

Connection to lambda-calculus

What's the great advantage of all this? As we've mentioned above, our new notation and interpretation fit in well with λ-calculus. And now that we know something about the interpretation of atomic formulas, we can see why this is so. If you've heard about the interpretation of λ-expressions, you will realize that the functional interpretations we've just discussed for our former predicate and relation symbols are constructed exactly along the lines of the semantics for λ-expressions such as $\lambda x.\text{MAN}(x)$ and $\lambda y.\lambda x.\text{LOVE}(x,y)$. In fact for any of our unary function symbols 'written on its own', there's an equivalent λ-expression where the functional character has been made explicit. We say that the function symbol is η-*equivalent* to its explicit λ-counterpart (and the other way round). Strictly speaking, there are always many λ-expressions that are η-equivalent to one function symbol, because (as usual) α-equivalence doesn't make a difference. Here's an example: LOVE and $\lambda y.(\lambda x.(\text{LOVE}@x)@y)$ are η-equivalent, and so are LOVE and $\lambda s.(\lambda r.(\text{LOVE}@r)@s)$, etc.

A simplification

For practical purposes this means that we can use the (shorter) function symbols for common nouns directly in semantic construction, instead of their η-equivalent (long) λ-terms. For example we used to write $\lambda x.\text{MAN}(x)$ (or, lately, $\lambda x.\text{MAN}@x$) as the translation of 'man', and translated the NP 'a man' as:

$$\lambda P.\lambda Q.\exists y.(P@y \land Q@y)@\lambda x.\text{MAN}(x)$$

But now we know the functional semantics of MAN on its own, and so we know that we can apply the determiner to that function directly, to the same effect:

$$\lambda P.\lambda Q.\exists y.(P@y \land Q@y)@\text{MAN}$$

We will make use of this simplification in our implementation of CLLS (see in particular Section 4.3.1).

4

Constraint Solving

In this section we show how we can extract the semantic representations of the vraious readings of a sentence from its constraimt graph, how to actually implement constraints and an enumeration procedure, and how we can use our semantics construction framework to derive constraints from the syntactic analysis of an NL sentence.

From Constraints to Formulae: Now we know how to describe the possible readings of a scope ambiguity by means of a constraint graph. The next thing we need to find out is how we can extract the semantic representations of the readings from the graph. This is going to be the topic of the first part of this section. Later we will show you how to actually implement constraints and an enumeration procedure. Last not least, you will see how we can use our semantics construction framework to derive constraints from the syntactic analysis of an NL sentence.

4.1 Constraint Solving

When we deal with solving underspecified descriptions, the two algorithmic problems that concern us most are the following:

Satisfiability Given a constraint graph, we need to decide whether there is a λ-structure into which it can be embedded.

Enumeration Given a constraint graph, we have to compute all λ-structures into which it can be embedded.

These are the problems adressed in this section. The concept of a *solved form* will be central to the solutions that we develop.

4.1.1 Satisfiability and Enumeration

One Remark!

Before we go into the matter of constraint solving, one remark is due. Below we will use the words "constraint" and "constraint graph" interchangeably. Strictly speaking, constraint graphs are the graphs we have drawn so far, whereas constraints are formulas of a certain simple logic, which we have announced in Section 3.2.9 without defining

it. But both representations can easily be translated into each other, so we'll allow ourselves some sloppy language.

When we deal with solving underspecified descriptions, the two algorithmic problems that concern us most are the following:

Satisfiability Given a constraint graph, we need to decide whether there is a λ-structure into which it can be embedded.

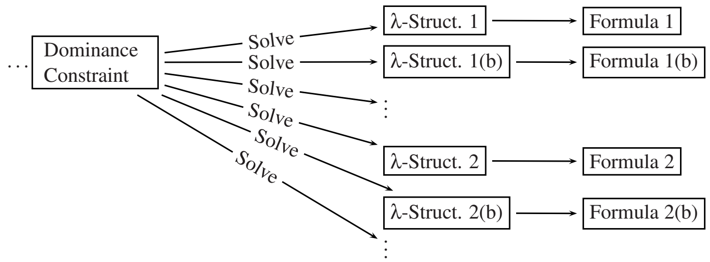
Enumeration Given a constraint graph, we have to compute all λ-structures into which it can be embedded.

It is clear that the first problem is simpler than the second one. Whenever you have an algorithm with which you can enumerate all solutions, you can just stop when you have found the first solution, and say that the constraint is indeed satisfiable. And if it turns out that you just can't find a solution with your enumeration algorithm, you can be sure that it's unsatisfiable.

4.1.2 Solved Forms

Many solutions...

In fact, if you think about the enumeration problem a bit, you'll notice that it is not realistic to enumerate all λ-structures that solve the constraint: In general, there can exist an infinite number of satisfying λ-structures into which the fragments can be embedded while respecting the dominances. However, the differences between most of the solutions are completely irrelevant additions of extra material. The situation looks like this:



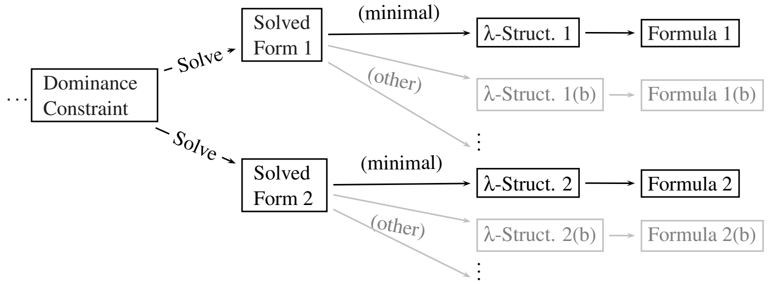
In this example, we might only be interested in the two solutions 1 and 2 while the "variants" 1(b) and 2(b) as well as all other variants with additional material inbetween are pointless to us.

...few solved Forms

So instead of really trying to enumerate *solutions* (i.e. λ-structures), we reformulate the problem to enumerate *solved forms* of the original constraint. Intuitively, solved forms are themselves constraint graphs that each represent a class of solutions that only differ in the addition of extra material. We will define them in a way that they have the following useful properties:

- Every constraint graph has a finite number of solved forms.
- The solutions of all solved forms of a constraint taken together are the same as the solutions of the original constraint.
- It is trivial to enumerate the solutions of a solved form.

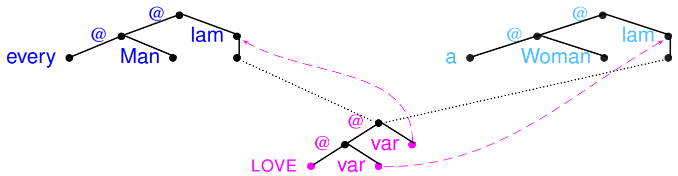
Before we look at an example, let's refine the scheme of our architecture (page 49) once more:



4.1.3 Solved Forms: An Example

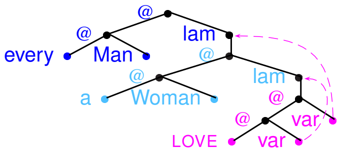
A constraint graph...

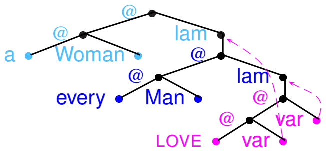
Since the definition of solved given before forms might be a little too much on the abstract side, let's have a look at our running example 'Every man loves a woman' again. Remember that the corresponding original constraint is (where, again, abbreviations like Woman stand for the simple λ-structures).



...two of its solutions...

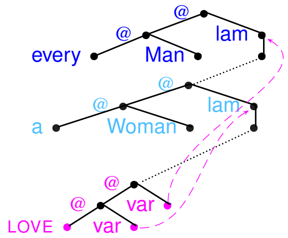
Two of its solutions are the following λ-structures. (In fact these are the only solutions we've been interested in so far. But remember that we can get lots of other solutions by integrating new material):

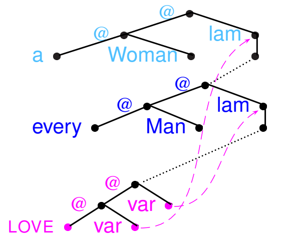




...and its two solved forms.

Now while the constraint graph has an infinite number of solutions, it has precisely two solved forms:





If you look at the solved forms, you'll see that they're very close to the λ-structures – the graphical difference only consists in the dominance edges, which allow the addition of extra material. We can get from the solved forms to the two solutions that we saw above by simply identifying the end points of each dominance edge.

However, it is important to remember that the solved forms are *not* solutions, i.e. λ-structures! They are still constraint graphs. They do have the special property that if you disregard the binding edges, these graphs are *trees*, but they can still contain dominance edges.

4.1.4 Defining Solved Forms

Summing up, we say that a constraint graph is *in* solved form if it has certain properties that guarantee its satisfiability and make it trivial to enumerate its solutions. The solved forms *of* a constraint are constraints in solved form that each represent a class of solutions of the original constraint that have only "irrelevant" differences. Every solution of the original constraint is a solution of one of the solved forms; and vice versa.

Let us now define what a *solved form* is. A constraint graph is in solved form if:

1. It has no cycles that use only solid and dominance edges.
2. It has no node with two incoming edges that are solid or dominance.

You can easily verify that if you disregard the binding edges, every graph in solved form is a tree. As it is forbidden that a node with a node label has an outgoing dominance edge in a constraint graph, you get a tree that consists of the little tree fragments of the original constraint, with dominance edges going from (unlabeled) leaves to roots.

From Solved Forms to Solutions

Now how do we get from a solved form to an actual solution? As we have already said, this step is going to be quite trivial. In general, there are two cases we have to distinguish:

1. If we're lucky, the solved form will not contain any nodes with two outgoing dominance edges anyway. In this case, we can simply identify the endpoints of each dominance edge, and we obtain a minimal λ-structure that satisfies the solved form. As it happens, all of the solved forms we'll get for underspecified semantics will belong to this class.
2. Otherwise, we could do the same operation as in Section 3.2.9: For each node with more than one outgoing dominance edge, we add an arbitrary node label, and make the dominance children real children over solid edges.

As you can see, it's always possible to construct a rather small solution to a solved form very easily. In particular, you know that solved forms are always satisfiable.

4.2 An Algorithm For Solving Constraints

Given a solved form of a constraint it is almost trivial to get to a solution for it. What we don't know yet is how to get to the solved form itself, given an arbitrary constraint. In this section, we'll formulate an algorithm that enumerates the solved forms of arbitrary constraints.

As we have argued before, it is almost trivial to get to a solution from a solved form. What we don't know yet is how to get to the solved form itself, given an arbitrary constraint. In this section, we'll formulate an algorithm that enumerates the solved forms of an arbitrary constraint.

Our overall strategy will be as follows: starting with an arbitrary constraint graph, we'll try to work our way towards a set of constraints in solved form that together have the same solutions as the original constraint. For this, we'll use the algorithm discussed in the following part of this lecture. The set of solved forms will be the output of this algorithm.

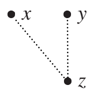
If we like to, we can then proceed to enumerate solutions (or just minimal solutions) of the solved forms after this, in the way described in the last section. But we don't have to.

Eliminate double incoming dominance edges

The main property of solved forms that we try to establish in our algorithm is that there are no nodes with two incoming dominance edges. Thus the central task of our algorithm is eliminating such nodes. We will now look in detail at the three steps that make up the algorithm: Applying the so called *Choice Rule*, *Parent Normalization* and *Redundancy Elimination*.

4.2.1 The Choice Rule

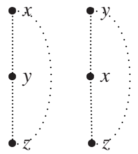
The key insight that we exploit in the algorithm is the following. Suppose you have a node with two incoming dominance edges:



Any solution of this constraint must be a tree (plus binding edges). Since trees don't branch upwards, this means that the only way in which two different nodes can dominate a third one is if one of them, in turn, dominates the other.

Pursuing two alternatives.

Of course we don't know beforehand which of the two has to be the higher node in the solution; in principle, both *choices* can lead to solutions. We can thus formulate the *Choice Rule* as follows: If Z is a node with dominance edges from X to Z and Y to Z, add either the dominance edge from X to Y or the dominance edge from Y to X. Graphically:



Because of the argument we just made, we don't lose solutions in this way: Any solution has either the dominance from X to Y or vice versa.

We will refer to this rule both as the *Choice Rule* (because it chooses either X or Y to dominate the third node) and as the *Distribution Rule* . This second name is motivated from a programming paradigm called *Constraint Programming*, in which case distinctions are referred to as "distribution". The Choice Rule is the only case distinction which we use in our enumeration algorithm.

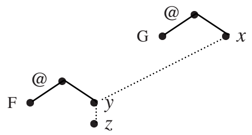
4.2.2 Normalization

Cleaning up

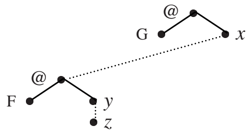
The Choice Rule is the driving force behind the enumeration process: It resolves one node that keeps the constraint from being in solved form by adding additional dominance edges. However, the Choice Rule can't operate on its own. It needs some helpers that clean up after it has done its job. This cleaning work is what we call *normalization* .

