Computational Semantics

Aljoscha Burchardt
Stephan Walter
Alexander Koller
Michael Kohlhase
Patrick Blackburn
Johan Bos

MiLCA, Saarbrücken

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. This is what we will be concerned with in this course.

MiLCA
Computerlinguistik, Universität des Saarlandes
Saarbrücken, Germany
November 2002

Contents

Course Schedule

Main Parts of the Course

One question ahead

But there's one more question, which we have to answer in the first place: Meaning as such is a very abstract concept. It's not at all easy to imagine what it means to 'work with meanings' directly. So what are we going to do? 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 representations - strings of a formal language that has technical advantages over natural language. In this course, we will represent meanings using formulas of first order logic. First order logic gives us (at least) two things: First, a formal language with desirable properties such as having a simple, well defined (and unambigous) syntax. This makes it fit for use with the programs we're going to implement. Second, there is the *truth-functional interpretation* telling us unambigously under which conditions formulae hold 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. So, via meaning representations we come a step closer to understanding how sentences manage to convey something about how the world is.

First Order Logic

First order logic and Prolog

The first two lectures of this course pave the way for the rest. They're directly concerned with first order logic, the meaning represenation language that everything we're going to do in this course is based upon. Chapter 1 contains the basic concepts of first order logic, giving us the background we need in order to understand how our representation language works. We will explain central terms of first order logic, such as *first order formula*, *model*, *satisfaction* and *truth in a model*. This lecture serves as a repetition and as a reference for later chapters. In Chapter 2 we then turn to the computational side of our enterprise. We show how to represent first order formulae and models in Prolog. Furthermore, we will see how to make sense in implementational terms of the definitions given in the previous lecture, We will look at the example of the definition of *truth in a model* and implement a small model checker.

In the next three lectures (which form the first main part of this course) we will mainly use first order logic in virtue of its nice properties as a *formal language*. We will not talk about truth or models for formulae. Still of course, these notions remain in the background as the main reason why we say for instance that the meaning of 'Every man walks' is captured by the first order formula $\forall x.\text{MAN}(x) \rightarrow \text{WALK}(x)$. They will

become the focus of our interest again (although from a new perspective) in the second main part of the course, which deals with inference. So now for the two main parts.

Semantic Construction

The first main part of the course (consisting of lectures 3-6) is concerned with the task of *semantic construction*. Because we use first order logic as our meaning representation language, our question from above now turns into: 'Given a sentence, how do we get to the first order formula that represents its meaning?'.

λ-Calculus

Chapter 3 shows that this is not a trivial question. Using a first, quite naive approach to the matter we will soon encounter a lot of problems. We solve these problems by introducing λ-calculus. This calculus has been a standard tool for semantic construction ever since the pioneering work of Richard Montague in the 1960s. It allows us to compose complex formulae in an elegant manner: Making use of β-reduction, the key technique introduced with λ-calculus, we arrive at formulae of first order logic for natural language sentences by a stepwise simplification of other expressions that correspond to the structure of those sentences more directly.

As an application, we present an implementation of semantic construction in the spirit of Montague semantics. This allows us to automatically construct the first order representations for natural language input sentences such as 'A therapist loves a siamese cat'. You can test this implementation here (page 73). So by the end of this lecture, we will have our first program that masters the task of semantic construction for a small fragment of English. But there's still a lot to be done.

Implementation

Starting off from the Prolog program we've just developed, we focus on implementational considerations in the next lecture (Chapter 4): We take a second look at our program, this time from the perspective of software engineering. We redesign it from a single-purpose implementation into a more general and modular framework for semantic construction. This framework will easily extend and adapt to cover new sorts of semantic phenomena that we may encounter.

Scope ambiguities and Underspecification

Then, in Chapter 5, we return to the linguistic side and take a look at one such new class of phenomena: *scope ambiguities*. In the case of a scope ambiguity, multiple meanings are associated with one and the same sentence in a systematic way. This may happen e.g. because a sentence contains several quantified noun phrases (the infamous 'Every man loves a woman.' is one such sentence).

The study of scope ambiguities has been a central issue in natural language semantics for a long time. We will first explain briefly why such ambiguities occur and then give a historical overview of how people have tried to deal with them. We explain why the early extensions of Montague semantics could not treat the central phenomena satisfactorily. Then we discuss the use of *underspecified representations*. Underspecified representations allow us to speak about the parts formulae (representing for instance

ambiguous sentences) are made of, but we don't have to commit to one arrangement of these parts (i.e. one reading). This turns out to be the key to solving a whole bunch of problems scope ambiguities pose for semantic construction.

Finally, we turn to CLLS, a calculus for semantics construction based on underspecification. In Chapter 6 we implement this calculus and integrate it into our semantics construction framework. The resulting system allows us for example to get the five readings for the following sentence: 'Every owner of a siamese cat loves a therapist.'. You can run the program here (page 112). When we develop this implementation we shall greatly benefit of the work we put into re-designing our semantic construction program to a general framework in the previous lecture: We just have to give Prolog code for the interesting changes, while being able to re-use all of the periphery that we already have at hand.

Inference

Up to this point in our course, we've mostly been concerned with the business of building a logical formula for a given natural language sentence that adequately describe its meaning (or - for that matter - several formulae for several meanings). Now, we are going one step further, approaching the second question we posed ourselves above: Given that we have the meaning of a sentence, what can we do with it?

The key idea that we will pursue when answering this question is that - intuitively speaking - doing something with the meaning of a sentence means finding out *what follows*. For example, when we know that Mutz is a siamese cat and we hear: 'John doesn't have any pet.', we *do something* with the meaning of this sentence when we come to the conclusion that John isn't Mutzi's owner.

On the level of meaning representations, 'finding out what follows' from a sentence means *drawing inferences* from the first order formula that corresponds to that sentence. Tasks of finding out what follows (that is, inference tasks) play an important role at many different stages in the process of understanding a sentence. In the example above we inferred from the formulae for a few sentences (often, we will also make use of additional world knowledge). This kind of inference may e.g. be neccessary to find out what the speaker intended us to do when he uttered the sentence, or it may help us to exploit the information the speaker conveys for our purposes. It extends to the interface between semantics and pragmatics (and sometimes beyond it). But inference may be of use in semantic processing already at much earlier stages. We can e.g. use inference mechanisms to find out what would follow from one reading of a sentence (as opposed to another one) when we have to decide whether to discard or prefer the reading, and base our decision on our findings..

Inference in Computational Semantics

For us as *computational* semanticists these are a lot of good reasons why we should try to get a grip on drawing inferences *computationally*. In Chapter 7 we take a closer look at techniques for this purpose. We introduce the notion of a *proof* as well as mechanisms to work with this notion. These mechanisms (called *calculi*) capture *semantic* concepts (like that of a valid argument, which we learned about in the first lecture) via methods for manipulating formulae (i.e. *syntactic* objects). They are therefore well

suited for use in computational semantics - after all manipulating formulae is something that can be done by a computer.

The calculus we discuss in detail is that of (semantic) tableaux. In this lecture, we shall look at tableaux for propositional logic. One advantage of tableaux is that we can also use them to generate models for a given formula. As we shall see this offers a new perspective on a variety of natural language phenomena. Another advantage is that this calculus is good for direct implementation in Prolog. Something we shall prove in practice by giving such an implementation.

First Order Inference

In Chapter 10 we generalize to first order logic what we learned about propositional inference in the previous lecture. We extend our tableaux calculus, and change the implementation accordingly. While the calculus extends readily, extending the implemenation is not a trivial task at all. We will discuss why this is so, and then take the trouble of integrating what non-trivial additions and changes we need into our implementation.

Discourse Representation Theory

Up until now our only concern was the meaning of single sentences. In this chapter, we look at discourse, i.e. sequences of sentences. Interesting challenges arise that go beyond the tools and techniques for computational semantics we have developed so far. One of these challenges is interpreting pronouns - words like he, she and it which indirectly refer to objects. In this chapter we will introduce and show how one can build semantic representations of texts and develop algorithms for resolving pronouns to their textual antecedents.

And finally...

The proof of the pudding

And finally, the proof of the pudding is the eating: We've implemented a number of programs - a model checker, a semantics construction engine, an inference system. What can they do if they work together? Chapter 12 asks you to explore this. It contains a collection of exercises that are meant to be starting points for your end-term projects. Have fun!

1

First-Order Logic

1.1 Basic Concepts

1.1.1 Vocabularies

Our ultimate goal in this lecture is to define how first-order formulas are evaluated in first-order models. In general terms, the purpose of the evaluation process is to tell us whether a description is true or false in a situation.

We shall look at this in a moment - but first, it's important to point out a relevant issue. 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. For example, if we examine a formula (that is, a description - see above) from a first-order language intended for talking about various relations and properties like *loving*, *being a moron*, and *being a therapist* that hold between the characters Mary, Anna, John, and Peter, while being provided with a model (that is, a situation - see above) recording information about something entirely different (say, about which household detergents are the best choice to get rid of nasty stains) then it makes no sense at all to evaluate this particular formula in that particular model. But a *vocabulary* (or a *signature* , as a vocabulary is also called) allows us to avoid such problems: It tells us which first-order language belongs to a given model.

Here is our first vocabulary:

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

Vocabularies connect Formulas and Models.

Intuitively, the vocabulary tells us the language the conversation is going to be conducted in. It tells us *in what terms* we will be able to talk about things. To be a bit more precise:

1. The first set in a vocabulary 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 MARY, JOHN, ANNA, and 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 entitities, 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 gives us all the information needed to define the class of models of interest (that means the kinds of situations we want to describe) and the relevant first-order language (that means the kinds of descriptions we can use).

Obviously, to really understand all of this we need to know something about first order models and how they're used to interpret formulae. So let's next have a look at what first-order models and languages actually are.

1.1.2 First-Order Models

Let's suppose we've established a vocabulary. What would a *first-order model* for this vocabulary be?

If you read it again thoroughly, our previous discussion pretty much contains the answer to this: Intuitively, a model is a situation. A situation is a *semantic* entity, providing us with a certain amount of things we can talk about. Thus, a model for a given vocabulary gives us two pieces of information. First, it tells us what kind of collection of entities (usually called the *domain* , or D for short) we can talk about. Secondly, for each symbol in the vocabulary, it gives us an appropriate semantic entity, built from the items in D, this task being carried out by a function F which, for each symbol in the vocabulary, specifies an appropriate semantic value. A function like this is what we call *interpretation function* .

What is a Model?

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

What are *appropriate* semantic values? There's no mystery here. Since constants are names, each constant should be interpreted as *an element of* D. (That is, for each constant symbol c in the vocabulary, $F(c) \in D$.) 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.)

1.1.3 An Example Model

Let's look at an example. We shall define a simple model using the vocabulary given above (page 5). 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.

1.1.4 Exact Models

What is an Exact Model?

Here's a second model for our vocabulary (page 5). We'll use the same domain (that is, $D = \{d_1, d_2, d_3, d_4\}$) but change the interpretation function. To emphasize that the interpretation function has been altered, we'll use a different symbol (namely F_2) for it.

$$
\begin{aligned}
&F_2(\text{MARY}) = d_2\\
&F_2(\text{ANNA}) = d_1\\
&F_2(\text{JOHN}) = d_4\\
&F_2(\text{PETER}) = d_3\\
&F_2(\text{THERAPIST}) = \{d_1, d_2, d_4\}\\
&F_2(\text{MORON}) = \{d_3\}\\
&F_2(\text{LOVE}) = \{(d_3, d_4)\}
\end{aligned}
$$

Exercise 1.1 *Give a (natural language) description of this model.*

It's important to mention one special trait both of the models we have defined show: *every entity in D exclusively corresponds to one constant*. From now on, we will call any model with this specific an *exact model* .

An Inexact Model

While exact models are particularly easy to work with, it is vital to realize that by no means every model is an exact one. Just suppose we add two more entities to the domain of the first model; thus creating a new domain $D' = D \cup \{d_5, d_6\}$, and furthermore define a new interpretation function F' on D' as follows:

$$
\begin{aligned}
&F'(\text{MARY}) = d_2\\
&F'(\text{ANNA}) = d_1\\
&F'(\text{JOHN}) = d_4\\
&F'(\text{PETER}) = d_3\\
&F'(\text{THERAPIST}) = \{d_1, d_2, d_4, d_5\}\\
&F'(\text{MORON}) = \{d_3, d_6\}\\
&F'(\text{LOVE}) = \{(d_3, d_4), (d_6, d_5)\}
\end{aligned}
$$

This new model is similar to our second model, the difference being that it has two extra entities. Neither of these new entities has a name, but we can see that one of them is a therapist, while the other one is a moron, and that the unnamed moron loves the nameless therapist. This is *still* a perfectly good first-order model, for the following two reasons:

- There is no requirement that every entity in the model must have a name (we only bother to name entities of special interest).
- There is no requirement that each entity in a model must be named by exactly one constant.

For example, if we wanted to we could assign both the names John and Peter to the same entity (for example if we wished that entity to have a real name and a nickname).

1.1.5 First-Order Languages

We now understand what a vocabulary is, and we've learned about models, the semantic entities corresponding to vocabularies. It's time now to turn to the notion of *first-order formula* e. First-order formulae are derived from *first-order languages*. And 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 $x, y, z, w, \ldots$, 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 common to all first-order languages: the only aspect that distinguishes first-order languages from one another is the choice of non-logical symbols (that is, of vocabulary).

1.1.6 Building Formulae

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 Formulae.