Parent Normalization

The first kind of normalization we need to apply is necessary because X and Y above are generally leaves of bigger tree fragments. This means that an application of the Choice Rule gets us into a situation where e.g. Y has an incoming dominance edge and an incoming solid edge – which is not allowed in a solved form:



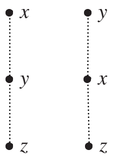
Fortunately, we can resolve this configuration easily. The key observation is that X and Y cannot be mapped to the same node in a solution, as their parents must be different. Thus we can infer that X must dominate not just Y, but the parent of Y:



We can continue with this kind of inference; the sequence of inference steps will stop when we have deduced that X dominates the root of Y's fragment, which now doesn't have an incoming solid edge any more. We call this step *parent normalization* .

Redundant Edges

The other kind of normalization we need removes *redundant* dominance edges. A redundant dominance edge is one which we can remove from a constraint graph without losing information. For example, if we add the edge from X to Y in the Choice Rule, the old dominance edge from X to Z becomes redundant: Even when we remove it, the graph still expresses that there must be a path from X to Y and a path from Y to Z, so there must of course also be a path from X to Z. Here are the solutions of our example constraint without the unnecessary edges:



We call the operation of removing unnecessary dominance edges *redundancy elimination* . Redundancy elimination can be done quite efficiently using a standard graph algorithm called *transitive reduction*.

?- Question!

It's important that we always apply parent normalization before redundancy elimination. Can you tell why?

4.2.3 The Enumeration Algorithm

The Algorithm

We obtain a sound and complete enumeration algorithm for solved forms by putting these steps together in the following way:

1. Apply redundancy elimination and parent normalization as long as possible.
2. If there are still nodes with two incoming dominance edges, pick one and apply the Choice Rule once. Then continue with step 1 for each of the two resulting graphs.
3. Otherwise, the graph either has a cycle or is in solved form.

Checking for Cycles

It is very easy to check whether a graph has a cycle: The standard algorithm for this is depth-first search. This check has to be performed once for each potential solved form. Because the algorithm has eliminated all nodes with more than one incoming edge by this time, we know that every graph that passes this final test is indeed a solved form.

Efficiency Issues

Each component of the enumeration algorithm is quite efficient, and even though it is very simple, the complete algorithm is one of the more efficient algorithms for enumerating solutions of underspecified descriptions. But because the Choice Rule has to make an uninformed choice, and it's quite possible that one of the two results is unsatisfiable, there is a possibility that our algorithm spends a lot of time failing, even if the input graph has very few solved forms. What's worse is that, if we're unlucky, we might explore the branches of the search tree that lead to unsatisfiable constraints first, and it might take a long time before we find even the *first* solution. This means that although the enumeration algorithm gives us a satisfiability test, it's by no means a very efficient one.

It's possible to write a special satisfiability test that runs very efficiently (in linear time); but this algorithm employs rather advanced graph algorithms that we can't discuss here. We can use this satisfiability algorithm in turn to guide the enumeration: Whenever we apply Choice, we can check both results for satisfiability, and if one of them is unsatisfiable, we don't need to spend any time at all on exploring the search tree below this constraint. Our algorithm above would have continued a fruitless computation on the unsatisfiable constraint, and only discovered the unsatisfiability in the very end. In effect, such an early satisfiability test can dramatically speed up the enumeration process.

4.3 Constraint Solving in Prolog

After that much of theory we come to our implementation of constraint solving in Prolog. First, we will see how our constraints are represented in Prolog. Then we will look at how they are brought into solved form. As we have said above, it is quite straightforward to convert solved forms into solutions and then into our good old λ-expressions. We leave it as an exercise to you to work through that part of our implementation.

4.3.1 Prolog Representation of Constraint Graphs

In the rest of this chapter, we will explain how to implement the constraint solver in Prolog. First of all, let's have a look at how we represent a constraint graph in Prolog. We represent such a graph as a collection `usr(Ns,LCs,DCs,BCs)` of four lists. These lists contain all ingredients of constraint graphs: nodes (`Ns`), labeling constraints (solid edges: `LCs`), dominance constraints (dotted edges: `DCs`), and binding constraints (dashed arrows: `BCs`). `usr` stands for *underspecified representation*. In other words, we represent the graph by specifying its nodes and its various types of edges. Incidentally, this syntax is extremely similar to the notation as logical formulas (constraints) that we have mentioned above.

Nodes and Labelings

Nodes are simply Prolog atoms: Each node gets a unique name. The labelings are terms which are composed with the Prolog inbuilt operator `:`. For example, `x0:(x2@x1)` means that the node `x0` is labeled `x2@x1`. Note that this labeling constraint tells you two things at once, namely that:

1. `x0` has the label `@`.
2. `x0` has two daughters over solid edges: `x1` and `x2`.

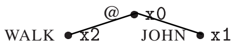
Dominances and Bindings

The Prolog notation for a dominance edge is `dom(x0,x1)`. Finally, a binding edge stating the fact that the variable for which the (var-)node `x1` stands for is bound by the (lam-)node `x0` is represented as `bind(x1,x0)`.

Here is a sample constraint for 'John walks':

```
usr([x0,x1,x2],[x0:(x2@x1),x1:john,x2:walk],[],[]).
```

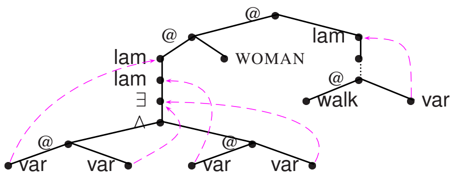
In the more familiar tree representation:



If you experiment with the implementation later, you will notice that the constraint graphs soon become large and somewhat unreadable. For example, here's the representation for 'A woman walks':

```
usr([x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17],
    [x0:(x17@x1),x1:var,x2:(x3@x4),x3:(x6@x16),x4:lambda(x5),
     x6:lambda(x7),x7:lambda(x8),x8:exists(x9),x9:(x10& x11),
     x10:(x12@x13),x11:(x14@x15),x12:var,x13:var,x14:var,x15:var,
     x16:woman,x17:walk],
    [dom(x5,x0)],
    [bind(x1,x4),bind(x12,x6),bind(x13,x8),bind(x14,x7),bind(x15,x8)]).
```

But if you look closely, you'll notice that this long term is really nothing but a representation of the following graph, constructed in a way that allows for better use in a program:



And if you look closely at this constraint graph, you'll notice another thing: We've used the predicate symbol WOMAN instead of the abbreviation Woman. But didn't we

say that Woman abbreviates $\lambda x.\text{WOMAN}@x$? In fact, for all common nouns (that is, also for 'man', 'siamese cat', etc.), we will directly use the predicate symbol instead of a λ-term in our implementation. This will make the semantic macro for common nouns a bit easier. Have a look at the sidetrack in Section 3.2.10 to see why this is possible.

4.3.2 Solve

See file `solveConstraint.pl`.

The main predicate of our constraint solver is `solve/2` (to be found in `solveConstraint.pl`). This predicate is a straightforward implementation of the enumeration algorithm from Section 4.2.3. It takes a constraint graph as its first argument, and returns the list of its solved forms in the second argument. Our `solve/2`-predicate calls some low-level predicates that implement normalization and distribution. We will define these predicates later.

The predicate `solve/2` itself consists of three clauses. Its first clause handles the case of a constraint graph that still contain one or more nodes with two incoming dominance edges. It normalizes the input graph, and then calls `distribute/3` to select one such node and compute the two constraints with the additional dominance edge (hence it corresponds to the Choice Rule discussed above (page 63)). Then it calls itself recursively, once for each of the two alternatives, and appends the two lists of solved forms thus obtained.

```
solve(Usr,Solutions) :-
        normalize(Usr,NormalUsr),
        distribute(NormalUsr,Dist1,Dist2),
        solve(Dist1,Solutions1),
        solve(Dist2,Solutions2),
        append(Solutions1,Solutions2,Solutions).
```

See file `cllsLib.pl`.

The second clause takes care of the base case, a constraint graph in which no node has two incoming dominance edges. It is used only if the call to `distribute/3` in the first clause fails (as we shall see shortly, `normalize/2` can never fail). Such a graph is a solved form iff it has no cycles, which we check with the predicate `hasCycle/1` (from `cllsLib.pl`).

```
solve(Usr,[NormalUsr]) :-
        normalize(Usr,NormalUsr),
        \+ hasCycle(NormalUsr).
```

If the constraint graph does have a cycle, the second clause will fail as well. In this case, the following final clause applies. It simply returns an empty list of solved forms. We can't just let it fail because in that case, the entire original call of `solve/2` would fail. This is wrong in general because it's very possible that the original constraint has solutions, but we have made some wrong choices along the way that have made our current graph unsatisfiable.

```
solve(_Usr,[]).
```

?- Question!

Look at the implementation of `hasCycle/1` in `cllsLib.pl` and explain in your own words how this test works.

4.3.3 Distribute

***See file* `solveConstraint.pl`.**

The predicate `distribute/3` (from `solveConstraint.pl`) implements the Distribution Rule (or Choice Rule (page 63)). It takes a constraint graph as its first argument, picks a node `Z` with two incoming dominance edges (from `X` and `Y`), and applies the Distribution Rule to them. It returns the two constraint graphs that are obtained by adding a dominance edge between `X` and `Y` in either direction. The predicate fails if the input constraint doesn't contain such a node `Z` with two incoming dominance edges.

```
distribute(usr(Ns,LCs,DCs,BCs),usr(Ns,LCs,[dom(X,Y)|DCs],BCs),usr(Ns,LCs,[d
        member(dom(X,Z),DCs),
        member(dom(Y,Z),DCs),
        X \== Y.
```

The two calls of `member/2` check that there are in fact two dominance edges `dom(X,Z)` and `dom(Y,Z)` in the list of dominance edges `DCs` of the incoming constraint. By adding `X \== Y`, we make sure that X and Y are different nodes. Otherwise, it would be possible to add dominances like `dom(X,X)`.

4.3.4 (Parent) Normalization

***See file* `solveConstraint.pl`.**

What remains to be implemented now is the normalization of constraints (see Section 4.2.2). It is done in two steps: First, parent normalization is performed, and secondly redundant dominance edges are removed. The predicate `normalize/2` (in `solveConstraint.pl`) calls the respective predicates. It accepts a constraint graph as input, and returns the normalized graph.

```
normalize(Usr,Normal) :-
        liftDominanceConstraints(Usr,Lifted),
        elimRedundancy(Lifted,Normal).
```

Parent Normalization

The predicate `liftDominanceConstraints/2` (from `solveConstraint.pl`) recursively goes through all dominance edges. If there is a dominance edge `dom(X,Y)` and the node Y is a child of the node Z via a solid edge, the dominance edge is lifted and becomes `dom(X,Z)`. Here is the code:

```
liftDominanceConstraints(Usr,Lifted) :-
        Usr = usr(Ns,LCs,DCs,BCs),
        mySelect(dom(X,Y),DCs,RestDCs),
        idom(Z,Y,Usr),!,
        liftDominanceConstraints(usr(Ns,LCs,[dom(X,Z)|RestDCs],BCs),Lifted)

liftDominanceConstraints(Usr,Usr).
```


***See file* `cllsLib.pl`.**

The predicate `mySelect/3` removes the dominance edge `dom(X,Y)` from `DCs` and the list `RestDCs` now contains the remaining dominance edges. Then, `idom/3` (see `cllsLib.pl`) succeeds iff the node Z *immediately dominates* the node Y, i.e. there is a solid edge from Z to Y. If this is the case, the parent normalization is continued with a constraint that is like the input constraint. Except for the list of dominance edges which now contains all but the removed ("lifted") edge, plus the new edge `dom(X,Z)`. In other words, `dom(X,Y)` has become `dom(X,Z)`.

An an exercise you can try to reimplement this predicate using a simple member check instead of `select`.

4.3.5 Redundancy Elimination

The second normalization step described in Section 4.2.2 is the elimination of redundant dominance edges. The implementation we present here doesn't use the transitive reduction algorithm we mentioned earlier, but simply goes through each dominance edge in the graph and checks whether the lower end can still be reached from the upper end if the dominance edge is removed. This algorithm is slower than transitive reduction, but much simpler.

```
elimRedundancy(usr(Ns,LCs,DCs,BCs),Irredundant) :-
        mySelect(dom(X,Y),DCs,DCsRest),
        reachable(Y,X,usr(Ns,LCs,DCsRest,BCs)),!,
        elimRedundancy(usr(Ns,LCs,DCsRest,BCs),Irredundant).

elimRedundancy(Usr,Usr).
```

***See file* `cllsLib.pl`.**

First, a dominance edge `dom(X,Y)` is removed from `DCs`. Second, it is checked whether the node Y is still reachable from the node X (the predicate `reachable/3` can be found in `cllsLib.pl`). If so, the removed dominance edge was redundant. In this case, the redundancy elimination continues with a constraint containing all remaining dominance edges (`DCsRest`). Note that in this case, the cut prevents any further backtracking.