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

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

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

1. All atomic formulae are wffs.
2. If φ and ψ are wffs then so are $\neg\varphi$, $(\varphi \to \psi)$, $(\varphi \vee \psi)$, 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, formulae built using $\neg$, $\rightarrow$, $\vee$ and $\wedge$ correspond to the natural language expressions 'it is not the case that ...', 'if ... then ...', '... or ...', and '... and ...', respectively. First-order formulae of the form $\exists x\varphi$ and $\forall x\varphi$ correspond to natural language expressions of the form 'some...' or 'all...'. We shall soon (page 12) use first-order models to give a more precise formulation of these intuitive correspondences.

Literals.

Finally, a widely used class of formulae are the so-called *literal* s which include atomic formulae and *negated* atomic formulae. Literals can (in contrast to atomic formulae) also be used to express the fact that Peter does *not* love Anna ($\neg\text{LOVE}(\text{PETER}, \text{ANNA})$).

1.1.7 Subformulae, Free Variables

Subformulae.

In what follows, we occasionally use the term *subformula* . As an explanation, the subformulae of any formula φ are φ itself as well as all of the formulae that are used to build φ. For example, the subformulae of

$$\neg\forall y\text{PERSON}(y)$$

are $\text{PERSON}(y)$, $\forall y\text{PERSON}(y)$, and $\neg\forall y\text{PERSON}(y)$.

Exercise 1.2 *Give an inductive definition of subformulahood.*

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 lok 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 $\forall$. Here's the full definition:

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* .

1.1.8 Free Variables versus Bound Variables

Although they are both called variables, free and bound variables are in reality two very different things. (In fact, some formulations of first-order logic even use two distinct kinds of symbol for what we have lumped together under the heading 'variable'.) As an analogy, try thinking of a free variable as something like the pronoun 'she' in the sentence 'She even has a stud in her tongue'. Uttered in isolation, this would be somewhat puzzling, as we don't know whom 'she' refers to. Normally though, such an utterance would be made in an appropriate *context*, this context being either a non-linguistic one (imagine, for example, the speaker pointing towards a heavily tattooed biker, in which case we would say that 'she' was being used *deictically* or *demonstratively*) or a linguistic one (for example with the preceding sentence being 'Anna is heavily into body piercing', in which case the name 'Anna' would supply a suitable anchor for an *anaphoric* interpretation of 'she').

The point of this analogy is: Just as the pronoun 'she' requires something else as a complement (namely, contextual information) in order to supply a suitable referent, formulae containing free variables will require additional information on how to link the free variables to the entities in the model. So just supplying a model isn't sufficient.

Sentences, on the other hand, are relatively self-contained. For example, consider the sentence $\forall x\text{MORON}(x)$. This sentence claims that *every* individual is a moron. Roughly speaking, the bound variable x in $\text{MORON}(x)$ acts as a kind of placeholder. In fact, the use of x is completely arbitrary; the sentence $\forall y\text{MORON}(y)$ would mean exactly the same thing. Both sentences are simply a way of stating that no matter what we take the second occurrence of x (or y) as standing for, that entity will invariably be a moron. In any model of appropriate vocabulary this sentence (and in fact *any* sentence over that vocabulary) will either be true or false.

Our discussion of the interpretation of first-order languages in first-order models will further clarify these distinctions (indeed, most of the actual work involved in interpreting first-order logic focuses on the correct handling of free and bound variables).

1.1.9 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.

$$(\text{THERAPIST}(\text{JOHN}) \wedge \text{MORON}(\text{PETER}))$$

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

$$\text{THERAPIST}(\text{JOHN}) \wedge \text{MORON}(\text{PETER})$$

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

$$\forall x((\neg\text{THERAPIST}(x)) \wedge \text{MORON}(x) \rightarrow \text{MORON}(x))$$

is shorthand for the following:

$$\forall x((\neg\text{THERAPIST}(x) \wedge \text{MORON}(x)) \rightarrow \text{MORON}(x))$$

In addition, we'll at the same time 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 Semantic Notions

1.2.1 Satisfaction

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 defined 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 to variables in $\mathbf{M}$ (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 the 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*.

1.2.2 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 such that $y \neq x$, $g'(y) = g(y)$ 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.

1.2.3 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 to variables in $\mathbf{M}$. 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.

1.2.4 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 $\mathbf{M}$ if and only if for *any* assignment g of values to variables in $\mathbf{M}$, we have that:

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

If φ is true in $\mathbf{M}$ 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 asked to do so in !!!UNEXPECTED PTR TO EX_EX.FOL.EX.FREEVARS!!!.

But for all the elegance of the truth definition, it's still satisfaction that is the fundamental concept. Not only is satisfaction the technical engine powering the definition of truth, but from the perspective of natural language semantics it is conceptually prior as well. By rendering *explicit* the role of variable assignments, it holds up an (admittedly imperfect) mirror to the process of evaluating descriptions in situations while making use of contextual information.

In !!!UNEXPECTED PTR TO EX_EX.FOL.EX.FREEVARS_CONSTANTS!!! you're asked to examine the relation between free variables and constants.

1.2.5 Validities

Later in this course we are going to make extensive use of *logical inference*, and by making use of the semantic concepts just introduced we can explain what we mean by this. This will be done in two separate steps. First we'll establish what *valid formulae* (or more simply, *validities*) are, then we'll give a definition of *valid arguments* (or *valid inferences*).

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 and a context in which φ would not be satisfied. We indicate that a formula φ is valid by writing $\models \varphi$.

For example:

$$\models (\text{MORON}(x) \vee \neg \text{MORON}(\text{X}))$$

In any model, given any variable assignment, one (and indeed, only one) of the two disjuncts must 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:

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

!!!UNEXPECTED PTR TO EX_VALIDITY!!! shows that the validity of arbitrary formulae is equivalent to the validity of certain sentences.

1.2.6 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. In a nutshell: the truth of the premises guarantees the truth of the conclusion.

Proof Theory.

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). The subject of *proof theory* is to define them in terms of syntax. We will cover this in Chapter 7.

Proof Systems

Syntactic calculi for inference may even be employed in computational implementation: Various *proof systems* have been developed, and we will see in in this course how the syntactic method of *first order tableaux* can be implemented in Prolog.

1.2.7 An Example

The argument with premises $\forall x(\text{MORON}(x) \to \text{THERAPIST}(x))$ and $\text{MORON}(\text{MARY})$ and the conclusion $\text{THERAPIST}(\text{MARY})$ is valid. That is,

$$\forall x(\text{MORON}(x) \to \text{THERAPIST}(x)), \text{MORON}(\text{MARY}) \models \text{THERAPIST}(\text{MARY}).$$

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

Deduction Theorem.

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

The example suggests that with the help of the Boolean connectives $\wedge$ and $\to$ we can convert valid arguments into validities. This is exactly what's stated by the *deduction theorem* . The reader is asked to explore this possibility in !!!UNEXPECTED PTR TO EX_FOL.EX.DEDTHEOREM!!! and !!!UNEXPECTED PTR TO EX_EX.FOL.EX.ARG_FREEVAR!!!.

1.3 Equality

1.3.1 Equality Symbol

The first-order languages we have defined so far have an obvious expressive shortcoming: we have no way of asserting that two terms denote the same entity. Still, we may sometimes want to express such identities. So what are we to do?

Actually, the solution is perfectly straightforward. We can turn any language of first-order logic into a first-order language *with equality* by adding the special two place relation symbol $=$. We use this relation symbol in the natural infix way: that is, if τ_1 and τ_2 are terms then we write $\tau_1 = \tau_2$ rather than the unsightly $=(\tau_1, \tau_2)$.

1.3.2 Semantics of Equality

Beyond the convention of writing $=$ in infix notation there isn't much more to say about the syntax of the equality symbol, other than the fact that it's just a two place relation symbol. But what about its semantics?

This is where matters become more interesting. Although, syntactically, $=$ is just a 2-place relation symbol, it is a very particular one in this respect. In fact, (unlike LOVE, or HATE, or any other two place relation symbol) we are *not* free to interpret it as we please. Thus, given any model $\mathbf{M}$, any assignment g in $\mathbf{M}$, and any terms τ_1 and τ_2, we shall insist that

$$\mathbf{M}, g \models \tau_1 = \tau_2 \text{ iff } I_F^g(\tau_1) = I_F^g(\tau_2).$$

Normal Models.

That is, the atomic formula $\tau_1 = \tau_2$ is satisfied if and only if τ_1 and τ_2 have exactly the same interpretation. Therefore, $=$ really means equality in this model $\mathbf{M}$. A model that fullfill this requirement is called *normal model* .

In fact, $=$ is often regarded as a *logical* symbol on a par with $\neg$ or $\forall$, for in *normal* models it has, just like those symbols, a fixed interpretation, and a semantically fundamental one at that.

1.4 Exercises

Exercise 1.3 *We've claimed that when evaluating* sentences*, it doesn't matter which variable assignment we start with. Formally, this means that given any* sentence φ *and any model* $\mathbf{M}$ *(of the same vocabulary), and any variable assignments* g *and* g' *in* $\mathbf{M}$*, then* $\mathbf{M}, g \models \varphi$ *iff* $\mathbf{M}, g' \models \varphi$*. Now, we would like you to do two things:*

First, show that the above claim is false *if* φ *is not a sentence but a formula containing free variables.*

Secondly, show that the claim is true *if* φ *is a* sentence*.*

Exercise 1.4 *This exercise shows that free variables and constants are very similar. In particular, if a free variable* x *and a constant* c *denote the same individual, we can even replace the free variable by the constant without affecting* satisfiability*.*

Formally, let $\mathbf{M} = (D, F)$ *be a model, let* g *be an assignment in* $\mathbf{M}$*, and suppose that* $F(c) = g(x)$*. Let* φ *be any formula, and let* $\varphi[c/x]$ *denote the formula obtained by replacing all free occurrences of* x *in* φ *by* c*. Then* $\mathbf{M}, g \models \varphi$ *iff* $\mathbf{M}, g' \models \varphi[c/x]$*, where* g' *is any* x*-variant of* g*.*

It follows that when working with exact *models, every formula is equivalent to a sentence.* Explain why.

Exercise 1.5 *This exercise shows that the* validity *of arbitrary formulae is equivalent to the validity of certain sentences. As a first step, we would like the reader to prove that if* φ *is a formula containing* x *as a free variable, then* φ *is valid iff* $\forall x\varphi$ *is valid.*

It follows that the validity of formulae is reducible to the validity of sentences. Explain why. *[Hint: we've just found a way of reducing the number of free variables by one while maintaining validity. Iterate this process.]*

Exercise 1.6 *The* Deduction Theorem *for first-order logic states that* $\varphi_1, \ldots, \varphi_n \models \psi$ *if and only if* $\models (\varphi_1 \wedge \cdots \wedge \varphi_n) \rightarrow \psi$*. (That is, there is a close link between validities and valid arguments.)* Prove the Deduction Theorem*.*

Exercise 1.7 *Show that the validity of arguments whose premises or conclusions contain free variables is reducible to the validity of arguments whose premises and conclusions are sentences. [Hint: think about !!!UNEXPECTED PTR TO EX_FOL.EX.DEDTHEOREM!!!.]*

2

Prolog and First-Order Logic

2.1 A Simple Model Checker

2.1.1 Representing Vocabularies

Let us first turn to the representation of vocabularies. Recall that a (page 5) specifies which (non-logical) symbols can be used, and how they can be used. We will soon see that having this information available greatly simplifies the way we can specify models. A vocabulary specifies all the constant and predicate symbols that can occur in formulae, and specifies an arity for each predicate symbol. We will simply assert terms to the database that record this information:

This is what the representation of our example vocabulary (page 5) will look like:

```
const(mary).

const(john).

const(anna).

const(peter).

pred(therapist,1).

pred(love,2).

pred(hate,2).
```

See file **`signature.pl`.**

This code can be found in `signature.pl` (along with some other constants, predicate symbols, and some stuff we will need later).

2.1.2 Representing Simple Formulae

Let us now decide how to represent first-order formulae in Prolog. But before that, there's one thing we still have to say: How do we represent first-order variables? We make the following choice:

Variables.

First-order *variables* will be represented by distinct Prolog atoms. To distinguish them from constant and predicate symbols, we will assert a term of the form

```
fovar(x).
```

to the database for each variable `x`, along with the vocabulary (maybe `var/1` would have been a more obvious choice of name, but that would be in conflict with Prolog's built-in predicate named `var/1`). The variables `x,y,z,u,v,w` are defined in `signature.pl` and can be used right away.

Atomic Formulae.

Next, we must decide how to represent *constant* and *predicate symbols*. We do so in the obvious way: 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`.

Given this convention, it is obvious how *atomic formulae* should be represented. For example, LOVE(JOHN,MARY) would be represented by the Prolog term `love(john,mary)`, and HATE(PETER,x) would be represented by `hate(peter,x)`.

2.1.3 Representing Complex Formulae

Next for Boolean combinations of simple formulae. The symbols

Connectives as Operators.

```
&       v       >       ~
```

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

See file `comsemOperators.pl`.

The following Prolog code from `comsemOperators.pl` ensures that these connectives have their usual precedences:

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

:- op(850,yfx,v).          % disjunction

:- op(900,yfx,>).          % implication

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

?- Question!

Have a look at *Learn Prolog Now!*[1] if you are unsure about what this code means. To test your understanding: How would the following 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 the quantifiers. Take for example the first order formula $\text{MAN}(x)$. `man(x)` is its representation as a Prolog term. Now $\forall x\text{MAN}(x)$ will be represented as

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

and $\exists x\text{MAN}(x)$ will be represented as

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

Remember that the fact `fovar(x)` has to be in the database because we want `x` to stand for a variable.

!!!UNEXPECTED PTR TO EX_EX.WFF!!!, !!!UNEXPECTED PTR TO EX_EX.REVERSIBLE_WFF!!! and !!!UNEXPECTED PTR TO EX_EX.WFFLIST!!! deal with checking the well-formedness (page 9) of formulae in Prolog representation, and with the relation between vocabularies and formulae.