But what happens if a dominance edge that is necessary for establishing some reachability in the graph is deleted? Well, in this case `reachable/3` fails and backtracking selects (and removes) another dominance edge instead, checking the reachability again afterwards. Eventually all redundant edges will have been removed, at which point all calls to `reachable/3` will fail, and the second clause is used to return the irredundant graph.

?- Question!

Look at the declaration of `reachable/3` in `cllsLib.pl` and explain in your own words, how reachability is checked there.

4.4 Semantics Construction for Underspecified Semantics

In this section we show you to you can derive underspecified representations from syntactic analyses. The semantics construction algorithm we present here uses the syntax/semantics framework laid out in the earlier chapters of this course.

To conclude our chapter on underspecification, we will show you how you can derive underspecified representations from syntactic analyses. The semantics construction algorithm we present here uses the syntax/semantics framework laid out in the earlier chapters of this course. The relevant changes are:

1. Devising the semantic macros that provide the meanings of words on the lexical level.
2. Giving the `combine`-rules that combine smaller constraint graphs for subphrases to larger ones representing the meanings of the larger phrases.

As you will see, the division of labour between the two types of rules is different than what it used to be. Our implementation of Montague semantics was highly *lexicalized*: The lexical entries were relatively rich, and the combine-rules just told us what was the functor and what was the argument in a functional application. Here it's going to be the other way round: Most lexical entries are going to be very simple, and the combination rules will do most of the work.

4.4.1 The Semantic Macros

***See file* `clls.pl`.**

Most of the semantic macros we need (to be found in `clls.pl`) are very simple constraint graphs representing a single labeled node. Let's look at the simple graphs first.

4.4.1.1 The Simple Macros

***See file* `clls.pl`.**

For example, the macro for proper names looks like this:

`pnSem(Symbol,usr([Root],[Root:Symbol],[],[])).`

That is, the meaning of the word 'John' (`Symbol=john`) is a node labeled JOHN:

JOHN •

The macros for nouns, transitive and intransitive verbs, and prepositions are similar

```
nounSem(Symbol,usr([Root],[Root:Symbol],[],[])).

tvSem(Symbol,usr([Root],[Root:Symbol],[],[])).

ivSem(Symbol,usr([Root],[Root:Symbol],[],[])).

prepSem(Symbol,usr([Root],[Root:Symbol],[],[])).
```


Prolog Variables Again

Note that the general semantics construction framework requires us to use *variables* for the nodes – called `Root` in the macros mentioned above. On the other hand, in Section 4.3.1 we said that we want to represent nodes as atoms. This is the same trick we saw before when doing semantics construction with the λ-calculus. Again, we will undo this cheat by "atomizing" all Prolog variables (nodes) after the semantics construction is finished.

4.4.1.2 Macros for the Determiners

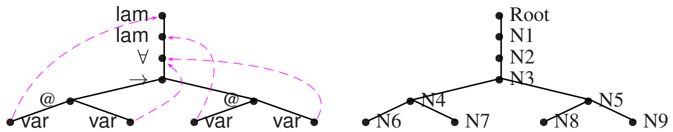
***See file* `clls.pl`.**

Determiners get a slightly more complex semantics. Here are the semantic macros for 'every' (with the label `uni`) and 'a' (with label `indef`):

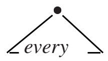
```
detSem(uni,usr([Root,N1,N2,N3,N4,N5,N6,N7,N8,N9],
          [Root:lambda(N1),N1:lambda(N2),N2:forall(N3),N3:(N4 > N5),
           N4:(N6@N7),N5:(N8@N9),N6:var,N7:var,N8:var,N9:var],
          [],
          [bind(N6,Root),bind(N7,N2),bind(N8,N1),bind(N9,N2)])).

detSem(indef,usr([Root,N1,N2,N3,N4,N5,N6,N7,N8,N9],
          [Root:lambda(N1),N1:lambda(N2),N2:exists(N3),N3:(N4 & N5),
           N4:(N6@N7),N5:(N8@N9),N6:var,N7:var,N8:var,N9:var],
          [],
          [bind(N6,Root),bind(N7,N2),bind(N8,N1),bind(N9,N2)])).
```

These structures look quite intimidating, but if you look at the corresponding constraint graphs, you'll see that e.g. the macro for 'every' is nothing but a description of the term $\lambda P\lambda Q.\forall x(P@x \to Q@x)$. To make it easier to check that the semantic macro `detSem(uni,...)` given above really corresponds to this tree representation, we have decorated the tree backbone with the respective PROLOG-variables (see below on the right).



Just in order to make life easier for us from now on, we'll abbreviate the graphs for determiners as follows:



We can do this safely because the root of the subgraph is the only node we'll have to refer to below.

4.4.2 The `combine`-rules

***See file* `clls.pl`.**

Now let's have a look at the `combine`-rules that parallel the syntax rules and combine the constraint graphs for subphrases to the constraint graph for a larger phrase.

The First Node is the Root

The general principle is that each constituent of the sentence is associated with a part of the final constraint graph. We're going to maintain the invariant that the first node in the node list of such a partial graph is the *root* of this subgraph. The graphs for different constituents are combined by adding constraints that relate their roots.

One simple but central predicate we make use of here is `mergeUSR/2` (see `cllsLib.pl`). It combines constraint graphs by merging their respective lists; the root of the merged graph will be the root of the first (leftmost) graph that was given to `mergeUSR/2`.

Most of the `combine`-rules are trivial. They simply lift the semantics of some syntactic category to that of a higher category without adding any further material. As an example, see the rule `NP` $\rightarrow$ `PN`:

```
combine(np1:A,[pn:A]).
```

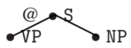
We present the more complex `combine`-rules below, except for one (namely `N` $\rightarrow$ `N Adj`). We leave the formulation of this rule to you as an exercise.

4.4.2.1 `S` $\rightarrow$ `NP VP`

Let's first look at the rule that builds sentences out of an NP and a VP. (Think of a sentence like 'John walks' for now; we'll get to quantifiers later.) The combine rule for this syntax rule is as follows:

```
combine(s1:S,[np2:NP,vp2:VP]):-
          NP = usr([NPRoot|_],_,_,_),
          VP = usr([VPRoot|_],_,_,_),
          NewUsr = usr([Root],[Root:(VPRoot@NPRoot)],[],[]),
          mergeUSR(merge(NewUsr,merge(NP,VP)),S).
```

Again, it may be helpful to look at the graph representation of this rule.



As you can see, the new constraint graph describes λ-structures in which the VP meaning is applied to the NP meaning. This is basically like our very first naive analysis of how NPs and VPs are combined semantically. It still works because the trick that we developed in order to get a uniform treatment of the NP semantics in λ-calculus is now compiled into the combine rule for NPs, which we'll deal with in a minute. For now, just observe that we combine the verb phrase and the noun phrase straightforwardly and according to our intuitions: The verb phrase is the functor and the noun phrase its argument. The results we get (after β-reduction) are the same as with Montague Semantics.

More technically, the clause of `combine/2` first extracts the roots of the constraint graphs for the two constituents, `NPRoot` and `VPRoot`. It then introduces a new node and a new labeling constraint, and merges it with the subgraphs of the constituents.

For the example we suggested above, "John walks", the following happens. First, the semantic macros provide the semantics on the lexical level:

John	• JOHN	`usr([x1],[x1:john],[],[])`
walks	• WALK	`usr([x2],[x2:walk],[],[])`

And here is the result of the `combine`-rule:

John walks

`usr([x0,x1,x2],[x0:(x2@x1),x1:john,x2:walk],[],[])`

This graph describes the λ-structure representing the λ-term $walk@john$, or written in a more familiar way, $walk(john)$.

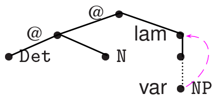
4.4.2.2 `NP` $\rightarrow$ `Det N`

Now let's see how we can combine determiners and nouns into NPs. This is a slightly complex but very interesting rule, as it takes care both of the correct binding of a variable bound by a quantifier, and of the introduction of the dominance edges we need in order to represent scope ambiguities.

The rule looks as follows:

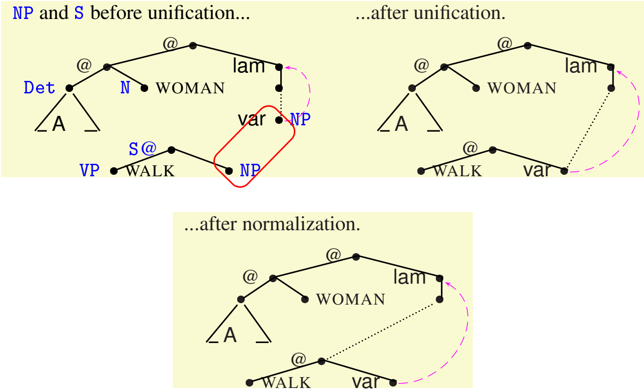
```
combine(np1:NP,[det:DET,n2:N]) :-
        DET = usr([DETRoot|_],_,_,_),
        N = usr([NRoot|_],_,_,_),
        NewUsr = usr(
        [Root,N1,N2,N3,N4],
        [N1:(N2@N3),N2:(DETRoot@NRoot),N3:lambda(N4),Root:var],
        [dom(N4,Root)],
        [bind(Root,N3)]),
        mergeUSR(merge(NewUsr,merge(DET,N)),NP).
```

Again, this becomes more readable when written as a constraint graph:



The root of an NP is a var-node.

The root of the constraint graph for the entire NP is the node `Root`, i.e. the node which is labeled with `var`. Although this may seem a bit counterintuitive, it is extremely useful considering how the NP will later combine with a VP (see Section 4.4.2.1). The VP semantics will be applied to the root of the NP graph. That is, the verb semantics will be applied to a variable that is bound by the quantifier – exactly what we want! Consider the constraint for 'a woman walks' given below. We show the last step of semantics construction and one parent normalization step.



Look at the leftmost graph. On top you see the representation of an NP as discussed above. Here, the representation of the determiner A (abbreviated in the figure) and the noun representation WOMAN have already been integrated (by unification). Below you can see the representation of an S where the VP WALK has already been found. The next (and last) step in semantics construction is the integration of the NP into the S, i.e. the unification of the root node of the NP and the NP node of the S.

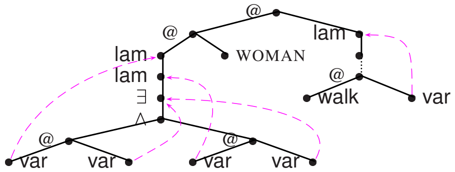
The result of this unification can be seen in the middle graph. This constraint graph is the underspecified representation for the sentence 'a woman walks'. With the help of one simple parent normalization step, we end up with the solved form of this graph which can be seen on the right. A minimal solution hereof is the following λ-expression:

$$\mathsf{A}@\text{WOMAN}@(\lambda x.\text{WALK}@x)$$

which probably looks much more familiar to you. Now we can now fully realize just how convenient binding constraints are. We can simply relate the variable and its binder within a single semantics construction rule; they are connected forever with an unbreakable link, and we don't have to think at all about variable naming. We could unify both NP nodes in the example above and got *walk@var* (we christened the variable represented by *var x* in the λ-expression given above). If the NP is a proper name as in 'John walks', the same mechanism unifies the NP node of the sentence with JOHN as we have seen in Section 4.4.2.1. Finally, the remaining dominance edge from in the solved form is one of those dominance edges that can lead to the representation of a scope ambiguity in other configurations. Next, you can see this example with fully expanded determiner.

4.4.2.3 Example: 'A woman walks'

Here you can see the example from above with fully expanded determiner.



Make sure that the λ-expression corresponding to this graph is reducable to a normal FO formula.

4.4.2.4 VP → TV NP, PP → PREP NP, and N → N PP

The `combine`-rules for VP → TV NP and PP → PREP NP work exactly like the rule S → NP VP (see Section 4.4.2.1). Again, their function is simply to introduce an application:

```
combine(v1:V,[tv:TV,np2:NP]) :-
        TV = usr([TVRoot|_],_,_,_),
        NP = usr([NPRoot|_],_,_,_),
        NewUsr = usr([Root],[Root:(TVRoot@NPRoot)],[],[]),
        mergeUSR(merge(NewUsr,merge(TV,NP)),V).

combine(pp:PP,[prep:Prep,np2:NP]) :-
        Prep = usr([PrepRoot|_],_,_,_),
        NP = usr([NPRoot|_],_,_,_),
        NewUsr = usr([Root],[Root:(PrepRoot@NPRoot)],[],[]),
        mergeUSR(merge(NewUsr,merge(Prep,NP)),PP).
```

Finally, here is the rule that combines a Noun and a PP to form an N (think of phrases like 'therapist with a siamese cat'):

```
combine(n1:N,[noun:Noun,pp:PP]) :-
        Noun = usr([NounRoot|_],_,_,_),
        PP = usr([PPRoot|_],_,_,_),
        NewUsr = usr(
        [Root,N1,N2,N3,N4,N5,N6],
        [Root:lambda(N1),N1:(N2 & N3),N2:(NounRoot@N4),N4:var,N5:(PPRoot@N6
        [dom(N3,N5)],
        [bind(N4,Root),bind(N6,Root)]),
        mergeUSR(merge(NewUsr,merge(Noun,PP)),N).
```

?- Question!