2.1.4 Representing Models

Let's suppose that we have fixed our vocabulary. For example, suppose we've decided to work with the vocabulary specified before (page **??**) and have it in our database. How should we represent models of this vocabulary in Prolog?

We only use exact models.

In this general formulation, this is an impossibly difficult question. We don't have the remotest chance of dealing with *all* models of this vocabulary, for there are models of this vocabulary based on *any* non-empty domain D whatsoever. In particular, there are lots of *infinite* models. Now, it is possible to give useful finite representations of some infinite models - but most are too big and too unruly to be worked with computationally. Hence we shall confine our attention to *finite* models of our vocabulary (i.e. ones with a finite domain). In addition, matters are much simpler if we only have to deal with *exact models* (page 7).

Recall that exact models are models where every element of the domain is named by *exactly one* constant. So, let's rephrase our question: How should we represent a finite exact model over the above vocabulary?

[1] `http://www.learnprolognow.org`

The vocabulary fixes the domain.

Because of the one-to-one correspondence between constants and domain elements in exact models, our vocabulary already tells us all we need to know about the domain of the model to be represented: It tells us how many elements there will be in its domain. In fact we shall simply assume that the domains of the models we represent consist of the constants in our vocabulary. Thus given our example vocabulary (page **??**), the models we speak of in the following will invariably have the domain $\{\text{MARY}, \text{JOHN}, \text{ANNA}, \text{PETER}\}$.

Herbrand Models

Such models, where the domain contains the constant symbols of the language in use, are called Herbrand models. Thus we not only restrict ourselves to considering exact models in the following, but even to the more narrow class of Herbrand models. Luckily, Herbrand models are enough in a very strong sense. In short, even if we are interested in a model with a fixed number of items of *any* kind (anything different from constant symbols; say, four human beings) in its domain, there will be an isomorphic model with a domain that instead consists of the same number of constant symbols (say MARY, PETER, JOHN and ANNA), and this model will do the same job at satisfying any formulae. We will encounter the concept of a Herbrand model again (and give it a formal definition) in Section 9.1.3.

Listing positive facts.

So we use the vocabulary (asserted to the database, see Section 2.1.1) to represent the domain of all models we want to speak about. Hence all we need to do in order to represent a model is to give an exhaustive listing of all the atomic facts that hold in it. And these facts can all be written as atomic formulae without variables. We can then simply assume all other atomic facts (i.e. all other variable-free atomic formulae licensed by our vocabulary) to be false in the represented model. Besides saying that such a listing of atomic formulae *represents* a model, we will also say that it *specifies* that model or call it a (model) *specification* .

?- Question!

Why don't we allow atomic formulae with variables into model specifications? Hint: Think about how variables are interpreted (see Section 1.2.2).

Now here is an example of how to represent a model in Prolog:

```
[therapist(john), therapist(mary), moron(peter), moron(anna), love(peter,anna)]
```

Fairly obviously, this list represents a model in which both John and Mary are therapists, both Peter and Anna are morons, and Peter loves Anna.

Negative facts are implicit

But it is important to be aware that other properties are *implicitly* recorded. For example:

- No facts about the *hate* relationships are given-and `hate` is one of the predicate symbols in our vocabulary. This is Prolog's way of recording the fact that the binary predicate *hate* is empty; in this little world, nobody hates anybody.
- Although Peter loves Anna, he doesn't love himself nor anybody else, for no such facts are recorded. In fact, no other loving relationships at all hold.

Thus, given our vocabulary we have completely described a unique exact model over it by explicitly listing positive information and *implicitly* listing negative information.

?- Question!

Which of the following are correct Prolog model specifications (over the example vocabulary) according to our conventions:

1. [therapist(john), therapist(mary), moron(peter), moron(anna), hate(peter,anna)]
2. [therapist(john), therapist(mary), ~moron(peter), moron(anna), love(peter,anna)]
3. [therapist(john)>therapist(mary), moron(peter), boring(anna), love(peter,anna)]
4. [therapist(mary), moron(x), moron(anna), love(peter,anna)]
5. [therapist(mary), moron(mary),love(mary,mary)]

See file `exampleModels.pl`.

The file `exampleModels.pl` contains the representations of some models (over our vocabulary). The models are numbered for easy access.

2.1.5 Another Example

Here's another example of a model represented in Prolog, again over the same vocabulary.

```
[therapist(peter),moron(anna),moron(john),love(peter,anna), hate(john,peter)]
```

Now for a trick question: how many individuals are there in the domain of the model that this Prolog term represents? The answer is *four*. If you thought the answer was three-because only John, Peter and Anna are explicitly mentioned-you haven't properly grasped the role played by the constants listed in the vocabulary. In *any* exact model of this vocabulary there is the individual corresponding to the constant MARY.

Again: Negative facts are implicit

As it happens, in this particular model no positive facts are listed about MARY-but lots of negative facts about Mary are implicitly given. For example:

- MARY is neither a therapist nor a moron.
- MARY neither loves nor hates anyone or anything.
- MARY is not loved or hated by anyone or anything.

Summing Up.

In short, be careful. The listing of constants in the vocabulary plays an important role for us: it tells us exactly how many (and in a way also which) individuals there are in an exact model of that vocabulary. For this, it doesn't matter if anything positive is mentioned about each of them or not.

In !!!UNEXPECTED PTR TO EX_EX.SETDESCR_MODEL!!! you can test if you understand how our way of representing models works. !!!UNEXPECTED PTR TO EX_EX.WELLFORMED_MODELS!!! asks you to implement well-formedness checks for representations of models.

2.1.6 Semantic Evaluation

We now turn to the fourth and final task: Designing the actual model checker program, which evaluates (representations of) formulae in (representations of) models. That is, our program should say `yes` if the input formula is true in the given model and `no` if it isn't. E.g. given our first example model (page **??**) and the formula `therapist(john) & moron(peter)`, the program should answer `yes`. In contrast, evaluating `therapist(john) & therapist(peter)` over the same model should lead to the answer `no`.

We now write a predicate called `eval/2` to carry out this task. It consists of seven clauses and takes two arguments: First the formula to be tested, and second the model in our list representation. As discussed above (page 22) we assume that the vocabulary can be found in the database.

See file `modelChecker.pl`.

The clauses of `eval/2` and (a driver predicate `evaluate/2`, see Section 2.1.9) can be found in `modelChecker.pl`.

Evaluating Atomic Formulas

First for the core of our little model checker. The clause that evaluates atomic formulae. It couldn't be much simpler. We have decided to represent an exact model by recording all the facts (in the form of atomic formulae) that are true in it. Hence an atomic formula is true if and only if it is one of the facts recorded in the list `Model`. So all we need to do is search through our list:

```
eval(Formula,Model):-
   member(Formula,Model).
```

2.1.7 Evaluating Complex Formulae

Now let's have a look at the clauses for Boolean combinations of formulae. First for the binary connectives:

```
eval(Formula1 & Formula2,Model):-
   eval(Formula1,Model),
   eval(Formula2,Model).

eval(Formula1 v Formula2,Model):-
   eval(Formula1,Model);
   eval(Formula2,Model).

eval(Formula1 > Formula2,Model):-
   eval(Formula2,Model);
   \+ eval(Formula1,Model).
```

Truth Conditions in Prolog.

These clauses mirror the first-order satisfaction definition (page 13) (on which the definition of truth in a model (page 13) is based) in an obvious way, using the Prolog versions of conjunction, disjunction and negation. For example, the clause for > translates to Prolog that an implication is true iff:

- the consequent is true, or
- the antecedent is false.

Negation and Prolog Negation.

We make use of Prolog negation (that is, negation as failure[2]) when we want to evaluate negated formulae. This is a simple and natural thing to do. However we shall see soon that it can have unexpected - and unintended - effects. Here's the code:

```
eval(~Formula,Model):-
   \+ eval(Formula,Model).
```

2.1.8 Quantifiers

∃

The clause for the existential quantifier also quite directly implements the respective part of the satisfaction definition. Our program picks one entity from the model and instantiates the quantified variable X to it.:

```
eval(exists(X,Formula),Model):-
   const(C),
   subst([X:C],Formula,Instantiated),
   eval(Instantiated,Model).
```

[2] http://learnprolognow.org

Instantiating the Quantified Variable.

Now because we work with exact models and assume our universe to consist of constants, to pick one entity simply means to access the vocabulary and choose a constant.

See file `substitute.pl`.

For this task we use the predicate `subst/3` from `substitute.pl`. It takes a substitution (of the form `[Substitute1:By1,Substitute2:By2...]`) and applies it to the Formula given as second argument. The third arguments contains the result. The code for `subst/3` is a bit more involved than you might expect, because it implements full first-order substitution . For the case at hand, though, it simply recurses down to literals, splits them and finally replaces the arguments according to the given substitution.

Backtracking on Instantiations.

Then, our program tests if the now instantiated `Formula` is true in the given model. If this is not the case, the program backtracks and tries another constant. Either it eventually succeeds. This means that our existential quantification is true because there exists an entity that supports the scope (after all our program has found one such entity). Or it doesn't succeed, which means that the existential quantification is false.

∀

The clause for the universal quantifier takes advantage of a logical equivalence: $\forall X\varphi$ is equivalent to $\neg\exists X\neg\varphi$. So instead of `forall(X, Formula)` we can equivalently consider `~exists(X, ~Formula)`. This is what happens in the following clause.

```
eval(forall(X,Formula),Model):-
   eval(~ exists(X,~ Formula),Model).
```

If you like to, you can prove the equivalence used by this clause as !!!UNEXPECTED PTR TO EX_EX.EQUIV_FA_EX!!!

2.1.9 Checking Models

See file `exampleModels.pl`.

So, it's time to start checking models. Some examples of models over our original vocabulary can be found in the file `exampleModels.pl`. Each example model is assigned a number, so we can easily select any of them.

And here is a driver predicate which evaluates a formula in an example model (selected via its number):

See file `modelChecker.pl`.

```
evaluate(Formula,ExampleNumber):-
   example(ExampleNumber,Model),
   eval(Formula,Model).
```

An example query looks like this: `evaluate((love(peter,anna) & moron(anna)),1).` Systematically test the model checker on our example models (download the files linked in Section 2.1.6). Are the results always as you expected? If not, why not?

2.2 Refinements

2.2.1 Problem One: Unknown Vocabulary

What happens if our input uses symbols that are not in our vocabulary? We run into problems if we ask our model checker to verify formulae that use completely new non-logical vocabulary.

For example, there's no constant symbol `cinderella` in our vocabulary. Let's try to evaluate the formula $\forall x \text{THERAPIST(X)} \wedge \neg \text{THERAPIST(CINDERELLA)}$, say in model 5. The formula states that 'Everyone is a therapist, but Cinderella isn't.' Clearly, this is contradictory. So whenever well-formed, this formula is unsatisfiable. It should thus never come out as true, no matter in which model we choose to evaluate it. So let's query `evaluate(forall(x,therapist(x)) & ~therapist(cinderella), 5).` Surprisingly the answer is `yes`. How could this happen?

Let us have a short look at a simpler example. Let's call `evaluate(therapist(cinderella),5).` Our model checker says `no`. This response, too, is *not* strictly speaking correct. Formally the satisfaction relation (and thus truth) is not defined for formulae and models of different vocabularies. The correct response of our model checker would be to reject this formula and say something like "Hey, I don't know anything about the symbol `cinderella`!".

This little flaw may seem harmless at first glance, but the same problem leads to the false positive we've seen above. Think about what our model checker does when it evaluates `forall(x,therapist(x)) & ~therapist(cinderella)` in a given model: First it checks if `forall(x,therapist(x))` is true in our model. (Remember the Prolog clause for universal quantification: The model checker will check that `~exists(x, ~therapist(x))` is true, by making sure that `therapist(c)` isn't false for any of the `const(c)` in the database). In model 5, this is the case. Then, our program proceeds to the second conjunct. There it finds the negation `~therapist(cinderella)`, which it evaluates using Prolog's `\+`: `~therapist(cinderella)` will become true if evaluating `therapist(cinderella)` fails. And in fact evaluating `therapist(cinderella)` does fail. Yet this not due to the situation in our model, but simply because `cinderella` is not in the vocabulary: `therapist(cinderella)` is *not* one of the facts that don't occur on our list because they are false. Rather, it does not come into consideration as a positive or as a negative fact at all, because it is not well-formed.

The moral is that the use of negation as failure may turn a somewhat sloppy `no` (like the answer of our model checker to unknown vocabulary) into a dangerously clear `yes`. So we really should take care that `no` really only means 'not true'.

Luckily the problem is not particularly deep; the solution is simply to be explicit about what a well formed formula over a given vocabulary is. That is, we should make use of the information in the vocabulary about which predicate symbols we are working with (and, of course, what their arities are), which constants we have, and explicitly state which combinations of these symbols are well formed. Our driver should check that any sequence of symbols it has to evaluate really is a well formed combination of the chosen predicate and constant symbols - and it should refuse to evaluate anything that isn't.

2.2.2 Problem Two: Formulae with Free Variables

We run into similar trouble if we feed our model checker formulae with free variables (i.e. ones that are not bound by a quantifier). For example `evaluate(therapist(x),1).` Our model checker answers `no`, although formally, truth in a model (page 13) is only defined for sentences, i.e. formulae with *no* free variables. There are ways of extending this definition to formulae with free variables (see !!!UNEXPECTED PTR TO EX_PROJ.ASSIGNMENTS!!!). But since we have not chosen any of these extensions, we needn't implement them. We will simply refuse non-sentences, just as we will refuse non-well-formed formulae.

2.2.3 Refining the Implementation

***See file* `revisedModelChecker.pl`.**

So let's re-visit our implementation and modify it according to our new insights. The simplest we can do is to modify our driver predicate as follows:

```
evaluate(Formula,ExampleNumber):-
   example(ExampleNumber,Model),
   wff(Formula),            % Implement as an exercise
   free(Formula,[]),        % Imported from module substitute
   eval(Formula,Model).


eval(Formula,Model):-
   member(Formula,Model).
```

In this piece of code `wff/1` is a predicate that checks whether a given formula is well-formed over the vocabulary in the database. You should implement this predicate as an exercise (page 29).

***See file* `substitute.pl`.**

The call `free(Formula,[])` solves our second problem: It enforces that the input `Formula` contains no free variables, hence that it really is a sentence. The predicate `free/2` (coming from `substitute.pl`) produces the list of unbound variables in the formula given as first argument. Hence calling it with an empty list as second argument makes sure that there are no free variables in the formula given as first argument.

We said before that we have to make sure that `no` really only means 'not true'. But up to now, we have not done so. We have only moved the `no` meaning 'unexpected input' to the beginning of our predicate, where it cannot interfere with the `no` meaning 'not true'. However it is now a simple task to exit with a message on unexpected input before the actual model checking is started. !!!UNEXPECTED PTR TO EX_EX.REFUSE_NONWFF!!! asks you to implement this little error-handling capability.

In `revisedModelChecker.pl` you find an incomplete implementation of the revised model checker: The (unchanged) clauses of `eval/2`, `rev_evaluate/2` as discussed above, and a dummy predicate `wff/1` that you can replace by your implementation from !!!UNEXPECTED PTR TO EX_EX.WFF!!!.

2.3 File Listing

2.3.1 All modules for the model checker

See file `modelChecker.pl`.	The driver predicate `evaluate/2` and the core clauses of the
See file `comsemOperators.pl`.	Operator definitions.
See file `revisedModelChecker.pl`.	The revised version of the model checker (excluding `wff/1`,
See file `signature.pl`.	Example vocabulary.
See file `exampleModels.pl`.	Some example models to play with.
See file `comsemLib.pl`.	Various helpers, `compose/3` may be useful in exercises.

2.4 Exercises

Exercise 2.1 *[Theoretical Exercise]*

Give the set theoretic description (page 6) of the models that the Prolog lists in Section 2.1.4 and Section 2.1.5 represent.

Exercise 2.2 *Write a predicate* `wff/1`*. It should succeed if and only if its first argument represents a well formed formula (page 9) over the vocabulary in the database. Thus given our example vocabulary See file* `signature.pl`*.,*

```
wff(exists(x, moron(x) & therapist(mary)))
```

should succeed, while

```
wff(hate(peter,cinderella)).
```

should fail, as should:

```
wff(love(mary)).
```

***See file* `comsemLib.pl`.**

[Hint: Use `=..` *or* `compose/3` *(page 201) from* `comsemLib.pl` *to split terms into functor and arguments.]*

Does your predicate (correctly) require quantifiers to bind variables*, i.e. does it rule out formulae like* `exists(mary,moron(mary))`*? If it doesn't, try to repair it. What does your predicate do if you replace subformulae in its input by Prolog-variables (e.g. if you check* `moron(mary)& X`*)?*

Use your predicate to complete the revised version of our model checker.

Exercise 2.3 *Can your predicate* `wff/1` *be used to generate formulae over a vocabulary by leaving the first argument uninstantiated? Explain your observations.*

Exercise 2.4 *Write a predicate* `wffList/1` *that takes a* list *of formulae and succeeds if and only if all formulae on the list are well formed over the vocabulary in the database.*

Exercise 2.5 *We did not do anything to make sure that first order variables and constant/predicate-symbols are really distinct. Write a predicate that looks at the database and gives a warning if any first order-variable is also a constant/predicate-symbol.*

Exercise 2.6 *Write a Prolog program which when given a list of terms, determines whether or not the list represents a model. That is, your program should check whether:*

- *Its input is a list.*
- *All members of this list are well-formed atomic formulae over the given vocabulary.*
- *None of these atomic formulae contains a free variable.*

If you've solved !!!UNEXPECTED PTR TO EX_EX.WFF!!! or !!!UNEXPECTED PTR TO EX_EX.WFFLIST!!!, you may be able to reuse parts of the code with minor changes.

Exercise 2.7 *[Theoretical Exercise (OPTIONAL)]*

We say that two sentences φ *and* ψ *are* logically equivalent *if and only if* $\varphi \models \psi$ *and* $\psi \models \varphi$. *Show that* $\forall x\varphi$ *and* $\neg\exists x\neg\varphi$ *are logically equivalent.*

Exercise 2.8 *Go through Section 2.1.9. Run the model checker and use* `trace/0` *to have a look at what the program really does. (The use of* `trace/0` *is discussed in Practical Session 2 of Learn Prolog Now![3]).*

What does the first version of the model checker do if its input formula contains Prolog-variables? Compare e.g. the following calls:

- `evaluate(X,1).`
- `evaluate(moron(peter) v X,1).`
- `evaluate(X & moron(mary),1).`

Obviously, the behaviour of the program isn't very consistent (in the last example, the model checker wouldn't come back at all on its own, but our web interface has a timeout of 3 seconds). Find out what's going on and why. Does it make a difference to use the revised `rev_evaluate/2` *instead of* `evaluate/2`*?*

We ask you to use the ILIAS newsgroup to discuss your findings from Section 2.1.9, as well as any problems you may have in understanding and using the model checker.

Exercise 2.9 *[For the code to work, you need a predicate* `wff/2`*. If you haven't solved !!!UNEXPECTED PTR TO EX_EX.WFF!!!, tell us[4] and we will send you the code you need.]*

[3] `http://www.learnprolognow.org`
[4] `mailto:stwa@coli.uni-sb.de`

If our revised model checker is called with an input that is not well-formed or that contains free variables, it will say `no`*. As we've discussed, this response is* not *yet what we'd like to have. Modify* `rev_evaluate` *so that it distinguishes between non well-formed input and well-formed input that is false in the given model. Non-wellformed input should produce a message and should not be evaluated.*

Exercise 2.10 *[This is a possible mid-term project]*

The definition of truth can be extended from sentences to arbitrary first-order formulae as follows: An arbitrary first-order formula φ *is true in a model* $\mathbf{M}$ *iff* there is *an assignment g in* $\mathbf{M}$ *such that* $\mathbf{M}, \mathbf{g} \models \varphi$*. Remember that for* sentences *we required that every assignment satisfied the sentence in the given model, but only to find out that assignments didn't matter for satisfying sentences anyway. This time the assignment* does *matter, because our formula may contain free variables. But we only require that* some *assignment do the job.*

Change the model checker so that it checks if an arbitrary first order formula is true in a model according to our new definition. You will probably have to add an argument that represents a (partial) variable assignment. This argument can then be used by all clauses of the model checker that deal with assigning or instantiating variables. You may use the predicate `subst/3` *(page 202) from See file* `substitute.pl`*..*

3

Lambda Calculus

3.1 Building Meaning Representations

3.1.1 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 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.

?- Question!

We have to form the first-order formula LOVE(JOHN,MARY) from the symbols LOVE, JOHN and MARY. But for this task, we haven't been specific yet about what we mean by working *in a systematic fashion*. For example, from the symbols LOVE, MARY and JOHN 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? Do you have an idea?

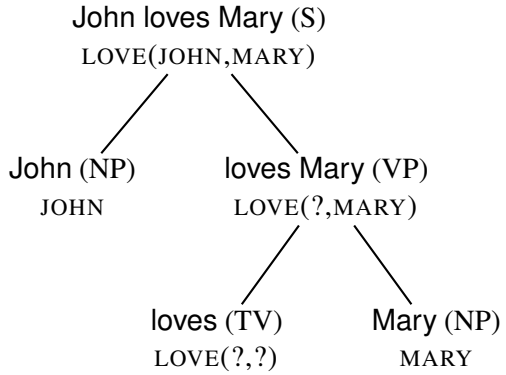
3.1.2 Being Systematic (II)

Syntactic Structure...

Our missing link here is the notion of *syntactic structure* . '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(?,?)). And why does JOHN need to be inserted into the *first* argument slot? 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.

3.1.3 [Sidetrack] Compositional Semantics

Our discussion has led us to one of the key concepts of contemporary semantic theory: *compositionality*.

The principle of compositionality

Compositionality is a simple and natural concept underlying most of the work in natural language semantics and in the semantics of programming languages. A common formulation of the principle of compositionality is the following: 'The meaning of a

compound expression is a function of the meaning of its parts and of the syntactic rule by which they are combined.' This at least requires that we have a notion of *parts* that form such an expression according to rules, and that each of these parts can be assigned a meaning.

Suppose we have some sort of theory of hierarchical syntactic structure providing us with a notion of *parts* of the expressions in our language. That is, the syntactic structures we would like to have should allow us to classify sentences into subparts, sub-subparts, and sub-sub-subparts, ..., and so on - ultimately into individual words (beyond that, it doesn't matter too much for our purposes what sort of syntactic theory it is). The syntactic structures provided by that theory then make it possible to tell an elegant story about where semantic representations come from.

Ultimately, semantic information flows from the lexicon, where each lexical item is associated with a representation. But how is this information combined? Here, we make use of the sentence hierarchy a syntactic analysis provides us with. Suppose the syntax tells us that some kind of sentential subpart (a VP, say) is decomposable into two sub-subparts (a TV and an NP, say). In that case it's our task to describe how the semantic representation of the VP subpart is to be built out of the representation of its two sub-subparts. If we succeed in doing this for all the grammatical constructions covered by the syntax, we'll have developed a *compositional semantics* for the corresponding language (or at least, for that fragment of the language covered by our syntactic analysis).

Why compositionality?

What is so nice about compositional semantics? First of all, the principle of compositionality is of great immediate appeal: The meaning of a sentence should somehow be derivable from that sentence itself. And two kinds of information that every sentence provides us with are its words and its structure. Moreover, in the course of compositional semantic construction, each of the syntactic subparts of a sentence gets assigned a meaning of its own at some point, and syntactic rules are correlated with actions on the semantic side. This may be seen as giving a semantic justification of syntactic structure.

More philosophically, the principle of compositionality offers an explanantion of how a human being can understand a possibly *infinite* number of sentences never heard before (namely by constructing their meaning from a *finite* set of rules and a *finite* set of known lexical meanings). Also, a compositional account of meaning suggests a plausible explanation of why we perceive a connection *in meaning* between sentences that share syntactic parts. Consider the sentences:

- 'John's father likes Mary.'
- 'John's father hates Mary.'

Clearly, these two sentences have meaning elements in common. Assuming a compostional semantics that is constructed part-by-part, along with the syntactic structure, this can be explained by pointing to their shared syntactic parts 'John's father' and 'Mary'.

Discussion

Of course, both of these arguments aren't really coercive. Satisfactory explanantions for the described phenomena could probably also be given on the basis of semantic construction methods that would not neccesarily be counted as compositional. In fact, in spite of its apparent simplicity, the notion of compositionality raises a number of interesting issues, and is by no means uncontroversial.

On the one hand, if we do not further constrain the class of functions that may be used to combine the meanings of the parts of expressions, the principle doesn't really say very much any more. It can even be shown that any semantic construction method can be made compositional in some sense. On the other hand, it can be doubted that *systematic* semantics always have to be compositional. There is for instance the standard top-down algorithm for Discourse Representation Theory (DRT). It is a systematic approach to semantic construction (under any reasonable interpretation of the word systematic) but it does not work by assigning meaning representations to each of the syntactic constituents of a sentence separately. Therefore many semanticists have argued that it is not truly compositional.

Apart from this, the principle of compositionality doesn't come with a fixed domain. One way to read it is in terms of meaning representations, as stating that we should have a separate meaning representation for each single syntactic part of an expression, and that the representation of the whole expression should be formed from these separate representations. This requirement is also known as surface- or *s-compositionality* . As another extreme, it can be read as referring to truth-functional content, e.g. as stating that we should be able to say exactly for each syntactic part how it is to be interpreted with respect to a given first-order model (one also speaks of deep or *d-compositionality*).

In the following, we shall stick to the first way of reading the principle of compositionality. In fact, we shall view it more or less as a practical design guideline than as a philosophical claim. This means that when constructing semantic representations, we will try to solve our problems locally and modularily, striving for a nice and clearly structured system. But occasionally, we shall also look at our semantic formalisms in view of d-compositionality.

3.1.4 Summing Up

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:

Three Tasks

Task1 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 (see also Section 3.1.3). That is, we need to *systematically* 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.

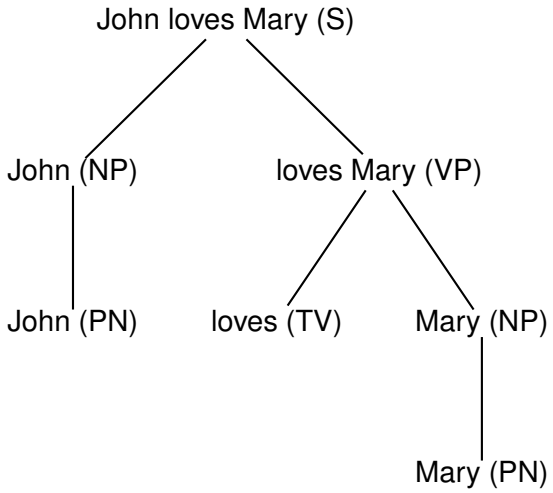
All of our three tasks need to be carried out in a way that naturally leads to computational implementation. Because this is a course on computational *semantics*, tasks 2 and 3 are where our real interests lie, and most of our attention will be devoted to them. But we also need a way of handling task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

3.2 Syntactic Analysis

3.2.1 A Simple Solution: CFG

Task 1 ✓

In order to approach Task 1, we have opted for an easy solution: We've decided to use a simple context free grammar. 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). Recall (the syntactic part of) the tree tasks from Section 3.1.4:



This tree model tells us that 'John loves Mary' is an S (sentence) that is composed of the subject NP (noun phrase) 'John' and the VP (verb phrase) 'loves Mary', which in turn consists of the TV (transitive verb) 'loves' and the NP 'Mary', which is made of the PN 'Mary', and so on. To enhance the readability of such trees, we will ommit the non-branching steps and take for instance Mary (NP) as a leave node.