Can you see what this rule does? Compare it to the local macros for the determiners (see Section 4.4.1.2)!

4.5 Running CLLS

This section introduces the driver predicate `clls/0` for CLLS-based semantic construction, and contains links to all files needed for running the program.

***See file* `clls.pl`.**

Now, it is time to present the driver predicate `clls`. Here it is:

```
clls :-
        readLine(Sentence),
        parse(Sentence,UsSem),
        resetVars,vars2atoms(UsSem),
%       printRepresentations([UsSem]),
        solve(UsSem,Sems),
%       printRepresentations(Sems),
        usr2LambdaList(Sems,LambdaTerms),
%       printRepresentations(LambdaTerms),
        betaConvertList(LambdaTerms,Converted),
        printRepresentations(Converted).
```

You have already seen the first three calls triggering the semantics construction and instantiating the Prolog variables in Section 2.5. Once we have constructed the constraint, `solve/2` computes all its solved forms. Finally, `usr2LambdaList/2` translates the list of solved forms to a list of "traditional" λ-terms. All that is left to do is to β-convert these terms.

Check it out!

Here, you can see the five readings for ‘Every owner of a siamese cat loves a therapist’: `clls(silent,[every,owner,of,a,siamese,cat,loves,a,therapist]).`

You may uncomment any of the calls of `printRepresentations/1` if you wish to inspect the constraint graphs more closely. For the above example, it will look like this: `clls(verbose,[every,owner,of,a,siamese,cat,loves,a,therapist]).`

Code Summary

See file `clls.pl`.	Driver, `combine`-rules, semantic Macros.
See file `solveConstraint.pl`.	Solving: normalization and distribution.
See file `cllsLib.pl`.	Working with USRs, tree predicates, translation of solved form
See file `englishGrammar.pl`.	The DCG-rules and the lexicon (using module See file `engli`
See file `betaConversion.pl`.	β-conversion.
See file `comsemLib.pl`.	Auxiliaries.
See file `comsemOperators.pl`.	Operator definitions.
See file `signature.pl`.	Generating new variables
See file `readLine.pl`.	Reading the input from stdin.

5

Inference in Computational Semantics

Up to now we have seen various ways to construct logical formulae as meaning representations for sentences. But we don't yet know what to do further with such formulae. We will now learn how to do useful work with such meaning representations.

If we utter a sentence, we transport information. One way of exploiting this information is to find out what follows from the sentence. The parallel task on the level of meaning representations is that of *inference* from the formula for that sentence. Knowing what follows from a sentence is an indispensable ingredient of understanding it. Correspondingly, finding out what can be inferred from the formula constructed for a sentence is a very important task in computational semantics. Here are some of the reasons why this is so:

- Often, we can only fully understand a sentence by inferring from it (together with our background knowledge). For example if we ask someone whether he has already listened to the latest record of Carla Bley, he may answer 'Oh, I hate Jazz!'. To understand this as an answer to our question, we have to infer that he in fact has *not* listened to the record (maybe due to his musical half-heartedness).
- Inference from a sentence may be necessary to react properly to it, e.g. to answer a question.
- Already in the process of meaning construction itself, inference may help us reduce the number of readings that can be constructed. This may greatly reduce the load for subsequent processing stages.

In this chapter, we will develop a method to get a grip on the notion of *logical consequence* operationally: We will see how we can use syntactic calculi to *compute* what follows from a formula. But in order to understand this, we first have to repeat some of the basic semantic concepts of first-order logic.

5.1 Basic Semantic Concepts

This section discusses the most important aspects of the semantics of first-order logic. The two central concepts introduced are those of *first-order model* and *truth in a model*.

Intuitively, we perceive of *first-order formulas* as *descriptions* of certain *situations*. First order models correspond to such situations, in which given descriptions may be true or false.

5.1.1 Models

The task of logical semantics is to define how formulas are evaluated in models. In general terms, the purpose of the evaluation process is to tell us whether a description is true or false in a situation.

So what is a model? Actually, what we've just said already pretty much contains the answer to this: A model is like a situation - and a situation is a *semantic* entity, providing us with a certain amount of things we can talk about. Thus, a model should give us two pieces of information. First, it should tell us what kind of collection of entities we can talk about. This is the task of the so called *domain* , or D for short. Secondly, a model should give us appropriate semantic entities, built from the items in D, for the symbols in our language. The function carrying out this task is called the *interpretation function* .

What is a Model?

In set theoretic terms, a model $\mathbf{M}$ thus is an ordered pair (D,F) composed of a domain D and an interpretation function F specifying semantic values in D.

However this definition hides one problem: Intuitively it doesn't make much sense to ask whether or not an arbitrary description is true in an arbitrary situation. Some descriptions and situations simply don't belong together. The same is true for the relation between formulas and models. The model used to evaluate a formula has to be a model *for* that formula (or, more precisely, for the language that formula is taken from). If we examine a formula $\textsc{love}(\textsc{john},\textsc{mary})$, while being provided with a model recording information only about the symbols HATE, ANNA and PETER, then it makes no sense at all to evaluate this particular formula in that particular model. The element connecting a formula with the right models for it is the *vocabulary* (or a *signature*) defining the language of that formula (see Section 1.2.1). So if we want to evaluate a formula in a model, we have to make sure that the model is a model for the vocabulary of the language that our formula belongs to.

We say that a given model D,F is a *model for a vocabulary* V if the domain of the interpretation function F consists of the symbols specified in V. So, according to our previous considerations, F should assign appropriate semantic entities (built from the items in D) to the symbols of V.

But what are *appropriate* semantic values? Given the arity information in V, there's no mystery here. Since constants are names, each constant should be interpreted as *an element of* D, the domain of the model. That is, for each constant symbol c in the vocabulary, $F(c) \in D$. And since n-place relation symbols denote n-place relations, each n-place relation symbol R should be interpreted as an n-place relation on D. That is, $F(R)$ should be a set of *n-tuples of elements of* D.

5.1.2 An Example Model

Let's look at an example. First, here's our vocabulary from (page 2) again:

$(\{\text{MARY}, \text{JOHN}, \text{ANNA}, \text{PETER}\}, \{(\text{LOVE}, 2), (\text{THERAPIST}, 1), (\text{MORON}, 1)\})$

We will now build a model for this vocabulary. Let D be $\{d_1, d_2, d_3, d_4\}$. This set, consisting of four items, is the domain of our little model.

Next, we must specify an interpretation function F. Here's one possibility:

$$\begin{aligned}
&F(\text{MARY}) = d_1\\
&F(\text{ANNA}) = d_2\\
&F(\text{JOHN}) = d_3\\
&F(\text{PETER}) = d_4\\
&F(\text{THERAPIST}) = \{d_1, d_3\}\\
&F(\text{MORON}) = \{d_2, d_4\}\\
&F(\text{LOVE}) = \{(d_4, d_2), (d_3, d_1)\}
\end{aligned}$$

Note that every symbol in the vocabulary neatly corresponds to an appropriate semantic entity:

- The four names correspond to individuals.
- The two arity-1 symbols correspond to subsets of D (that is, properties, or 1-place relations on D).
- The arity-2 symbol corresponds to a 2-place relation on D.

Intuitively, in this model d_1 is called Mary, d_2 is called Anna, d_3 is called John and d_4 is called Peter. Both Anna and Peter are morons, while both John and Mary are therapists. Peter loves Anna and John loves Mary. But for example we also know that sadly, Anna does not love Peter and Mary does not love John.

Given a model of appropriate vocabulary, a formula such as $\forall x\text{MORON}(x)$ is either true or false in that model. To put it more formally, there is a relation called *truth* which holds, or does not hold, between sentences (i.e. formulas without free variables, see Section 1.2.4) and models of the same vocabulary. Now, how to verify if a given sentence is true in a given model is obvious in most cases (for example, in order to check the truth of $\forall x\text{MORON}(x)$ we simply need to check if every individual in the model is a moron). What is not so clear is how to give a precise definition of this relation for arbitrary sentences. This is going to be the problem that the rest of this section deals with.

5.1.3 Satisfaction, Assignments

We cannot give a *direct* inductive definition of truth. Truth is a relation that holds between *sentences* and models. But the matrix of a quantified sentence typically won't be a sentence. For example, $\forall x\text{MORON}(x)$ is a sentence, but its matrix $\text{MORON}(x)$ is not. Thus an inductive truth definition given solely in terms of sentences couldn't explain why $\forall x\text{MORON}(x)$ would be true in a model, for there are no sentential subformulae for such a definition to refer to.

An indirect Approach

So instead, we're going to proceed indirectly by defining a three place relation-called *satisfaction*-which holds between a formula, a model, and an *assignment of values to variables*. Given a model $\mathbf{M} = (D, F)$, an assignment of values in $\mathbf{M}$ to variables (or more simply, an *assignment* in $\mathbf{M}$) is a function g from the set of variables to D. Assignments are a technical aid that tells us what free variables stand for. By making use of assignment functions, we can inductively interpret *arbitrary* formulae in a natural way, which will make it possible for us to define the concept of truth for *sentences*.

5.1.4 Interpretations and Variant Assignments

Let's suppose we've fixed our vocabulary. (Note: Whenever we talk of a model $\mathbf{M}$ from now on, we mean a model of this vocabulary, and whenever we talk of formulae, we mean the formulae built from the symbols in that vocabulary.) We now give two further technical definitions which will enable us to state the satisfaction definition in a concise manner.

Interpretations

First, let $\mathbf{M} = (D, F)$ be a model, let g be an assignment of values to variables in $\mathbf{M}$, and let τ be a term. The *interpretation* of τ with respect to $\mathbf{M}$ and g is $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a variable. We denote the interpretation of τ by $I_F^g(\tau)$.

Variant Assignments

Another concept we need is that of a *variant* of an assignment of values to variables. So, let g be an assignment of values to variables in some model, and let x be a variable. If g' is an assignment of values to variables in the same model, and for all variables y other than x, $g'(y) = g(y)$ then we say that g' is an *x-variant* of g. Variant assignments are the technical tool that allows us to try out new values for a given variable (say x) while keeping the values assigned to all other variables the same.

5.1.5 The Satisfaction Definition

Having established this, we now are ready to define satisfaction. Let φ be a formula, let $\mathbf{M} = (D, F)$ be a model, and let g be an assignment of values in $\mathbf{M}$ to variables. Then the relation $\mathbf{M}, g \models \varphi$ (φ is satisfied in $\mathbf{M}$ with respect to the assignment of values to variables g) is defined inductively as follows:

$$\begin{array}{lll}
\mathbf{M}, g \models R(\tau_1, \cdots, \tau_n) & \textit{iff} & (I_F^g(\tau_1), \cdots, I_F^g(\tau_n)) \in F(R)\\
\mathbf{M}, g \models \neg\varphi & \textit{iff} & \text{not } \mathbf{M}, g \models \varphi\\
\mathbf{M}, g \models \varphi \wedge \psi & \textit{iff} & \mathbf{M}, g \models \varphi \text{ and } \mathbf{M}, g \models \psi\\
\mathbf{M}, g \models \varphi \vee \psi & \textit{iff} & \mathbf{M}, g \models \varphi \text{ or } \mathbf{M}, g \models \psi\\
\mathbf{M}, g \models \varphi \rightarrow \psi & \textit{iff} & \text{not } \mathbf{M}, g \models \varphi \text{ or } \mathbf{M}, g \models \psi\\
\mathbf{M}, g \models \exists x\varphi & \textit{iff} & \mathbf{M}, g' \models \varphi, \text{for some x-variant } g' \text{ of } g\\
\mathbf{M}, g \models \forall x\varphi & \textit{iff} & \mathbf{M}, g' \models \varphi, \text{for all x-variants } g' \text{ of } g
\end{array}$$

(Here 'iff' is shorthand for 'if and only if'.) Note the crucial - and indeed, intuitive - role played by the x-variants in the clauses for the quantifiers. For example, what the clause for the existential quantifier boils down to is this: $\exists x\varphi$ is satisfied in a given model, with respect to an assignment g, if and only if there is *some* x-variant g' of g that satisfies φ in the model. That is, we have to try to find *some* value for x that satisfies φ in the model, while keeping the assignments to all other variables the same.