Our approach to syntactic structure is not very elaborate. There are many interesting syntactic phenomena it won't allow us to describe. Hence, the approach won't take us far in terms of coverage of our grammar. But it has one important advantage: It enables us to make use of Definite Clause Grammars (DCGs, Lecture 7 of Learn Prolog Now!)) DCG[1], the in-built Prolog mechanism for grammar specification.

[1] `http://www.learnprolognow.org/`

3.2.2 Using DCGs

See file **`usingDCG.pl`.**

Here is a DCG for the fragment of English we shall use in our initial semantic construction experiments (this DCG can be found in the file `usingDCG.pl`).

A Context-free Grammar using DCG

```
s --> np, vp.


np --> pn.
np --> det, noun.


pn --> [john].
pn --> [mary].


det --> [a].
det --> [every].


noun --> [woman].
noun --> [siamese,cat].


vp --> iv.
vp --> tv, np.


iv --> [walks].


tv --> [loves].
```

This grammar tells us how to build certain kinds of sentences (`s`) out of noun phrases (`np`), verb phrases (`vp`), proper names (`pn`), determiners (`det`), nouns (`noun`), intransitive verbs (`iv`), and transitive verbs (`tv`). In addtion it gives us a tiny lexicon to play with.

An analysis of why this grammar accepts the simple sentence 'John walks' would be this: 'John' is declared a proper name, with proper names being noun phrases according to this grammar; 'walks' is an intransitive verb, and hence a verb phrase; and our grammar licenses sentences that consist of a noun phrase followed by a verb phrase.

Since DCGs are part and parcel of Prolog, we can actually compute with them. For example, by posing the query `s([mary,likes,a,siamese,cat],[])` we can test whether 'Mary likes a siamese cat' is accepted by the grammar, or we can use `s(X,[]),write(X),nl,fail` to generate all grammatical sentences.

?- Question!

How many sentences are accepted by this grammar? How many noun phrases, how many verb phrases does it allow? Check your answers by generating the relevant items.

3.3 Semantics Construction

3.3.1 A First Attempt

So how do we associate semantic representations with sentences using our little DCG?

Task 2 ✓

Let us first address the lexical items. We need to associate 'John' with the constant JOHN, 'Mary' with the constant MARY, 'walks' with the unary relation symbol WALK, and 'loves' with the binary relation symbol LOVE. The following piece of DCG code makes these associations. Note that the arities of `walk` and `love` are explicitly included as part of the Prolog representation.

```
pn(john)--> [john].

pn(mary)--> [mary].

iv(walk(_))--> [walks].

tv(love(_,_))--> [loves].
```

Task 3✓

How do we combine semantic representations? Here's a first (rather naive) attempt: Let's introduce a new symbol to mark all places where two representations are combined into one. For example, we can define a prolog operator `+` for this purpose.

?- Question!

Give the Prolog code needed to define such a `+` operator.

Let us incorporate this idea into our DCG. We'll do as we did for the lexical items, and supply additional arguments. For example, we get an `s` rule:

```
s(NPSem+VPSem) --> np(NPSem), vp(VPSem).
```

and a `vp` rule for transitive verbs:

```
vp(VSem+NPSem) -->  tv(VSem),np(NPSem).
```

Additionally, we will need the following rules that do not combine anything but rather lift the semantic representation of a lower category to that of a higher one:

```
np(X) --> pn(X).

vp(X) --> iv(X).
```

See file **`firstAttempt.pl`.**

With our newly augmented DCG, we are now able to construct semantic representations for the sentences we parse. For instance, parsing the sentence 'John loves Mary.' by posing the query `s(Sem,[john,loves,mary],[])` gives us the semantic representation `john+(love(_,_)+mary)`.

But is that what we want? No, not yet. What we want of course is to arrive at the first-order formula LOVE(JOHN,MARY). So far, the only thing we have is a complex Prolog term consisting of the semantic representations for the words in our input sentence in a sequence. Or - rather - not just in a sequence. If we look at it more closely, the structure of the term we get reflects the structure of the syntactic tree constructed by our DCG in the way it is bracketed. Only so far we have no clue how we can use this structure to put `john` and `mary` in their right places, namely the first and, respectively, the second argument slot of `love(_,_)`.

3.3.2 Putting Things in the Right Place I

Obviously, in order to obtain true first order formulae, the output of our little DCG needs to be further processed. The problem is that our resulting expressions consist of parts of a first-order formula glued together, instead of one complete-first order formula with all of its parts in the right place. What do we have to do? Let us first have a look at a simple case: From parsing the sentence 'Mary walks.' we get the term `mary+walks(_)`. From this, we would like to obtain the first-order formula WALK(MARY).

Postprocessing: Inserting the Arguments

In this example, we could do with simply inserting the item to the left of the `+` into the argument slot of the item to the right. The predicate `insertArgs/2` does so:

```
insertArgs(N+V,V) :-
        compose(V,_,[N]).
```

Here is an example call: `s(Sem,[mary,walks],[]), insertArgs(Sem,Inserted).`

What is going on here? Recall that the predicate `compose/3` (see Section 12.1) states that its first argument consists of a functor (given as first argument) and arguments (second argument, a list). In our example, the call to `compose/3` states that the term `walk(_)` is to consist of a functor (which is unified with the anonymous _) and exactly one argument, which is unified with `N`. Now `N` is in turn instantiated to `mary`, and so of course the only solution is `walk(mary)`.

Would this strategy of "left-to-right-insertion" work in general? It wouldn't. Recall that we obtained the term `john+(love(_,_)+mary)` for the sentence 'John loves Mary.'. In this case, the lexical representation for the verb has two argument slots. Moreover, it occurs to the left of one `+` and to the right of another at the same time. Our predicate `insertArgs/2` doesn't work any more.

So, let us design a second clause that takes care of sentences like 'John loves Mary' of the form (noun+(verb+object)). Here it is:

```
insertArgs(N+(V+O),V) :-
        compose(V,_,[N,O]).
```

?- Question!

Try this call: `s(Sem,[john,loves,mary],[]), insertArgs(Sem,Inserted).` Explain in your own words how the above clause of `insertArgs/2` works. Add lexical rules for a few other words (along the examples) to the DCG and play a bit with the program on your own computer:

1. Download `firstAttempt.pl`[2] and extend the DCG.
2. Start Prolog and consult the file:

   ```
   ?- [firstAttempt].
   ```

3. Parse a sentence or some other phrase (you have to give it as a list of atoms) and post-process the resulting `+` term:

   ```
   ?- s(Sem,[peter,sleeps],[]), insertArgs(Sem,Inserted).
   ```

   (Assuming you've added lexical rules for `peter` and `sleeps`).

3.3.3 Putting Things in the Right Place II

Things get even more complicated once we start considering quantified noun phrases. Quantified noun phrases are noun phrases that contain a determiner. For example the sentence 'A woman walks.' contains the the quantified noun phrase 'a woman'. The meaning of this sentence is captured by the first order formula $\exists X(\text{WOMAN}(X) \wedge \text{WALK}(X))$. Obviously, the contribution made to this formula by the word 'woman' should be the one place predicate symbol WOMAN. The verb 'walks' contributes the predicate symbol WALK. Thus, the determiner must contribute the quantifier *and* the pattern of the quantification. The lexical rules for the noun 'woman' and the determiner 'a' are as follows:

```
n(woman(_)) --> [woman].

det(exists(_,_&_)) --> [a].
```

The rule for the determiner 'a' gives us a template for building a quantified first-order formula by supplying the information that the formula will be an existential quantification, and that there will be a conjunction in the scope of this quantification.

To combine determiner and noun to an NP, we need a suitable combination rule:

```
np(DetSem+NSem) --> det(DetSem),n(NSem).
```

But now what happens when we parse 'A woman walks.' with our DCG? Try it yourself: `s(Sem,[a,woman,walks],[])` . How do we arrive at `exists(x,woman(x) & walk(x))` from there? Clearly 'inserting left into right' isn't an option. All in all, simply inserting subterms of our complex representation into each other no longer looks like a feasible solution at all. Even if we had a systematic way of putting each of the subterms into its final place, we hardly could ensure that the arguments of the two predicates `woman` and `walk` are bound by the existential quantifier.

[2] `./prolog/firstAttempt.pl`

Try to extend the predicate!

In !!!UNEXPECTED PTR TO EX_EX.LAM.INSERTARGS!!! you're asked to extend `insertArgs/2` so that quantified formulae can be handled as well. This will confront you quite directly with the shortcomings of our naive approach, which we want to discuss next. So it's a good idea to do the exercise right away!

What we have learned...

We've managed to teach our DCG how to construct first-order formulae for some simple sentences quite quickly. But obvioulsy our approach is not yet very elaborate (as you could clearly see in the above exercise...). Its most severe shortcomings are:

- Even in our tiny fragment of natural language semantics the formulation of the clauses of `insertArgs/2` already required quite some care. Contrary to our resolutions, we obviously didn't work systematically at all.
- Consequently, we will in the worst case have to give one clause of `insertArgs/2` for each (of the infinitely many) possible syntactic structures.

Obviously, what we have just implemented is not a simple postprocessing step, but a little promising ad hoc improvisation. As soon as we want to scale it up only a little bit, it becomes unpleasantly complex.

3.4 The Lambda Calculus

3.4.1 Lambda-Abstraction

λ-expressions are formed out of ordinary first order formulae 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* . The variable abstracted over is (λ-*)bound* by its respective λ-operator within an abstraction, just as a quantified variable is bound by its quantifier inside a quantification.

Abstractions

So the following two are examples of λ-abstractions:

1. $\lambda x.\text{WOMAN}(x)$
2. $\lambda u.\lambda v.\text{LOVE}(u,v)$

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 will 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.

While the λ-bound variables in our two examples act as placeholders for missing constant symbols, we will also use λ-prefixes to mark other kinds of missing information. For instance we will use λ-prefixes and variables to mark the gaps where information is missing in the template associated with the indefinite article 'a' (don't care about the @-symbol for now. We'll explain what it means in a minute (page 43)):

$$\lambda P.\lambda Q.(\exists x(P@x \wedge Q@x))$$

Here, P and Q stand for missing predicate symbols. 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.

3.4.2 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 our first example, the binding of the free variable x in WOMAN(x) explicitly indicates that WOMAN has an argument slot where we may perform substitutions. Operationally speaking, the purpose of abstracting over variables is thus to mark the slots where we want substitutions to be made. We will use concatenation 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):

$\lambda x.\text{WOMAN}(x)@\text{MARY}.$

Functional Application, β-Reduction

This compound expression consists of the abstraction $\lambda x.\text{WOMAN}(x)$ 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:

$\text{WOMAN}(\text{MARY})$.

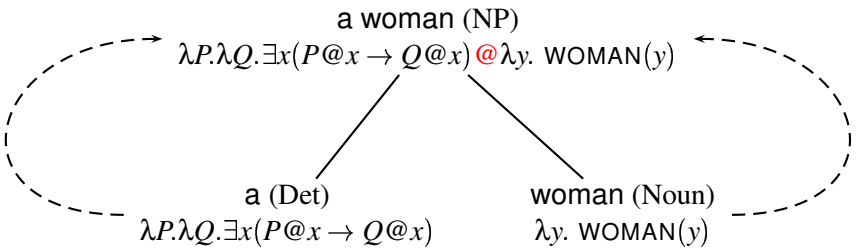
So basically the @-symbol has taken the place of the `+`-operator which we used in our first approach. And β-reduction will replace our `insertArgs/3`-postprocessing step. Abstraction, functional application, and β-reduction together thus drive our first really systematic semantic construction mechanism. Next, let's see how it works in practice.

3.4.3 Using Lambdas

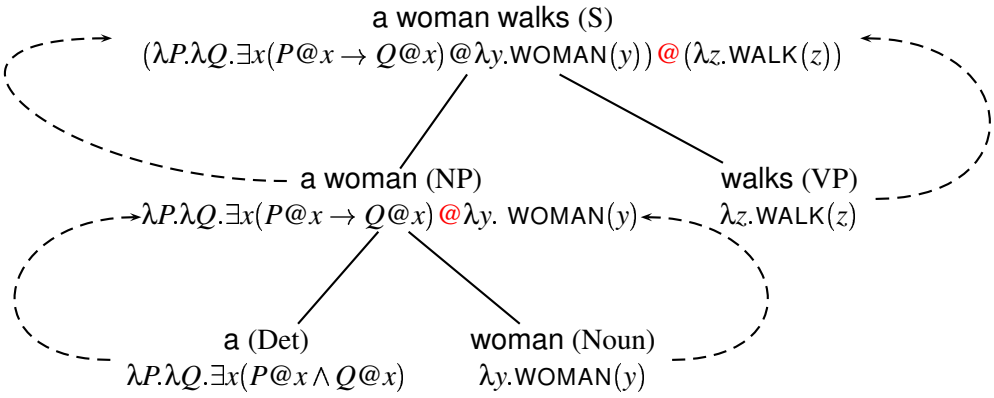
Fine, now we have a tool for juggling around with missing information in incomplete first order formulae. But does it really solve our problems? Does it help us when we want to construct a semantic representation for a sentence compositionally?

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 (we mark this application by inserting an @ between functor and argument).