5.1.6 Truth in a Model

We can now define what it means for a *sentence* to be *true in a model* :

A sentence φ is true in a model **M** if and only if for *any* assignment g of values to variables in **M**, we have that:

$$\mathbf{M}, g \models \varphi$$

If φ is true in **M** we write:

$$\mathbf{M} \models \varphi$$

This elegant definition of truth beautifully mirrors the special, self-contained nature of sentences. It's based on the following observation: *It doesn't matter at all which variable assignment is used to compute the satisfaction of sentences*. Sentences contain no free variables, so the only free variables we will encounter when evaluating one are those produced during the process of evaluating its quantified subformulae (if it has any). But the satisfaction definition tells us what to do with such free variables, namely, to try out variants of the current assignment and see whether they satisfy the matrix or not. In short, you may start with whatever assignment you like; the result will be the same. It is reasonably straightforward to make this informal argument precise, and the reader is encouraged to do so.

5.1.7 Validities

Our main topic in this chapter is *logical inference*. Given the semantic concepts just introduced, we're now in a position to state precisely what we mean by this. We will do so in two separate steps. First we'll establish what *valid formulae* (or more simply, *validities*) are. Then we'll define the concept of a *valid argument* (or *valid inference*).

Valid Formulae

A *valid formula* is a formula that is satisfied in *all* models (of the appropriate vocabulary) given *any* variable assignment. That is, if φ is a valid formula, it is impossible to find a situation in which φ would not be satisfied. We indicate that a formula φ is valid by writing $\models \varphi$.

For example:

$$\models (\textsc{moron}(x) \vee \neg\textsc{moron}(\textsc{x}))$$

In any model, given any variable assignment, one of the two disjuncts must be true (incidentally in the case at hand, only one can be true), and hence the whole formula will be satisfied too.

Valid Sentences

Note that for sentences the definition of validity can be rephrased as follows, without reference to assignments: A *valid sentence* is a sentence that is true in all models (of the appropriate vocabulary). That is, it is impossible to falsify a valid sentence. For example:

$$\models \forall x(\textsc{moron}(x) \rightarrow \textsc{therapist}(x)) \wedge \textsc{moron}(\textsc{mary}) \rightarrow \textsc{therapist}(\textsc{mary})$$

5.1.8 Valid Arguments

Now, validities are clearly *logical* in a certain sense; they are descriptions featuring a cast-iron guarantee of satisfiability. But logic has traditionally appealed to the more dynamic notion of *valid arguments*, a movement, or *inference* , from premises to conclusions.

Valid arguments

Suppose $\varphi_1, \ldots, \varphi_n$, and ψ are a finite collection of first-order formulae. We then call the argument with *premises* $\varphi_1, \ldots, \varphi_n$ and *conclusion* ψ a *valid argument* if and only if the following is true for this argument: Whenever all the premises are satisfied in some model using some variable assignment, then the conclusion is also satisfied in the same model using the same variable assignment. The notation

$$\varphi_1, \ldots, \varphi_n \models \psi$$

means that the argument with premises $\varphi_1, \ldots, \varphi_n$ and conclusion ψ is valid.

Terminology

There is an extensive terminology when it comes to talking about valid arguments, allowing us for example to refer to ψ as a *valid inference* from the premises $\varphi_1, \ldots, \varphi_n$, or to ψ as a *logical consequence* of $\varphi_1, \ldots, \varphi_n$.

Note that if the premises and the conclusion are all *sentences* the definition of valid arguments can be rephrased as follows: an argument is valid if whenever the premises are true in some model, the conclusion is true as well. *The truth of the premises guarantees the truth of the conclusion.*

An Example

Let's have a look at an example. The argument with premises $\forall x(\textsc{moron}(x) \rightarrow \textsc{therapist}(x))$ and $\textsc{moron}(\textsc{mary})$ and the conclusion $\textsc{therapist}(\textsc{mary})$ is valid. That is:

$$\forall x(\textsc{moron}(x) \rightarrow \textsc{therapist}(x)), \textsc{moron}(\textsc{mary}) \models \textsc{therapist}(\textsc{mary})$$

The truth of the premises $\forall x(\textsc{moron}(x) \rightarrow \textsc{therapist}(x))$ and $\textsc{moron}(\textsc{mary})$ guarantees that of the conclusion $\textsc{therapist}(\textsc{mary})$.

As the reader may suspect, there is a connection between the validity of this argument and the fact that

Deduction Theorem

$$\models \forall x(\text{MORON}(x) \rightarrow \text{THERAPIST}(x)) \wedge \text{MORON}(\text{MARY}) \rightarrow \text{THERAPIST}(\text{MARY}).$$

The example suggests that with the help of the Boolean connectives $\wedge$ and $\rightarrow$ we can convert valid arguments into validities. This is exactly what is stated by the *deduction theorem* .

5.1.9 Calculi

Validity and valid arguments are the fundamental logical concepts underlying the notion of inference. Both concepts are semantically defined, that is, they are defined in terms of models and variable assignments. But from a computational perspective, using models to actually *compute* what follows from a sentence is impossible. First of all, models may be infinite. But even if we restrict ourselves to using finite models, the fact that the semantic notion of logical consequence (as well as that of validity) is defined with respect to *all* models makes computation intractable.

Proof Theory

It is the subject of *proof theory* to capture the notion of inference in terms of syntax. This is done by defining a so-called *calculus* . A calculus is a set of rules that transform formulas into other formulas by considering only their *syntactic structure*. This transformation process starts from an input formula. Under certain conditions, a resulting sequence of rule applications is then regarded as a *proof* for the input formula. If such a proof can be generated for a formula F (in a calculus C), F is called *provable* or a *theorem* (in C). One writes $\vdash_C F$ (or just $\vdash F$ if it is clear what calculus one is talking about).

For a calculus to count as a syntactic counterpart of the semantic notion of inference, provability in that calculus ($\vdash$) and validity must co-incide. This is shown by establishing two properties of that calculus.

Correctness (*provable* implies *valid*) A calculus $\mathcal{C}$ is called *correct* or *sound* , iff $\vdash_{\mathcal{C}} \mathbf{B}$ implies $\models \mathbf{B}$.

Completeness (*valid* implies *provable*) A calculus $\mathcal{C}$ is called *complete* , iff $\models \mathbf{B}$ implies $\vdash_{\mathcal{C}} \mathbf{B}$.

In other words, if a calculus is correct and complete, all formulas that are provable in it are also valid, and all formulas that are valid are provable. Of course not every arbitrary set of rules will be a correct and complete calculus - the rules of the calculus have to be the right ones. But if a calculus is correct and complete, syntactic formula manipulation can totally replace semantic considerations. Calculi may even be employed in computational implementation: Various *automatic theorem provers* have been developed.

In what follows, we will look at a so called *tableaux calculus* for propositional logic, which is in fact correct and complete. We will use the calculus to prove valid formulae and to verify valid arguments, and in the next chapter we will give it an implementation.

5.2 Tableaux Calculi

We have discussed semantic construction methods at some length in earlier chapters. The next thing we need in order to really do natural language semantics is an appropriate calculus. In this section we will introduce a so-called tableaux calculus. Before we come to the formal characterization of our tableaux calculus, we will give you a more intuitive introduction.

5.2.1 Tableaux for Theorem Proving

We would like to show in a systematic manner that a given formula is valid (i.e. a *theorem*). If the formula is valid, it must always be true. In order to proof this we will proceed indirectly: We will try to make our formula false. If this turns out to be impossible, we know that it is valid. To try out all ways of making our formula false, we will transform it into a tree structure according to the semantics of its logical connectives. Such tree structures are known as *(semantic) tableaux* .

Here's an example. Let's try to show that the formula $\neg((p \vee q) \wedge (\neg p \wedge \neg q))$ is valid. (In contrast to the formulas we've seen so far, this formula belongs to propositional logic. Propositional logic is a less complex part of first-order logic, where all non-logical symbols are interpreted as truth values.) We start with only a single node containing our suspected theorem. As discussed, we will try to make it false. We indicate this with a superscript F (called a negative *sign*):

$$(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}$$

Signed Formulae

All formulae in our tableaux calculus will be *signed* in that way with either a T or F, instructions that tell us that we have to make a formula true or false, respectively.

So let's see how we can make our input formula *false*. It has a negation as its main connective. Thus we know that it is false iff its unnegated version is true. So, we add a node (we give it number 1 here) to our tableaux with the respective unnegated formula. We mark formulae that we have already expanded by ✓:

$$(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark$$

$$1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{T}$$

Conjunctive Expansion

Now, look at the formula at node 1. The sign tells us that we have to make this formula true. We can do so by making both of its conjuncts true. So, we add two new nodes to our tableaux (we call this a *conjunctive expansion*):

$$(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark$$

$$1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark$$

$$2 : (p \vee q)^{T}$$

$$3 : (\neg p \wedge \neg q)^{T}$$

Up to now, we have found out that we make our input formula false iff we make both (less complex) formulae $p \vee q$ and $\neg p \wedge \neg q$ true. You might already see that this is not possible, but a computer probably won't. So let us further expand these formulae!

5.2.2 Tableaux for Theorem Proving (continued)

Disjunctive Expansion

We continue our tableaux construction by expanding the formula $p \vee q$. Under what conditions is this formula true? There are two possibilities: Either p is true or q is true. We express this in a tableaux by introducing a branching (we call this a *disjunctive* expansion):

$$\begin{array}{c}
(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark \\
| \\
1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark \\
| \\
2 : (p \vee q)^{T}\checkmark \\
| \\
3 : (\neg p \wedge \neg q)^{T} \\
\begin{array}{cc} / & \backslash \\ 4 : p^{T} & 5 : q^{T} \end{array}
\end{array}$$

On each new branch, we pursue one of the possibilities to make the decomposed formula true. Now there is only one complex formula left to be expanded: $\neg p \wedge \neg q$. This expansion is like the second expansion again: The formula is true if *both* subformulae are true. So, we add these formulae conjunctively.

But to which branch should we add the new formulae? The answer is: to both. The expanded formula occurs on both branches and so both should get to see the result.

$$\begin{array}{c}
(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark \\
| \\
1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark \\
| \\
2 : (p \vee q)^{T}\checkmark \\
| \\
3 : (\neg p \wedge \neg q)^{T}\checkmark \\
\begin{array}{cc} / & \backslash \\ 4 : p^{T} & 5 : q^{T} \\ | & | \\ 6 : (\neg p)^{T} & 8 : (\neg p)^{T} \\ | & | \\ 7 : (\neg q)^{T} & 9 : (\neg q)^{T} \end{array}
\end{array}$$

This time there are four expansions that we could apply next: We could expand node 6 or 7 on the left branch, or node 8 or 9 on the right one. We choose to expand nodes 6 and 9 next. The expansion of node 6 gives us p^{F} on the left branch and the expansion of node 9 gives us q^{F} on the right branch.

$$\begin{array}{c}
(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark \\
| \\
1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark \\
| \\
2 : (p \vee q)^{T}\checkmark \\
| \\
3 : (\neg p \wedge \neg q)^{T}\checkmark \\
\begin{array}{cc} / & \backslash \\ 4 : p^{T} & 5 : q^{T} \\ | & | \\ 6 : (\neg p)^{T}\checkmark & 8 : (\neg p)^{T} \\ | & | \\ 7 : (\neg q)^{T} & 9 : (\neg q)^{T}\checkmark \\ | & | \\ 10 : p^{F} & 11 : q^{F} \\ | & | \\ \bot & \bot \end{array}
\end{array}$$

(Signed) Branch Closure

If you look at the tableaux, you will see that after these two expansions we have added a $\bot$ to both branches. We use $\bot$ to mark branches that contain a contradiction (i.e. two occurrences of the same atom with different signs). For example the left branch of our tableaux contains the contradiction p^{F} and p^{T}. Such branches are called *closed*.

This is important because each branch with a contradiction represents a failed attempt to make the input formula false. So if all branches contain contradictions, there's no way to make the input false - exactly what we wanted to know. Since this is the case in our example, we know that our input formula is always true, i.e. valid. On the other hand, if one branch had still remained contradiction-free after expanding all of its complex formulas, that branch would show a way to make the input formula false. In other words, it would specify a model in which the input formula is false, that is a model for its negation.

Branches enumerate models

In general, each fully-expanded contradiction-free tableaux branch represents a model. A model for the negation of the input formula if, as in our example, the input formula is signed F, and a model for the input formula itself otherwise. (In the first case, the F must be 'translated' into normal negation ¬). Moreover, all contradiction-free branches together specify *all* models of the (negated) input fomula.

Given this, let's state in terms of models what we've shown: Both branches of our example tableaux contain a contradiction. Therefore we know that there is no model for the formula $\neg(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))$ (where we have translated the negative sign into negation). So $(\neg((p \vee q) \wedge (\neg p \wedge \neg q))$ is true in all models, that is valid.

5.2.3 Summing up

Let us now sum up what we've done in the example with the help of a few more concise definitions. We've decomposed a formula in a tree that represents a set of case distinctions for satisfiability, or in other words, a set of candidate models. We started with an *initial tableaux* containing only one node with one signed formula.

Tableaux Inference Rules

Then we applied the following *tableaux inference rules* :

$$\frac{\mathbf{A}\wedge\mathbf{B}^{\mathtt{T}}}{\begin{array}{c}\mathbf{A}^{\mathtt{T}}\\ \mathbf{B}^{\mathtt{T}}\end{array}}\mathcal{T}(\wedge)^{\mathtt{T}} \qquad \frac{\mathbf{A}\wedge\mathbf{B}^{\mathtt{F}}}{\mathbf{A}^{\mathtt{F}}\;\big|\;\mathbf{B}^{\mathtt{F}}}\mathcal{T}(\wedge)^{\mathtt{F}} \qquad \frac{\neg\mathbf{A}^{\mathtt{T}}}{\mathbf{A}^{\mathtt{F}}}\mathcal{T}(\neg)^{\mathtt{T}} \qquad \frac{\neg\mathbf{A}^{\mathtt{F}}}{\mathbf{A}^{\mathtt{T}}}\mathcal{T}(\neg)^{\mathtt{F}} \qquad \frac{\begin{array}{cc}\mathbf{A}^{\alpha} & \\ \mathbf{A}^{\beta} & \alpha\neq\beta\end{array}}{\bot}\mathcal{T}(\bot)$$

These inference rules act on tableaux. They have to be read as follows: If the formulae above the line appear in a tableaux branch, then the branch can be extended by the formulae or branches below the line. There are two rules for each primary connective - one for each sign. Additionally, there is a rule that adds the special symbol $\bot$ to branches that contain (atomic) contradictions.

We have only given the rules for conjunction and negation. The other connectives can be defined as usual: $\mathbf{A}\vee\mathbf{B}\equiv\neg(\neg\mathbf{A}\wedge\neg\mathbf{B})$, $\mathbf{A}\Rightarrow\mathbf{B}\equiv\neg\mathbf{A}\vee\mathbf{B}\equiv\neg(\mathbf{A}\wedge\neg\mathbf{B})$

Open, Closed, Saturated

We call a tableaux branch *closed* iff it contains $\bot$, and *open* otherwise. We will call a tableaux closed, iff all of its branches are closed, and open otherwise. We use the above tableaux rules with the convention that no occurence of a formula is expanded more than once. We will call a branch (and also a complete tableaux) *saturated* if it is fully expanded, i.e. if no rule can be applied to it sticking to the convention just introduced.

Termination

This convention helps us ensure that the tableaux construction process always terminates (at least for propositional logic). Our inference rules always eliminate the primary logical connective from their antecedent (except for $\mathcal{T}(\bot)$). So, their succedents always have fewer logical connectives. As a consequence, the tableaux construction process terminates when all of the connectives are used up. In this case the formulae on all branches have been reduced to so-called *literals* and the tableaux is saturated. Alternatively, branches may be closed (by $\mathcal{T}(\bot)$) before they're saturated. Of course they need not be further expanded in this case either.

Tableaux Proof

We will call a closed tableaux with the signed formula $\mathbf{A}^{\alpha}$ at its root a *tableaux refutation* of $\mathbf{A}^{\alpha}$, and we will call a tableaux refutation of $\mathbf{A}^{\mathtt{F}}$ a *tableaux proof* for $\mathbf{A}$.

If a branch is closed, this means that there is no model for the formulae on that branch taken together; and if a tableaux is closed altogether, this means that there is no model for the input formula at all. Constructing a tableaux proof for $\mathbf{A}$ means performing an exhaustive search for models that give $\mathbf{A}$ the truth value $\mathtt{F}$. If all branches are closed, this search has failed and so $\mathbf{A}$ cannot have the truth value $\mathtt{F}$. Thus $\mathbf{A}$ must evaluate to $\mathtt{T}$ in all models, which is just our definition of validity. So a tableaux proof for $\mathbf{A}^{\mathtt{F}}$ can be constructed iff $\mathbf{A}$ is valid. To formally prove this fact (that is to establish

correctness and completeness of the tableaux-method), one has to make the relation between branches and models more precise. The reader is referred to the literature to see how this can be done.

5.2.4 Using Tableaux to test Truth Conditions and Entailments

Let us look at some further examples. To make things more interesting, we will use our proof method with a fragment of first-order predicate logic that allows us to express simple natural language sentences without introducing the whole complications of (undecidable!) first-order inference. Our fragment uses formulae of first-order logic, but without variables and quantifiers. This means that it is equivalent to propositional logic in expressivity: atomic formulae take the role of propositional variables.

We will first prove the implication 'If Mary loves Bill and John loves Mary then John loves Mary.'. We do this by exhibiting a tableaux proof of the formula

$$(\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\Rightarrow\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY})$$

which is equivalent to

$$\neg((\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\wedge\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))$$

if we eliminate the defined connective $\Rightarrow$. By exhaustively applying the inference rules above, we arrive at the following tableaux.

$$\begin{array}{c}
\neg((\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\wedge\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))^{\mathtt{F}}\\
((\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\wedge\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))^{\mathtt{T}}\\
(\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))^{\mathtt{T}}\\
\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY})^{\mathtt{T}}\\
\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY})^{\mathtt{F}}\\
\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})^{\mathtt{T}}\\
\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY})^{\mathtt{T}}\\
\bot
\end{array}$$

This tableaux has only one branch, which is closed. So the whole tableaux is closed and constitutes a tableaux proof for our implication.

?- Question!

Annotate each of the nodes of the above tableaux with the rule that has been used to add it.

As a second example let us now look at a variant problem:

1. 'Mary loves Bill or John loves Mary' $\models$ 'John loves Mary'
2. $(\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\vee\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\Rightarrow\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY})$
3. $\neg(\neg(\neg\mathrm{LOVE}(\mathrm{MARY},\mathrm{BILL})\wedge\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))\wedge\neg\mathrm{LOVE}(\mathrm{JOHN},\mathrm{MARY}))$

To prove the entailment (1) we represent it as an implication (2). Recall that the *deduction theorem* allows us to do so. We then eliminate the implication, arriving at (3).

Intuitively, (1) does not hold, because in the situation where the antecedent of the implication is true (i.e. Mary loves Bill), John need not love Mary. If we try to prove the entailment using our tableaux method, we get:

$$\begin{array}{c}
(\neg(\neg(\neg\text{LOVE}(\text{MARY},\text{BILL})\wedge\neg\text{LOVE}(\text{JOHN},\text{MARY})))\wedge\neg\text{LOVE}(\text{JOHN},\text{MARY}))^{\text{F}}\\
(\neg(\neg\text{LOVE}(\text{MARY},\text{BILL})\wedge\neg\text{LOVE}(\text{JOHN},\text{MARY}))\wedge\neg\text{LOVE}(\text{JOHN},\text{MARY}))^{\text{T}}\\
\neg\text{LOVE}(\text{JOHN},\text{MARY})^{\text{T}}\\
\text{LOVE}(\text{JOHN},\text{MARY})^{\text{F}}\\
\neg(\text{LOVE}(\text{MARY},\text{BILL})\wedge\text{LOVE}(\text{JOHN},\text{MARY}))^{\text{T}}\\
(\neg\text{LOVE}(\text{MARY},\text{BILL})\wedge\neg\text{LOVE}(\text{JOHN},\text{MARY}))^{\text{F}}\\
\begin{array}{c|c}
\neg\text{LOVE}(\text{MARY},\text{BILL})^{\text{F}} & \neg\text{LOVE}(\text{JOHN},\text{MARY})^{\text{F}}\\
\text{LOVE}(\text{MARY},\text{BILL})^{\text{T}} & \text{LOVE}(\text{JOHN},\text{MARY})^{\text{T}}\\
 & \bot
\end{array}
\end{array}$$

The tableaux we've just constructed is saturated (so we cannot expand it any further) and *not closed*. In fact, as we've already convinced ourselves above, our initial entailment conjecture doesn't hold, and so there is no tableaux proof for it.

A Model on an Open Branch

But we know more. The open and saturated branch represents a way of giving the input formula the truth value indicated by the sign, that is, a model for its negation. Let's see how. The literals on the open branch of the above tableaux (marked green) taken together characterize the situation in which the conjectured entailment fails to hold (namely the situation where Mary loves Bill but John does not love Mary). These are the 'minimal requirements' on a model for the negated input formula.

5.2.5 An Application: Conversational Maxims

Conversational Maxims

Now that we have a computational method for solving inference problems, let's look at a case where we can apply it in semantic interpretation. We shall use inference to check whether a speaker obeys the *conversational maxim* s in his utterance. The notion of conversational maxims was introduced by H.P. Grice in 1975 ([9]). He postulates a set of constraints on discourses, which he formulates as maxims for the speaker. These maxims characterize discourse as rational cooperative activity. The hearer can assume that the speaker follows these maxims, and on this assumption can draw inferences to the intended interpretation of the discourse: Often if one of multiple readings violates a maxim, then it simply cannot be the the intended one.

Conversational Implicatures

In other cases a violation allows to infer 'backwards' to an intention or assumption on the side of the speaker. Propositions that can be inferred from violations of the conversational maxims are called *conversational implicature* s.

Be cooperative!

Grice assumes that participants in a discourse follow a general *cooperative principle* . This principle leads to more specific submaxims, falling into one of four categories:

1. Quality Try to make your contribution one that is true.

2. Quantity Make your contribution as informative as is required.
3. Manner Be relevant.
4. Relation Be perspicuous

Generally, Grice's maxims are viewed as pragmatic in nature. As regards the maxims of manner and relation, it may indeed not be easy to see how *being relevant* or *being perspicuous* could be defined solely in semantic terms, without reference to more general factors such as e.g. the intentions, mutual knowledge or the sociolect of speakers/hearers. In contrast, we *can* get a grip on the first two maxims without having to tackle all of the complexities of pragmatics, if we use inference techniques on our semantic representations.

5.2.6 The Maxim of Quality

We now show how we can use inference to check whether an utterance - given some previous discourse - conforms to the maxims of quantity and quality (or, more precisely, we show how to detect a lot of cases where it doesn't). We will formulate inference tasks that help us decide this question and that we can give to (for instance) a tableaux prover.

Quality

First, we shall look at the *maxim of quality*. An utterance must at least be consistent with the preceding discourse in order to be true. Now this is definitely something we can decide using a theorem prover.

An Inference Task

Let's suppose we want to check the consistency of an utterance φ (more precisely the formula representing the meaning of the utterance) with respect to a preceding (consistent) discourse, which as a first approximation, we take to be the conjunction of the logical forms of the n sentences uttered so far $\psi_1\wedge\ldots\wedge\psi_n$. How can we do this?

We proceed indirectly: We check whether $\psi_1\wedge\ldots\wedge\psi_n\wedge\varphi$ is *un*satisfiable. If so, then we know that φ is *not* consistent with $\psi_1\wedge\ldots\wedge\psi_n$ (because we have assumed that the preceding discourse is consistent, we know that φ is to blame for the inconsistency). Otherwise, we know that φ is consistent with $\psi_1\wedge\ldots\wedge\psi_n$.

How can we use our tableaux calculus to find out if $\psi_1\wedge\ldots\wedge\psi_n\wedge\varphi\wedge\psi$ is unsatisfiable? Up to now, we've only seen how to prove theorems. But how can we reduce inconsistency checks to this task? We just have to look at the negation of the formula that we want to prove unsatisfiable. If this negation is a theorem, we know that the unnegated formula is unsatisfiable. So we will take the negation of the conjunction for the complete discourse (i.e. $\neg(\psi_1\wedge\ldots\wedge\psi_n\wedge\varphi)$), and check if it is a theorem. This theorem-check is where our tableaux-prover comes in. We feed the negated formula $\neg(\psi_1\wedge\ldots\wedge\psi_n\wedge\varphi)^{\text{F}}$ to it and try to construct a closed tableaux. If we manage to build one, we can 'infer backwards' a little.

Here is how, step by step:

1. $\neg(\psi_1 \wedge \ldots \wedge \psi_n \wedge \varphi)$ is a theorem (this is what we've proven on our tableaux).
2. Hence the unnegated $\psi_1 \wedge \ldots \wedge \psi_n \wedge \varphi$ must be unsatisfiable.
3. This means that the discourse corresponding to $\psi_1 \wedge \ldots \wedge \psi_n \wedge \varphi$ is inconsistent.
4. But the previous discourse $\psi_1 \wedge \ldots \wedge \psi_n$ is consistent (by assumption).
5. Hence the inconsistency can be traced back to adding utterance φ.
6. Finally, this means that uttering φ after having uttered $\psi_1 \wedge \ldots \wedge \psi_n$ violates the Maxim of Quality.