Building a structured application...



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



The S node is associated with $(\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)@\lambda y.\text{WOMAN}(y))@\lambda z.\text{WALK}(z)$. We obtain this representation by a procedure analogous to that just 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.

Up to now it may seem as if we haven't gained much. The representation at the S-node has the same internal structure as

```
(exists(_,_&_)+woman(_))+walks(_)
```

produced by our first attempt (page 39) for the same sentence. But now comes the time that the λ-prefixes in our new representations really help us solve a problem.

...and reducing it.

Instead of hand-tailoring some specially dedicated post-processing method, we will simply β-reduce the expression we find at the S node. Let's see what happens. 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. 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, but this time it should be clear 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:

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

?- Question!

Look at the representations $\lambda y.\textsc{woman}(y)$ and $\lambda z.\textsc{walk}(z)$ again. It may seem strange at first sight that we give semantic representations for intransitive verbs that are structurally like the ones for common nouns. Yet from the point of view of logics, this is the obvious thing to do. Why?

3.4.4 [Sidetrack] Simply Typed Lambda-Calculus

The perspective we've been taking on λ-calculus so far has been a very technical one. We 've been viewing it simply as a tool for controlling substitutions, without caring much what single λ-expressions actually mean. If you've heard about λ-Calculus before in other semantics textbooks or lectures, you will probably feel that we have left out a discussion of at least two points:

1. *Types:* The version of λ-calculus most commonly used in linguistics is simply typed λ-*calculus* . In this calculus, expressions get assigned types that regulate function application.

2. *Functions:* λ-expressions can be given an interpretation in terms of functions over a system of complex domains. (In fact λ-calculus was originally designed to investigate function definition, function application and recursion.)

As regards the first point: The version of λ-calculus we're going to use is untyped, and we'll argue below that we don't have to interpret any λ-expressions for our purposes. Nevertheless, the two points just mentioned are very important theoretical topics concerning the use of λ-calculus in formal semantics. Therefore, we will now give a short sketch of some of the central ideas involved in an interpreted simply typed λ-calculus.

Types

Every expression gets assigned a unique type in simply typed λ-calculus. On the one hand such types restrict how applications can be formed and regulate what the results of applications will be after β-reduction. On the other hand types correspond to certain classes of semantic entities and thereby tell us that the typed expressions are to be interpreted in terms of these entities.

Basic types

But how does all of this work? Let's assume we are using simply typed λ-calculus to construct first order meaning representations with the goal of evaluating them in a given model. So we know what the domain of that model is. Now for our type system. Initially, we have two basic types, written as e and t. The type e corresponds to the domain of our model (i.e. the set of our basic individuals). It gets assigned to constant symbols, which pick out entities from that set in our model. So for instance, a constant symbol MARY would be assigned type e because it picks out a basic entity (for instance Mary). Moreover, standard first order variables (which range over the domain of our model) are also of type e.

The other basic type is t. It corresponds to the set of truth values (that is, the set $\{\text{TRUE}, \text{FALSE}\}$, and it gets assigned to complete well-formed first order formulae (i.e. to expressions that can be evaluated to a truth value in our first order model).

Complex types

We can then recursively construct complex types from these two basic types. Such complex types are written as $< \tau_1, \tau_2 >$, where τ_1 and τ_2 are themselves types. For example from the basic types e and t we can form the complex type $< e, t >$, and from this complex type and type t, we can form $<< e, t >, t >$. A complex type of the form $< \tau_1, \tau_2 >$ corresponds to the set of functions from things of type τ_1 (the *argument type*) to things of type τ_2 (the *result type*). For instance, type $< e, t >$ corresponds to the functions from basic entities (like, for instance, John and Mary) to the truth values.

But what do we need complex types for? They are the types that get assigned to λ-expressions. A λ-abstraction of the form $\lambda x.P$ is of type $< \tau_1, \tau_2 >$, where τ_1 is the type of the λ-bound variable x, and τ_2 is the type of the scope P of the abstraction.

For example (assuming that $\text{MAN}(x)$ is a well-formed first order formula), the type of the abstraction $\lambda x.\text{MAN}(x)$ is $< e,t >$. The reason is that x is a first order variable ranging over individuals, hence of type e, and $\text{MAN}(x)$ is a first-order formula, hence of type t.

Well-Typedness

Types regulate which applications are possible: In typed λ-calculus, applications are only allowed if the type of the argument (to the right of the @ in the expression) is the same as the *argument type* of the functor (to the left of the @). The result of β-reducing such an application is then always of the *result type* of the functor. If all applications in an expression are allowed according to this criterion, the expression is *well-typed* .

Consider the following expression:

$$\lambda x.\text{MAN}(x)@\text{MARY}$$

The application is allowed (it is well-typed), because MARY is of type e and this is also the argument type of the functor

$$\lambda x.\text{MAN}(x).$$

Recall that this functor is of type $< e,t >$. The result of β-reducing the above application is $\text{MAN}(\text{MARY})$ of type t - the result type of the functor.

In contrast the application

$$\lambda y.\text{WOMAN}(y)@\lambda x.\text{MAN}(x)$$

is not admissible (it is not well-typed), because the type of the argument ($< e,t >$, this time) and the functor's argument type (e) don't match.

Thus one aspect of the type system is that it formalizes what we express by a typographic convention in our untyped approach: The different kinds of missing information that λ-bound variables may stand for. But more than this. If we form our abstractions only from well-formed first order formulae and respect the well-typedness restrictions, we really know for sure that we can always return to a well formed first-order formula through a series of well-typed applications and β-reductions. Consider for instance the λ-expression $\lambda X.\lambda z.(X@\lambda x.\text{LOVE}(z,x))$, where X is of type $<< e,t >,t >$, and z and x are of type e (incidentally, this is the kind of λ-expression we will be using for transitive verbs). The expression is built up from the first-order formula $\text{LOVE}(z,x)$, abstractions and a well typed application. Hence we can be sure that we will be able to get a well-formed first order formula (namely $\text{LOVE}(z,x)$ with z and x possibly replaced by some constants) if we feed our original expression arguments of the right type and then β-reduce. Notice that this 'guarantee of full β-reducibility' is a property we plainly took for given with our untyped λ-calculus when we said above that we would view it as a simple notational extension of first order logic.

Interpreting Abstractions and Applications

We have not said much about the interpretation of λ-abstractions and applications so far. Yet keeping in mind what sets correspond to complex types, this becomes an easy matter: For each complex type, we have the corresponding set of functions from its

argument type to its result type. Thus, we will simply interpret each λ-abstraction as one function from the set corresponding to its (complex) type. Applications are interpreted by really applying the interpretation of the functor to the interpretation of the argument.

For instance, we will interpret $\lambda x.\text{MAN}(x)$ (of type $< e,t >$) as a function from individuals to truth values. The application $\lambda x.\text{MAN}(x)@\text{MARY}$ will be interpreted by applying this function to the interpretation of MARY.

Moreover, given a model, we will choose as interpretation for $\lambda x.\text{MAN}(x)$ the function that yields TRUE exactly for those members of the domain that are in the set interpreting the predicate MAN (the so-called *characteristic function* of that set). Hence, the interpretation of $\lambda x.\text{MAN}(x)@\text{MARY}$ will be the same as the interpretation of the β-reduced formula MAN(MARY) in our given model. Generally, choosing characteristic functions as the interpretation of λ-abstractions of type $< e,t >$, we can couple the interpretation of λ-terms with the first-order interpretation of our goal formulae: We can be sure that the interpretation of any well-typed application that can be β-reduced to a first order formula will be the same as the interpretation of this formula itself.

Let's add one final, more fundamental remark to our discussion. We have just seen how simply typed λ-calculus allows us to give an interpretation of the λ-expressions involved in semantic construction in terms of well-understood mathematical entities (namely functions). This is a great achievment from the point of view of compositionality. Remember that one of the requirements made by the principle of compositionality (page 34) was that we should be able to give individual semantic interpretations to the syntactic parts and sub-parts of the expressions in our language. Semantic construction using simply typed λ-calculus fulfills this requirement in the particularily strict sense of *d-compositionality*, thus even beyond the level of meaning representations: The interpretation in terms of functions which we've dicussed enables us to state exactly which of the functions built up over the domain of our model corresponds to any syntactic part of a sentence.

Then why do we use the untyped version?

But although very interesting from a theoretical perspective, the issues just discussed are not nearly as important for understanding the λ-based semantic construction mechanism we're going to implement. In particular:

- For a start, we do not care to interpret any λ-expressions before they're fully β-reduced. Also, we will not ask for the meaning of any individual parts of our sentence relative to a model. Below sentence level, we will be happy with having a meaning representation to work ourselves through to the whole sentence. Most of the time, the first-order formula that we get after β-reducing is all that interests us. So any arguments about interpreting unreduced λ-expressions aren't of great relevance for our purposes.

- As regards the guarantees about getting useful results after β-reduction that come with simply typed λ-calculus: We will basically use our untyped version of λ-calculus as if it was typed. All the (well formed) expressions we will encounter would also be well typed in a typed λ-calculus. Thus we may safely take it for granted that we can always reduce the meaning reprentations for sentences to

first order formulae. This is only due to our own discipline, we could do less decent things with our untyped calculus (and the implementation we're going to base on it).

Implementing typed λ-calculus would require a lot of efforts particularily for managing the type system. So we will use a disciplined approach to the untyped version instead.

3.4.5 Advanced Topics: Proper Names and Transitive Verbs

It looks as if there are clouds on the horizon. For example, while the semantic representation of a quantifying NP such as 'a woman' can be used as a functor, surely the semantic representation of an NP like 'John' will have to be used as an argument. Will this problem throw a spanner in the works of our new 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, the same way as quantified noun phrases. So let's simply keep this intended use in mind when designing the semantic representations for proper names. It then all becomes a matter of abstracting cleverly. Indeed the λ-calculus offers a delightfully simple functorial representation for proper names, as the following examples show:

'Mary': $\lambda P.P@\text{MARY}$

'John': $\lambda Q.Q@\text{JOHN}$

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. They are abstractions, thus they can be used as functors in the same way. However looking at the inside, note what such functors do. They are essentially instructions to substitute their argument in P (Q), which amounts to applying their own arguments to themselves! 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 $\lambda u.\lambda v.\text{LOVE}(u,v)$ (which we've seen in Section 3.4.1) 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

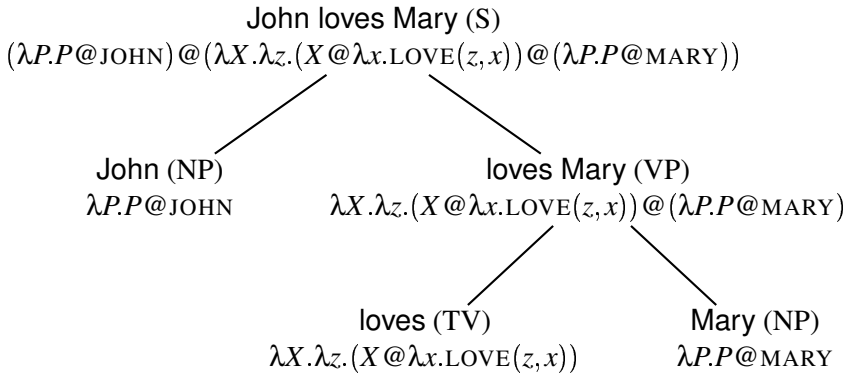
time? Let's try something like our role-reversing trick again; we'll assign 'loves' the following λ-expression:

$\lambda R.\lambda z.(R@\lambda x.\text{LOVE}(z,x))$.

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.



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 43) 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 (namely to $\lambda x.\text{LOVE}(z,x)$). This application is going to be no problem - it's structurally the same we would get if our object NP was the subject of an 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, as if it was the representation of an 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!

3.4.6 The Moral

What λ-calculus gives us

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 how to make 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. All this could be seen most clearly when we designed the representation for the transitive verb 'loves'. We had to consider that it would be applied to its object NP and that the result of this application would then itself be an argument, namely of the subject NP. And all the time, we had to keep in mind that we wanted to arrive at formulae like $\text{LOVE}(a,b)$ for sentences like 'A loves B'.

For sidetrack-readers...

If you've read the sidetrack (page 45) on typed λ-calculus, you will remember that there, the type system would help us with the kind of considerations just discussed.

In fact one way to think about the type system is to view it as a static, predetermined encoding of how semantic material fits together. Also, remember the denotations that were assigned to typed λ-expressions according to their type, in terms of functions. Assigning such denotations, it is possible to ask whether a decision for one representation instead of another one is semantically adequate. For instance, we may ask if it is more adequate to interpret the proper name 'Mary' as Mary herself (or whatever we take to be MARY in a given model), or as the set of all of Mary's properties (characterised by the function $\lambda P.P@\text{MARY}$). Maybe our semantic intuitions help us in answering such questions, or some deeper philosophical reasons commit us to one specific kind of answer.

Summing up

For our purposes here 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.

3.4.7 Accidental Bindings

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:

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

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

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

$\text{LOVE}(y,x)$ has been moved inside the scope of $\exists y$. In result, the occurence 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:

$$\lambda y.(\exists y(\text{WOMAN}(y)\wedge\text{LOVE}(y,y)))$$

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.

3.4.8 Alpha-Conversion

But such interactions need never happen. Obviously at the heart of our problem lies the fact that 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 $\lambda x.\text{MAN}(x)$, $\lambda y.\text{MAN}(y)$, and $\lambda z.\text{MAN}(z)$ are equivalent, as are the expressions $\lambda Q.\exists x(\text{WOMAN}(x) \wedge Q@x)$ and $\lambda Y.\exists x(\text{WOMAN}(x) \wedge Y@x)$. 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 $\text{MAN}(\mathcal{A})$, and the result of applying either of the last two expressions to $\mathcal{A}$ is $\exists x(\text{WOMAN}(x) \wedge \mathcal{A}@x)$.