Let us look at a (very) small discourse as an example: 'If Mutz is a Siamese cat, then Mary likes her. Mutz is a Siamese cat.'. Given this 'discourse', we can use our tableaux calculus to detect that the sentence 'Mary doesn't like Mutz' violates the maxim of quality. We have to construct a closed tableaux for the following input (since we do not have any treatment of pronouns, we formalize 'her' as if it was 'Mutz'):

$$(\neg((\mathrm{SIAMESECAT}(\mathrm{MUTZ}) \Rightarrow \mathrm{LIKE}(\mathrm{MARY},\mathrm{MUTZ})) \wedge \mathrm{SIAMESECAT}(\mathrm{MUTZ}) \wedge \neg\mathrm{LIKE}(\mathrm{MARY},\mathrm{MUTZ})))^F$$

This is equivalent to:

$$(\neg((\neg(\mathrm{SIAMESECAT}(\mathrm{MUTZ}) \wedge \neg\mathrm{LIKE}(\mathrm{MARY},\mathrm{MUTZ}))) \wedge \mathrm{SIAMESECAT}(\mathrm{MUTZ}) \wedge \neg\mathrm{LIKE}(\mathrm{MARY},\mathrm{MUTZ})$$

5.2.7 The Maxim of Quantity

Quantity

Let's now turn to the *maxim of quantity*. To be 'as informative as required', an utterance must (most of the time...) at least be informative *at all*. We can get a grip on this minimal requirement using inference. The key idea is that an utterance must contain something new to be informative. And to count as something new logically, the content of the utterance must not be implied by the preceding discourse anyway. We know that if it *is* implied, the implication with the preceding discourse as antecedent and the (not so) new utterance as consequent will be valid.

The Inference Task

So (again given a preceding discourse $\psi_1 \wedge \ldots \wedge \psi_n$) let's suppose we want to to find out whether some utterance φ is informative. As we said, we check whether

$$\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$$

is valid (that means, whether it is a theorem). In our tableaux calculus, we thus have to attempt to construct a closed tableaux for the equivalent:

$$\neg((\psi_1 \wedge \ldots \wedge \psi_n) \wedge \neg\varphi)^F$$

If we manage to do so, we know that the new utterance is *not* informative and thus violates the maxim of quantity. Otherwise, we shall take it to be informative.

?- Discussion!

Give examples of violations of the maxims of quality and quantity that would not be detected by our approach!

Let us emphasize that Grice's point is not that utterances violating any of the conversational maxims are ill-formed in the sense of ungrammatical strings. Rather, a speaker may violate a maxim on purpose, allowing the hearer to infer 'backwards' to the speaker's intention. Can you think of situations where this happens?

?- Discussion!

Our treatment of the informativity constraint is obviously oversimplified in that it counts to many utterances as violating the maxim of quantity. The problem is that we assume that all consequences of the complete discourse are always equally present to a hearer. How could we solve (or at least alleviate) this problem?

5.3 Tableaux Web-Interface

Click here![1]

In the next chapter, we will discuss the implementation of our propositional tableaux calculus just presented. If you want to use the calculus right away, have a look at our Web-Interface[2]. You can either generate tableaux for some given example formulae or type in formulae yourself (using our familiar Prolog syntax). This might help you doing your exercises.

Propositional example formulae like the ones we discussed in this chapter can be found in the choice box *Propositional*. If you type in examples by hand, don't care about the *QDepth* input field.

Try this!

Take for example the tableaux we have seen in Section 5.2.4 and compare it to the tableaux our system generates:

`love(mary,bill) & love(john,mary) > love(john,mary).`

Note that you can feed the formula

```
love(mary,bill) & love(john,mary) > love(john,mary)
```

directly to our system. But of course you can also feed the equivalent formula without defined connectives:

```
~((love(mary,bill) & love(john,mary)) & ~love(john,mary)).
```

Don't forget to choose whether you want to make the formula true or false.

[1] http://www.coli.uni-saarland.de/projects/milca/cgi-bin/Tableaux/tableaux.cgi
[2] http://www.coli.uni-saarland.de/projects/milca/cgi-bin/Tableaux/tableaux.cgi

6

Tableaux Implemented

In this chapter, we present an implementation of the tableaux algorithm we've presented in the last chapter.

6.1 Implementing PLNQ

Now let's turn to an implementation of what we've learnt. Basically, we only have to do two things:

1. We have to devise data structures for handling tableaux.
2. We have to represent the tableaux inference rules (page 89) in Prolog.

As regards the first task, our decision is this: We do not *represent* tableaux explicitely. Rather, we use Prolog's backtracking mechanism to 'crawl along' the tableaux under construction. This makes the second task almost trivial - really all we have to do is *translate* our inference rules into Prolog. The resulting program is somewhat different from what you might expect if you think about constructing tableaux with pen and paper. Nevertheless it's elegant and easily implemented.

6.1.1 Literals

The recursive predicate `tabl/3` implements the core of our tableaux system.

The first argument of `tabl/3` is the input formula. In the first call, this is the formula with which we start our tableaux. The second argument (`InBranch`) stores the literals we have derived so far on the branch under construction. In the first call this argument will just be the empty list `[]`, but we need it as an accumulator in the recursion.

The last argument (`OutBranch`) of `tabl/3` will finally contain the model we've constructed on some branch (as a list of literals). In this setting, `OutBranch` will be the output of our predicate. But for the time being, you can safely ignore this argument except when we mention it explicitely.

The predicate `tabl/3` has six clauses. The base case is for literals, whereas the recursive clauses handle complex formulae. We will first look at the base case. It is the last clause in the program, after the clauses for the complex formulae, so we can be sure that its input `F` is a literal. (Of course to be sure of this, we additionally have to include cuts in the other clauses to prevent backtracking to the 'literal case'). Here it is:


```
tabl(F,InBranch,OutBranch) :-
        OutBranch = [F|InBranch],
        \+ clash(OutBranch).
```

In this clause, we determine whether `F` is compatible with our input model. We add `F` to `InBranch`, then test if it was compatible. If it was, we return the result (in `OutBranch`). Otherwise the clause fails. Remember that `InBranch` contains all the literals that we have already derived on the current branch. If the new literal we're considering contradicts any of these facts, the current branch is closed. So in effect we signal *branch closure* by letting the above clause of `tabl/3` fail.

The compatibility check we do is actually a negated incompatibility check. It is done by calling the auxiliary predicate `clash/1` on `OutBranch` (which is equivalent to `InBranch` together with the new literal `F`).

The predicate `clash/1` is implemented as follows:

```
clash(List) :-
        member(true(A),List),
        member(false(A),List).
```

In our implementation we simply translate the signs of our calculus ($\mathtt{T}/\mathtt{F}$) into the Prolog atoms `true` respectively `false`. Our predicate `clash/1` looks whether the list `Literals` contains the same atomic formula `A` twice, once signed `true` and once signed `false`.

?- Question!

A simpler clash test would suffice for our purposes (and make the program more efficient). Do you have an idea how to implement one?

6.1.2 Complex Formulae: Negation

***See file* `propTabl.pl`.**

We now have dealt with the literal case. But we still have to deal with complex formulae. Let us start with the clauses for negation, directly modeled on the rule $\mathcal{T}(\neg^{\mathtt{T}})$ (page 89).

```
tabl(true(~A),InBranch,H) :-
        !,tabl(false(A),InBranch,H).

tabl(false(~A),InBranch,H) :-
        !,tabl(true(A),InBranch,H).
```

These clauses are almost self-explaining. They simply strip off the negation symbol and turn the sign of the formula. The cuts are there to prevent backtracking to the clause for literals, which would also match.

The recursive call covers the branch of the tableaux below the negated formula. Generally, all clauses for complex formulae will consist of recursive calls to `tabl/3` on the decomposed input. Failure in one of these calls always means that the corresponding part of the tableaux is closed. In Section 6.1.5 we illustrate all this by an example.

6.1.3 Complex Formulae: Conjunctive Expansion

See file **`propTabl.pl`.**

We now look at the clause for positive conjunction, which corresponds to the rule $\mathcal{T}(\wedge)^{\mathtt{T}}$ (page 89). Again, the cut prevents backtracking to the clause for literals.

```
tabl(true(A & B),InBranch,OutBranch) :-
        !,tabl(true(A),InBranch,K),
        tabl(true(B),K,OutBranch).
```

To make a conjunction true, we first make the first conjunct true. Then we take what model we've generated for the first conjunct (contained in `K`) and use it as input to make the second conjunct true. Note that here we really need the last argument of `tabl/3`.

If the second call to `tabl/3` succeeds in the end, `OutBranch` contains all the literals generated when verifying both the first and second conjunct.

This is related to the way we would normally (with pen and paper) construct a tableaux as follows: Each one of the two recursive calls to `tabl/3` in this clause covers one part of the branch below the conjunctive formula. If any of these two calls fails, so will the whole clause containing them. This is correct because both calls cover part of *the same* branch, and closure in any of these parts should affect the branch as a whole.

6.1.4 Complex Formulae: Disjunctive Expansion

See file **`propTabl.pl`.**

Let's finally look at conjunctions in a negative context (i.e. disjunctions). Remember that the rule $\mathcal{T}(\wedge)^{\mathtt{F}}$ (page 89) introduces a branching. If we find a negated conjunction, we have to falsify either the first or the second conjunct. We express this 'either... or...' in Prolog by writing two clauses, each of them covering one of the two branches:

```
tabl(false(A & _),InBranch,H) :-
        tabl(false(A),InBranch,H).

tabl(false(_ & B),InBranch,H) :-
        !,tabl(false(B),InBranch,H).
```

In the first clause, `tabl/3` is called with the first conjunct (signed `false`) as input. If this call succeeds, everything is fine and we get the resulting open `OutBranch` as output. Prolog will then simply forget about the second clause (resp. disjunct/branch). Otherwise (i.e. if the first call fails), Prolog backtracks to the next clause in the program, the second clause for negative conjunction. There `tabl/3` is called with the second conjunct as input, generating the resulting `OutBranch`. This corresponds to the second branch of the $\mathcal{T}(\wedge)^{\mathtt{F}}$-rule.

Note that this time we put a cut only in the second of the two clauses. The reason is that we want to allow backtracking from the first clause to the second one. But of course we still do not want to have backtracking to the clause for literals: If both of the clauses for the negative conjunction fail, the cut in the second clause prevents any further backtracking. So this call of `tabl/3` fails. This is exactly as it should be, because in this situation our program has found a contradiction on *both* branches opened by the $\mathcal{T}(\wedge)^{\mathtt{F}}$-rule. Hence the whole subtableaux for the negative conjunction is closed.

6.1.5 An Example - first Steps

Here comes an example that will help us understand how the clauses of `tabl/3` are related to the construction of a tableaux as we would usually draw it. Let's take the formula $\neg((\mathrm{RUN(JOHN)} \wedge \neg\mathrm{SLEEP(MARY)}) \wedge (\mathrm{SLEEP(MARY)}))$, and suppose we want to make it false. This should result in a closed one-branch tableaux, containing the contradiction $\mathrm{SLEEP(MARY)}^{\mathtt{T}}$ vs. $\mathrm{SLEEP(MARY)}^{\mathtt{F}}$. Now let's see how our program finds this tableaux.

Computation Tree

We represent Prolog's computation using a so called *computation tree* . This tree shows the sequence of calls that Prolog has to execute in order to prove its main goal (the one given in the initial call). Each node of a computation tree corresponds to a state of the computation. Each node contains a stack with the goals that have to be proven at the corresponding state of the computation. The topmost goal on the stack is always the one just under consideration at that state. If this goal produces new subgoals, they replace it on top of the stack in the next state (i.e. at the daughter node in the computation tree). If a goal succeeds, it is removed, but the resulting instantiations of variables are kept and used for the goals on the stacks below in the tree, which are still to be processed.

Computation trees branch whenever there are multiple clauses compatible with a call (i.e. in the case of a disjunction in the Prolog code). The branchings correspond to backtracking points in the programm. The computation tree for our running example does not branch, but we will soon see one that does.

Initial State

We start with an initial tableaux consisting of our input formula only, and a computation tree with only one node that contains the top goal:

$\neg((\mathrm{RUN(JOHN)} \wedge \neg\mathrm{SLEEP(MARY)}) \wedge (\mathrm{SLEEP(MARY)}))^{\mathtt{F}}$

```
[tabl(false(~((run(john)& ~sleep(mary))& sleep(mary))),[],Out)]
```

Steps One and Two

In the following step, the negation symbol is stripped off and the sign of the whole formula is turned from `F` to `T`. We don't show the tableaux and computation tree for this step. We directly look at the next one, where the conjunction is handled. From now on, we will abbreviate $\mathrm{RUN(JOHN)}$ as P and $\mathrm{SLEEP(MARY)}$ as Q. The dotted boxes on the tableaux indicate how the formulae on the tableaux derive from each other: All formulae within a box (transitively) derive from the topmost one in that box. So the tree and tableaux after handling the first conjunction look as follows:

$\neg((P \wedge \neg Q) \wedge Q)^{\mathtt{F}}$

$(P \wedge \neg Q) \wedge Q^{\mathtt{T}}$

$\boxed{\mathbf{P \wedge \neg Q^{\mathtt{T}}}}$

$\boxed{Q^{\mathtt{T}}}$

```
(0)  [tabl(false(~((P&~Q)&Q)),[],Out)]
                 |
(1)  [tabl(true((P&~Q)&Q),[],Out)]
                 |
(2)  [tabl(true(P&~Q),[],Out1)  tabl(true(Q),Out1,Ou
```

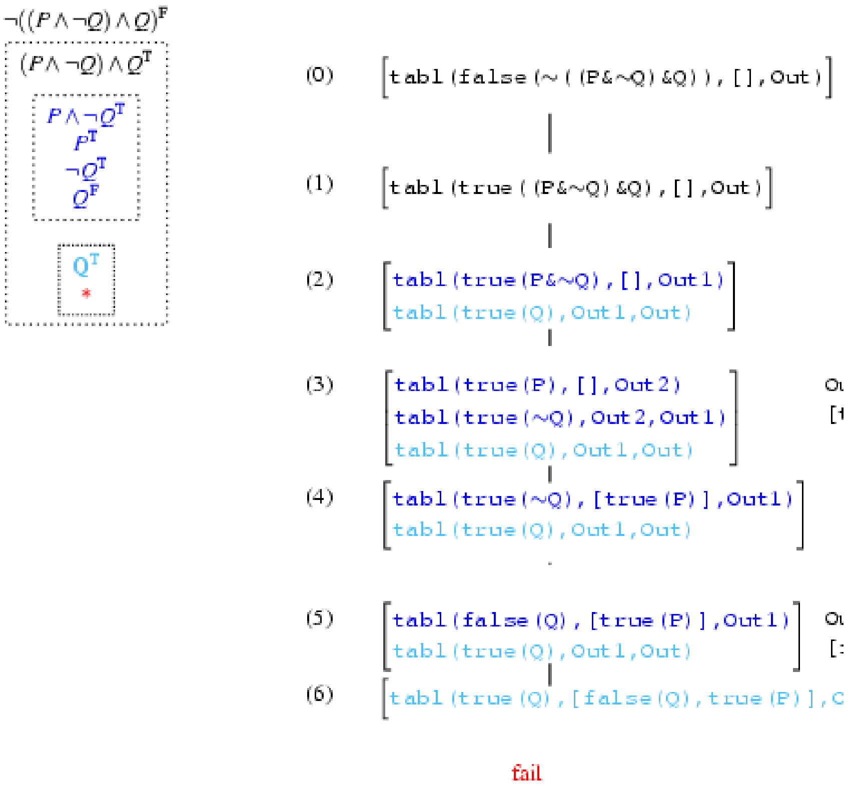
Notice that now for the first time, the stack in our computation tree contains two goals. This is because we had to handle a conjunction in step (1) and the clause of `tabl/3` for conjunction consists of *two* subgoals, namely the two recursive calls to `tabl/3` on the two conjuncts. The call for the first conjunct is top - Prolog will first consider this goal.

Looking at the Tableaux

But what does this mean on the side of the tableaux? It means that the tableaux will (for some time) grow "in the middle": below the formula we've added for the first conjunct (the blue one), but *above* the one for the second conjunct. Thus the next formulas are added *inside the dotted box* around the first conjunct.

6.1.6 An Example - final Step

Let's now look at the final step of the computation, when the program finds the contradiction `false(P)` vs. `true(P)` and fails:



How did we get here?

How did we get here? In step (3), there are three goals on the stack: The top goal from step (2) has produced two new subgoals, which are now on top. The third goal is the one that was second in step (2). It remains untouched for the time being. Steps (4) and (5) merely consist of the obvious recursive calls triggered by the subgoals on the stack.

Literals

In steps (3) and (5), `tabl/3` is called on a literal. Therefore these are the steps where output arguments get instantiated. We've indicated this by notating the corresponding assignments to the right of the stack at the respective nodes. In step (3) `Out2` is instantiated to `[true(P)]`, and in step (5) `Out1` gets determined to `[false(Q),true(P)]`. If there were no other goals on the stack in step (5) this would be the output of the whole computation.

Finding the Contradiction

But there still is the goal that was put on the stack in step (2) (corresponding to the rightmost conjunct in the original input formula). Finally, in step (6) this goal becomes top of the stack and is processed. The literal `true(Q)` is added to the incoming branch and a `clash/1` test is performed (in the computation tree we've left out the `clash/1`-subgoals that come from `tabl/3` calls on literals. You may have noticed this already at steps (3) and (5)). Because `true(Q)` contradicts `false(Q)` on the incoming branch, there is a clash and the computation fails.

In our example, there are no possibilities for doing anything else now. The `fail` means that Prolog answers `no`, all previous instantiations are undone and the variable for the outgoing model of our top level call (`Out`) remains undetermined. This reflects the fact that the tableaux for our input formula has only one branch, and that this one branch is closed, i.e. contains no model.

But next, we'll look at a case where there is an alternative branch that remains open. We'll have a branching computation tree there, and Prolog's `fail` will trigger backtracking.

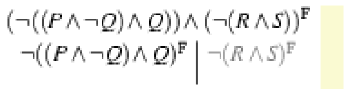
6.1.7 Another Example

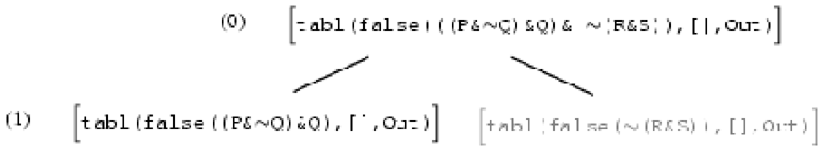
Suppose we are falsifying the formula:

$(\neg((\text{RUN(JOHN)}\wedge\neg\text{SLEEP(MARY)})\wedge(\text{SLEEP(MARY)})))\wedge\neg(\text{CUSTOMER(MARY)}\wedge\text{CUSTOMER(JOHN)})$

We will abbreviate this formula as $\neg((P\wedge\neg Q)\wedge Q)\wedge\neg(R\wedge S)$. Notice that the main connective of this formula is a conjunction and the first conjunct is exactly the same formula as in the last example. Expanding this conjunction (under negative polarity - we said that we want to falsify the formula) leads to the following *branching* tableaux and computation tree:

A Branching





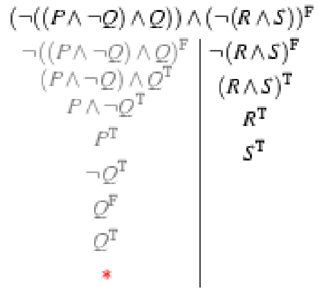
***See file* `propTabl.pl`.**

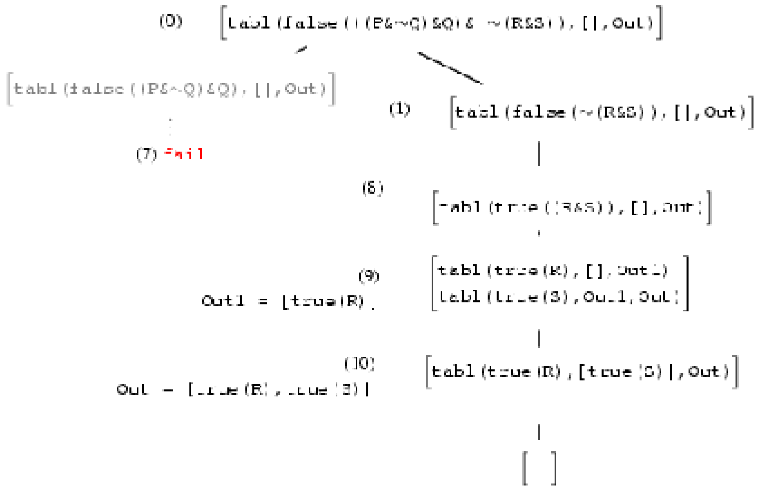
The tableaux has two branches according to our rule $\mathcal{T}(\wedge)^{\mathbf{F}}$. The left one is exactly as in our last example and will end up closed. The computation tree also branches, and here again the left branch is as in our last example and will end in a `fail`. But why does the computation tree branch? Look at the clauses of `tabl/3`: There are two clauses that match for a negative conjunction. One leads to a recursive call on the left subformula, the other to a recursive call on the right one. The two branches of the computation tree correspond to these two matching clauses.

Backtracking and Computation Trees

The branches of the computation tree are visited from left to right. After a while the `fail` on the left branch is reached (i.e. the left branch of our tableaux closes). As we know, `fail` triggers backtracking. On our computation tree, this means that Prolog goes upward from the `fail` to the next branching point above, and chooses the next branch to the right to continue its computation. The branch to the left is discarded (and so are all instantiations that took place below the branching point).

Hence the right conjunct of our original input formula is further expanded, adding to the right branch of the tableaux. Finally, when the right conjunct is fully broken down into literals, the top-level output argument `Out` is instantiated to `[true(R), true(S)]`.





What is the result?

Now we know that our input formula (page 100) is no theorem. Moreover, we have generated a (in fact *the*) counter example in `OutBranch`: If `customer(mary)` and `customer(john)` are true, the whole input formula is false.

6.1.8 Two Connectives

Notice that we've only discussed (and in fact that we've only implemented) tableaux rules for the connectives ¬ and ∧ so far. This made things simpler for us, and as you probably know it is always possible to treat all other connectives as defined in terms of these two.

***See file* `comsemLib.pl`.**

We will now give an implementation of the predicate `naonly/2` that takes a formula which uses the full set of connectives to an equivalent one that uses only ¬ and ∧. All we have to do is to replace the defined connectives according to their definitions:

```
naonly(~(X),~(Z)) :-
        !,naonly(X,Z).

naonly(X & Y, Z & W) :-
        !,naonly(X,Z),
        naonly(Y,W).

naonly(X v Y,~(~(Z) & ~(W))) :-
        !,naonly(X,Z),
        naonly(Y,W).
```

```
naonly(X > Y,~(Z & ~(W))) :-
        !,naonly(X,Z),
        naonly(Y,W).

naonly(X,X) :- !.
```

We have to use cuts in the first clauses because we don't want to allow backtracking to the last one (which always matches). If we didn't, we would get additional spurious solutions with only parts of the input formula converted.

6.2 Wrapping it up (Theorem Proving)

The file `prop.pl` contains a simple driver for theorem proving:

```
theorem(Formula) :-
        naonly(Formula,ConvFormula),
        \+ tabl(false(ConvFormula),[],_).
```

Test it! `theorem(walk(john) v (~walk(john))).`

All Files

Here's a summary of the files that make up the implementation we've discussed:

See file `propTabl.pl`.	The core of the implementation: `tabl/3` and `clash/3`
See file `comsemLib.pl`.	Auxiliary predicates: `naonly/2` and `toconj/2`
See file `prop.pl`.	The wrapper for model generation and theorem proving: `modGe`
See file `comsemOperators.pl`.	Our usual operator definitions

Further Reading

A textbook introduction to formal logics including a discussion of propositional and first-order tableaux methods is [6].

Bibliography

[1] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language*. CSLI Press, 2004. Accepted for publication. See also `www.comsem.org`, `www.blackburnbos.net`, and `http://www.loria.fr/~blackbur/`.

[2] Aljoscha Burchardt, Stephan Walter, Alexander Koller, and Manfred Pinkal. The MiLCA Saarbrücken Project. `http://www.coli.uni-saarland.de/projects/milca/`, 2003.

[3] Aljoscha Burchardt, Stephan Walter, and Manfred Pinkal. MiLCA - Distance Education for Computational Linguistics. Accepted for EDEN 2004 conference, Budapest, 2004.

[4] Denys Duchier. Oz Documentation DTD. `http://www.mozart-oz.org/documentation/ozdoc/index.html`.

[5] Markus Egg, Alexander Koller, and Joachim Niehren. The constraint language for lambda structures. *Journal of Logic, Language, and Information*, 10:457–485, 2001.

[6] Melvin Fitting. *First-Order Logic and Automated Theorem Proving. Second Edition*. Springer, 1996.

[7] Ruth Fuchss, Alexander Koller, Joachim Niehren, and Stefan Thater. Minimal recursion semantics as dominance constraints: Translation, evaluation, and analysis. In *Proceedings of the 42nd ACL*, Barcelona, 2004.

[8] L. T. F. Gamut. *Logic, Language, and Meaning*, volume 1: Introduction to Logic. The University of Chicago Press: Chicago, London, 1991.

[9] H. P. Grice. Logic and conversation. In P. Cole and J. L. Morgan, editors, *Syntax and Semantics: Vol. 3: Speech Acts*, pages 41–58. Academic Press, San Diego, CA, 1975.

[10] W. R. Keller. Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 432–447. Reidel, Dordrecht, 1988.

[11] Richard Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, CT, 1974.

Index