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 the process of semantic construction in this course, 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.

3.4.9 Summing Up

Let's sum up what we have achieved. Our decision to move to the disciplined approach of the λ-calculus was sensible. For a start, we don't need to spend any time thinking

about how to combine two semantic representations - functional application and β-conversion give us a general mechanism for doing so. Moreover, much of the real work is now being done at the lexical level; indeed, even the bothersome problem of finding a decent way of handling NP representations uniformly now has a simple lexical solution.

What's next?

For the remainder of this lecture, the following version of the three tasks listed earlier (page 36) 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.

3.5 Implementing Lambda Calculus

3.5.1 Representations

First, we have to decide how to represent λ-expressions in Prolog. Something as simple as the following will do:

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

Be careful!

According to our convention (page 20), `x` is a Prolog constant representing the variable bound by the λ. `F` is (the Prolog representation of) either a first-order formula or a λ-expression. Note that in the system we are going to implement, you will often see Prolog variables in place of the `x`. In Section 3.5.3 you will see why.

Secondly, we have to decide how to represent concatenation. 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. So - as we discussed above (page 43) - the `@`-operator will finally take over the role of the `+`-operator in our first attempt (page 39) at semantic construction, and β-conversion will replace the post-processing formerly done by `insertArgs/2`.

?- Question!

We could as well perform β-conversion steps interleaved with the construction of the application. For instance, we could always fully reduce the λ-expressions at the nodes of our syntactic trees before going on. Why? Look again at one of our previous examples and see how it works out if you β-convert at different stages!

3.5.2 Extending the DCG

Let's see how to use this notation in DCGs. We'll deal with the rules first. Actually, there's practically nothing that needs to be said here. If we work with rules in the manner suggested by (our refined version of) Task 3 (page 53), all we need is the following:

***See file* `firstLambda.pl`.**

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

np(PN)--> pn(PN).

np(Det@Noun)--> det(Det), noun(Noun).

vp(IV)--> iv(IV).

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

The unary branching rules just percolate up their semantic representation (here coded as Prolog variables `NP`, `VP` and so on), while the binary branching 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.

3.5.3 The Lexicon

The real work is done at the lexical level. Nevertheless, the lexical entries practically write themselves:

***See file* `firstLambda.pl`.**

```
noun(lambda(X,siamesecat(X)))--> [siamese,cat].

noun(lambda(X,woman(X)))--> [woman].

iv(lambda(X,walk(X)))--> [walks].

tv(lambda(X,lambda(Y,X@lambda(Z,love(Y,Z)))))--> [loves].
```

If you do not remember the somewhat difficult representation of `TV`*s*, look at Section 3.4.5 again.

Prolog Variables?

Note that we break our convention (page 20) of representing FO variables by constants here. A clever idea, because this way we can do without implementing α-conversion. We will see why in Section 3.5.7.

And here's the code stating that $\lambda P.P@\text{JOHN}$ is the translation of 'John', and $\lambda P.P@\text{MARY}$ the translation of 'Mary':

```
pn(lambda(P,P@john))--> [john].

pn(lambda(P,P@mary))--> [mary].
```

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,forall(X,(P@X)>(Q@X)))))--> [every].

det(lambda(P,lambda(Q,exists(X,(P@X)& (Q@X)))))--> [a].
```

3.5.4 A First Run

***See file* `firstLambda.pl`.**

Now, this makes semantic construction during parsing extremely easy: we simply use @ to record the required function/argument structure. (The new DCG with additional argument is part of the file `firstLambda.pl`.) Here is an example query:

```
?- s(Sem,[mary,walks],[]).

Sem = lambda(_G358, _G358@mary)@lambda(_G364, walk(_G364))
```

Or generate the semantics for 'John loves Mary': `s(Sem,[john,loves,mary],[]).`

?- Question!

Where do the anonymous variables in the output of our predicate (for instance `_G358` and `_G364` in the above example) come from?

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. Next, we will implement the predicate `betaConvert/2`, which will do the job.

3.5.5 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 three clauses of the predicate in detail.

See file **`betaConversion.pl`.**

```
betaConvert(Var,Result):-
        var(Var),
        !,
        Result=Var.

betaConvert(Functor@Arg,Result):-
        compound(Functor),
        betaConvert(Functor,lambda(X,Formula)),
        !,
        substitute(Arg,X,Formula,BetaConverted),
        betaConvert(BetaConverted,Result).
```

The first clause of `betaConvert/2` simply records the fact that variables cannot be further reduced.

The second clause 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 compound,
3. The functor must be (recursively!) reducible to the form `lambda(X,Formula)` (and is in fact reduced to that form before going on).

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

3.5.6 Beta-Conversion Continued

Finally, there is a clause of `betaConvert/2` that deals with those expressions that do not match the first two clauses. Note that the first two clauses both contain cuts. So, this last clause will deal with all (and only) the non-variable 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.

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

This clause breaks down such expressions and recursively reduces their arguments or subformulas. This is done with the help of:

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

?- Question!

We said that the last clause handles atomic formulas with arguments that are still reducible. Given our grammar and lexicon, can we ever get this kind of expressions?

Here is an example query with β-conversion:

```
?- s(Sem,[mary,walks],[]), betaConvert(Sem,Reduced).

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

Try it for 'John loves Mary': `s(Sem,[john,loves,mary],[]), betaConvert(Sem,Res).`

?- Question!

Above, we said that complex formulas like `walk(john) & (lambda(X,talk(X))@john)` are reduced to 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.

?- Question!

In Chapter 6, we will learn about a way of semantics construction that differs in some respects from what we are doing now with the λ-calculus. Without going into detail, (sub-)formulas might be of the form `walk@john` there. Nevertheless, we will use the predicate `betaConversion/2` given above to reduce formulas containing λ*s*. Now, the question is: Which of the clauses of `betaConvert/2` handles queries like `betaConvert(walk@john,Res)`? What is the result?

Our implementation of `betaConvert/2` uses a version of the Sterling and Shapiro `substitute/4` predicate. Because this is an important predicate, we recommend that you look at its definition in Section 12.1.

3.5.7 An Afterthought on Alpha-Conversion

One thing that apparently still needs to be done is to implement α-conversion. Recall from Section 3.4.8 that α-conversion can prevent accidental variable bindings in the course of β-reduction by renaming variables.

In principle, things can go wrong...

Now, as we have a module for β-reduction at our disposal, let's first practically see whether things can in fact go wrong because we choose inadequate variable names. Below, you will find the final β-reduction step of the semantics construction for 'Every woman loves a siamese cat'. Here are the translations of the two (sub-)phrases where the variables have been named the same:

'Every woman'	`lambda(X,forall(Y,woman(Y)>(X@Y)))`
'loves a siamese cat'	`lambda(X,exists(Y,siamesecat(Y)& love(X,Y)))`

If we form an application out of these two expressions and β-reduce it, we get this:

```
?- betaConvert(lambda(X,forall(Y,woman(Y)>(X@Y)))@lambda(X,exists(Y,siamesecat

R = forall(Y,woman(Y)>exists(Y,siamesecat(Y)& love(Y,Y)))
```

We've already seen that problem in Section 3.4.8: The formula does *not* represent the meaning we intended. It rather says something like 'There is a self-loving siamese cat for every woman.'

So, do we run into trouble with our approach if we don't use α-conversion?

...but not in our implementation!

No, we don't! The representations in the example above were hand-written. In our implementation, all representations coming from the lexicon contain *new and fresh* PROLOG variables. If you look at the lexical entries (page 55) again, you can see that it is not possible to build any two representations using the same variable names. Check what representations our approach generates for 'Every woman' and 'loves a siamese cat': `np(NP,[every,woman],[]),vp(VP,[loves,a,siamese,cat],[]).` (We conjoin the queries in order to execute them in one go). Are there any shared variables between the two phrases?

Prolog Variables

Once again: Using Prolog variables for first-order variables during semantic construction violates our convention (page 20) - strictly speaking. But now we can see what we gain breaking that rule: We always automatically get brand new variables with every lexical lookup, without ever having to generate or compare them. Thus we need never bother with α-conversion - and all for free. Well, almost for free. If we want to work with our formulas afterwards, we have to convert them into our standard format. In the next Section (page 59) you will see that this is easily done by one small helper predicate.

Project

In !!!UNEXPECTED PTR TO EX_EX.LAM.ALPHY!!! you are asked to implement α-conversion.

3.6 Running the Program

We've already seen a first run of our semantically annotated DCG, and we've implemented a module for β-conversion. Now we shall put them together in a predicate `firstLambda/0` to get our first real semantic construction system. Here's the code of `firstLambda/0`:

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

This predicate first converts the keyboard input into a list of Prolog atoms (using `readLine/0` (page 201) form `comsemLib.pl`). Then it uses the semantically annotated DCG from `firstLambda.pl` and tries to parse a sentence.

So far, so good. But what are the two calls `resetVars,vars2atoms(Formula)` for? The reason why we have to include them is that we've broken our representational convention for variables (we've explained why in Section 3.5.7). Hence the `Formula` coming from the call `s(Formula,Sentence,[])` may still contain Prolog variables. For instance when we construct the semantics for the sentence 'John walks', we will get something like `lambda(_G365,_G365@john)@lambda(_G371,walk(_G371))`.

Undo Variable Cheat

So we undo our little cheat before going on. Calling `resetVars,vars2atoms(Formula)` (see Section 12.1) instantiates the Prolog variables in `Formula` with new (and consistently distinct) variables-as-atoms, according to our convention (page 20). Finally, the resulting λ-expression is β-converted by our predicate `betaConvert/2`. In the last step, the resulting formula is printed out (the predicate `printRepresentations/1` used for this purpose is documented here (page 201)).

Before we turn to the exercises for this chapter, we give a listing of the files that contain the code discussed in this chapter. You run our λ-based semantic construction program by:

1. Consulting the file `runningFirstLambda.pl`, which contains the driver predicate we've just seen:

   ```
   ?- [runningFirstLambda].
   ```

2. Calling the driver predicate:

   ```
   ?- firstLambda.
   ```

3. Entering a sentence of your choice at the prompt.

Code For This Chapter

See file `usingDCG.pl`.	Our very first experimantal DCG.
See file `firstAttempt.pl`.	The code for our first attempt at semantic construction, using t
See file `firstLambda.pl`.	DCG for semantic construction using λ-calculus.
See file `betaConversion.pl`.	β-conversion.
See file `runningFirstLambda.pl`.	Driver predicate for our first lambda approach.
See file `signature.pl`.	Generating new variables
See file `comsemLib.pl`.	Auxiliaries.
See file `comsemOperators.pl`.	Operator definitions.

3.7 Exercises

Exercise 3.1 *Add a clause to* `insertArgs/2` *(see Section 3.3.2) in* `firstAttempt.pl` *that handles quantified sentences like 'A man walks.'*

*[*Hint: *If you follow the pattern of the clauses of* `insertArgs/2` *already given, you may run into trouble with your new clause since terms like* `exists(_A,_B&_C)+man(_D)+walk(_E)` *also match on one of the first two clauses. You may either catch this inside the respective clause or simply put your clause* above *the other clauses.]*

Exercise 3.2 *Look at the semantics construction in Section 3.4.5 again. Work through the functional applications and* β*-reductions required to build the VP and S representations. Make sure you understand the role-reversing idea used in the TV semantic representation.*

Exercise 3.3 *Starting off from our treatment of transitive verbs (Section 3.4.5), how would you handle ditransitive verbs such as 'offer'? Give a semantic representation for 'offer', as well as semantically annotated grammar rules to analyse sentences like 'Mary offers John a siamese cat.'*

Exercise 3.4 *Find a suitable* λ*-expression for the determiner 'no' and add it to our implementation of* λ*-calculus. Test your solution with sentences like 'No man walks.'*

Extend our implementation of λ*-based semantic construction accordingly.*

Exercise 3.5 *[This is a mid-term project]*

See file **`betaConversion.pl`.**

We could prevent accidental variable bindings as shown in Section 3.5.7 once and for all if we had a module doing α*-conversion.*

Design such a module and include it as a preprocessing step in a new driver predicate for β*-conversion:*

```
alphaBetaConvert(F@A,Result):-
   alphaConvert(F,ConvertedF),
   betaConvert(ConvertedF@A,Result).
```

Please comment your code and provide a short documentation. The documentation should include a brief description of the problem and of your solution.

Test your module on some examples and include your results in the documentation. Does your program support the use of Prolog atoms *to represent first order variables (i.e could you for instance write* `lambda(x walks(x))` *instead of* `lambda(X walks(X))`*)?*

Please contact[3] *us if you would like to do this exercise as your mid-term project.*

[3] `mailto://stwa@coli.uni-sb.de`

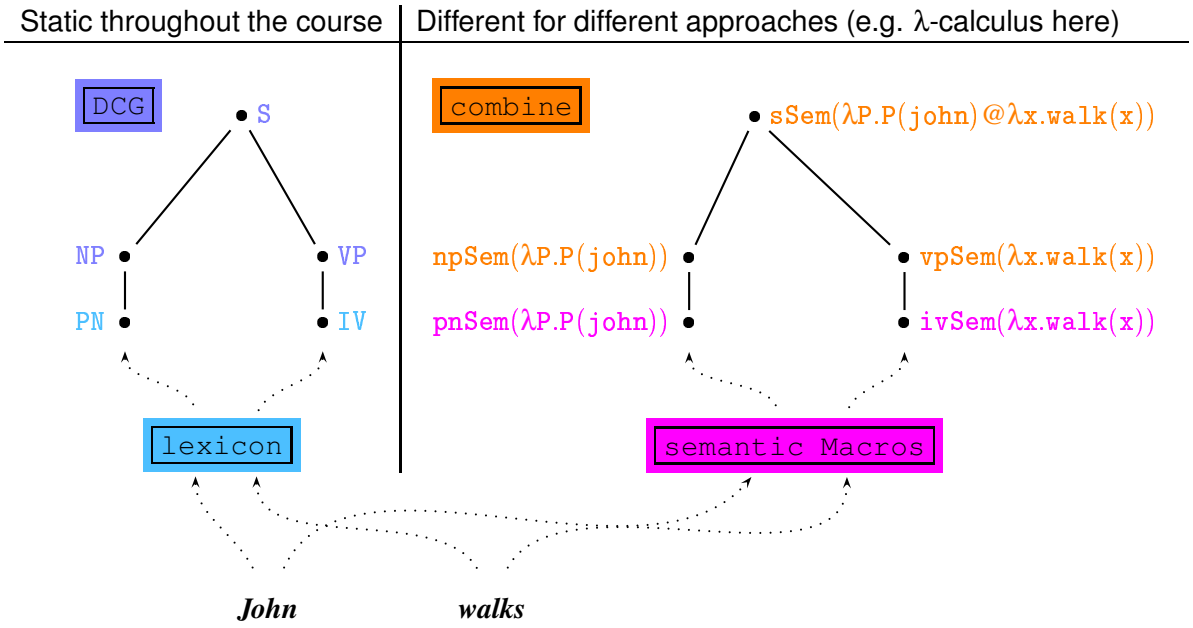
4

Towards a Modular Architecture

4.1 Architecture of our Grammar

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 parts in detail, we will give an overview over what exactly the tasks are of each of these components.

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` $\rightarrow$ `PN` and `VP` $\rightarrow$ `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` $\rightarrow$ `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: $\lambda P.P(\text{JOHN})$ and $\lambda x.\text{WALK}(x)$. 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: $\lambda P.P(\text{JOHN})@\lambda x.\text{WALK}(x)$.

To get $\text{WALK}(\text{JOHN})$, a postprocessing step is neccessary, 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.

4.2 The Syntax Rules

4.2.1 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.

4.2.2 The Syntax Rules we will use

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.'

If you want to test the grammar, you will have to do this within a semantic framework, e.g. λ-calculus: `lambda([mary,knows,every,owner,of,a,siamese,cat],Sem).`

4.3 The Semantic Side

4.3.1 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 (page 55), 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 72) 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`.

4.3.2 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. Let's illustrate what we've achieved at an example. As a case study, we will implement the combination technique we've got used to by now: Functional application. So the task is simply building the obvious 'apply the function to the argument statements' expressed with the help of `@`. How do we have to define `combine/2` for this purpose? 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 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.

?- Question!

Recall our first attempt at semantic construction with extended DCGs. There (page 39), we used the predicate `insertArgs/2` to postprocess terms glued together with `+`. Doesn't the predicate `combine/2` here do exactly the same with λ-terms? Why don't we run into the same trouble as with `insertArgs/2`?

4.4 Looking Up the Lexicon

4.4.1 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 on to 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 69), 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 changing 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. And again, to change the semantic formalism we will simply re-implement our interface predicates (i.e. the semantic macros) - that's it.

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

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

4.4.2 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 is to be found in the file `englishLexicon.pl`. All these rules will work for us unchanged throughout the course.

4.4.3 '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 here the semantic macros will again be the sole source of semantic information.

In !!!UNEXPECTED PTR TO EX_EX.ARCHITECTURE.COPULA!!!, you are asked to find out how copula constructions are handled in our approach.

4.4.4 Semantic Macros for Lambda-Calculus

Semantic Macros

Let's now turn to our case study again: semantic construction with λ-calculus and functional application. We saw in the last chapter (page 39) 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. Now the only semantic information that our `lexicon/4`-facts supply are the relevant constant and relation symbols. So our 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 any noun given the predicate symbol `Sym`, 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 49) 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]).
```

In fact, the macro does exactly the same as `nounSem/2`. This is not surprising at all - after all the λ-expressions we saw (page 44) for intransitive verbs are also exactly like the ones for nouns.

Finding the right abstraction pattern for transitive verbs turned out (page 49) to be a little more involved. Nevertheless now we've got it, this pattern too 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 Section 3.5.3:

```
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. While the first argument does have a value, this 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 see more types of semantic macros as we work our way through the course. But let's emphasize: 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`.

4.5 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 (see Section 12.1). 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 lexica
See file `comsemOperators.pl`.	Operator definitions.
See file `englishGrammar.pl`.	The DCG-rules and the lexicon (using module `englishLexico`
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.

4.6 Exercises

Exercise 4.1 *Find out how copula verbs are handled in the lexicon and grammar, and how our implementation generates the semantic representations for sentences like 'Mary is a therapist' and 'Mary is not John'. You may either hand in your solution in (hand-)written form or send us a an E-mail. In the latter case, you can use the Prolog notation for λ-terms. E.g. for 'John walks', the solution might look like this:*

```
John  ~> lambda(A,A@john)
walks ~> lambda(B,walk@B)

John walks ~> lambda(A,A@john)@lambda(B,walk(B))

           ...

           <=> walk(john).
```

Exercise 4.2 *Add a treatment of ditransitive verbs such as 'offer' to the implementation discussed in this chapter. Use the following formalization as a starting point:*

$$\lambda R\lambda O\lambda g.O@\lambda o.R@\lambda r(\text{OFFER}(g,o,r))$$

Hint: Move along the lines of transitive verbs. First specify the `lexicon/4`*-fact and the lexical and phrasal DCG rules. Then specify the semantic macro and finally design an appropriate* `combine` *rule. Don't forget the brackets in your complex applications, e.g. ((A@B)@C)!*

Test your solution with sentences like 'Mary offers John a siamese cat.'

Exercise 4.3 *Find a suitable λ-expression for the determiner 'no' (or use what you've thought out in !!!UNEXPECTED PTR TO EX_EX.LAM.NO!!!) and add it to our implementation. Test your solution with sentences like 'No man walks.'*

[Hint: remember the special treatment of determiners!]

Exercise 4.4 *Extend our implementation such that it covers negated sentences like 'It is not the case that Mary walks.'*

Hint: Find out how if-then-sentences are treated.

Exercise 4.5 **This is an important exercise!**

Test the coverage of our system. Try to find some sentences where the semantic representations returned by our implementation are not satisfactory. Are the problems due to our implementation, or due to the mechanism of λ-based semantic construction itself?

Exercise 4.6 *[Mid-term project]*

Localize the system we've implemented. That is, take it to a language of your choice by adapting the lexicon and grammar accordingly. The range of constructions and phenomena covered should be the same or comparable to what is covered by the English version we've discussed. If you want to cover different semantic phenomena, you'll probably have to extend the semantic macros, too.

Your solution should contain a sort of report, documenting the changes you make, as well as the difficulties you encounter. Do you think that the difficulties are particular to the language you've chosen? Does the modular character of the system support you in your work?

5

Scope and Underspecification

5.1 Scope Ambiguities

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

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. $\forall x.\text{STUDENT}(x) \rightarrow \neg\text{PASS}(x)$
2. $\neg\forall x.(\text{STUDENT}(x) \rightarrow \text{PASS}(x))$

Exercise 5.1 *Quantifiers and negation aren't the only scope taking elements. Other candidates (and thus other sources of genuinely semantic ambiguity) include certain adverbs. For example* modal adverbs *like 'possibly' may interfere with the scope of determiners. Consider the sentence 'Possibly a dog is barking'. What are the different readings of this sentence?*

5.1.2 Scope Ambiguities and Montague Semantics

Using our Implementation

Now let's see what Montague Semantics has to say about this. In !!!UNEXPECTED PTR TO EX_EX.ARCHITECTURE.COVERAGE!!!, we asked you to test our implementation of λ-calculus and to try to find sentences the approach does not handle properly.

Maybe you've come up with 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 $\lambda P\lambda Q\forall x(P(x) \rightarrow Q(x))$. We will of course have to expand those abbreviations at some point when we want to perform β-reduction.

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

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

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

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 $\mathsf{A}@\lambda y\textsc{woman}(y)$ part in front of the Every. Something like the following expression would do:

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

Exercise 5.2 *Convince yourself - by expanding the abbreviations and* β*-reducing - that the above expression really is the second reading.*

Exercise 5.3 *Notice that we not only moved* $\mathsf{A}@\lambda w.\textsc{woman}(w)$ *in front of the* Every*, but tacitly also moved the adjacent* $@\lambda y$ *bit with it. Explain why this makes sense. It may help you to look at these colored representations of the two readings:*

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

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

5. $\exists z.\text{THERAPIST}(z) \wedge \exists y.\text{SIAMESECAT}(y) \wedge \forall x.((\text{OWNER}(x) \wedge \text{OF}(y,x)) \rightarrow \text{LOVE}(x,z))$
   $(\mathsf{A@therapist})@\lambda\mathsf{z}.[(\mathsf{A@s\_cat})\lambda\mathsf{y}.[(\mathsf{Every}@\lambda\mathsf{x}.[\text{OWNER}(\mathsf{x}) \wedge \text{OF}(\mathsf{y},\mathsf{x})])@\lambda\mathsf{x}.\text{LOVE}(\mathsf{x},\mathsf{z})]]$

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.

Exercise 5.5 *Give natural language paraphrases for the first four readings (try to make them as unambiguous as possible).*

5.1.4 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.

Exercise 5.6 *Give a natural language paraphrase for the fifth reading and compare it to the paraphrase for the fourth one. Can you explain why both readings are equivalent by examining the corresponding formulas?*

...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. $(\mathsf{A}@\mathsf{company})\lambda x.[(\mathsf{Most}@\mathsf{sample})@\lambda y.[(\mathsf{Every}@\lambda z.[\textsc{researcher}(z) \wedge \textsc{of}(z,x)])@\lambda z.\textsc{see}(z,y)$
2. $(\mathsf{Most}@\mathsf{sample})@\lambda y.[(\mathsf{A}@\mathsf{company})\lambda x.[(\mathsf{Every}@\lambda z.[\textsc{researcher}(z) \wedge \textsc{of}(z,x)])@\lambda z.\textsc{see}(z,y)$

Exercise 5.7 *Do you see the difference (in meaning) between the two readings? Give a natural language paraphrase for each of the readings and try to think of contexts that would favour one or the other. Can you explain why the ordering of the representations of the two determiners* A *and* Most *makes a difference in this case (in contrast to the corresponding determiners in our siamese-cat-example)?*

5.1.5 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 $\forall x.(\textsc{man}(x) \rightarrow (\exists y.\textsc{woman}(y) \wedge \textsc{love}(x,y)))$, 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.

5.1.6 Quantifying In: An Example

Let's look at our example from before (page 79): 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.

5.1.7 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 (page **??**). 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.

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

Exercise 5.8 *Do you see what's problematic about this formula? Try to give a natural language paraphrase for it. What could have gone wrong in a semantic construction process that has led to this formula for our example sentence?*

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*). 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.

5.1.8 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.

5.2 Underspecification

5.2.1 Introduction

A Clean and Declarative Approach

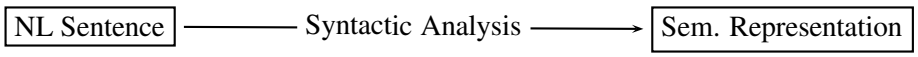
So basically, we are going to *separate semantic construction from the enumeration* of readings of ambiguities. We thus divide the problem into two independent parts, which we can in turn solve independently. This means we can stick to our original setup, where we derive *one* representation from *one* 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'.

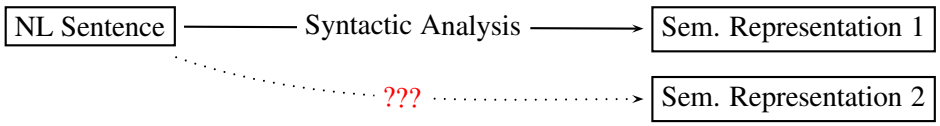
Standard Montague



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:

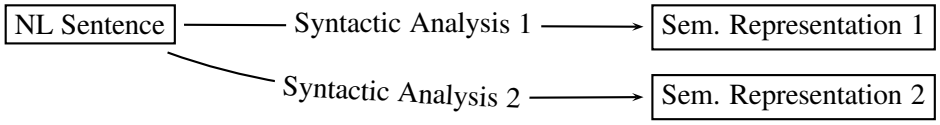
The Problem



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.

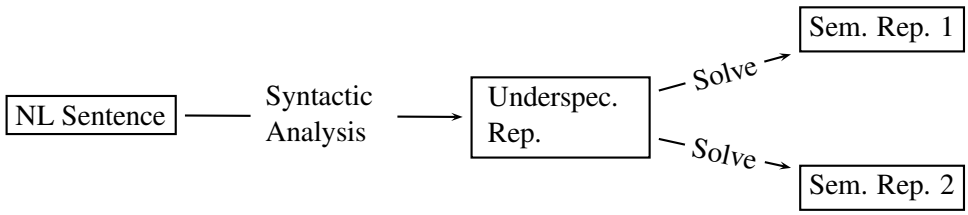
Montague with Quantifying In



Now we would be able to construct the second semantic representation together with this second syntactic analysis. As we've said (in Section 5.1.5), 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.

5.2.2 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 (we're going to hear about first-order deduction in Chapter 10) 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.

5.2.3 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. $\forall x.\text{MAN}(x) \rightarrow (\exists y.\text{WOMAN}(y) \wedge \text{LOVES}(x,y))$
2. $\exists y.\mathsf{woman}(y) \wedge (\forall x.\mathsf{man}(x) \rightarrow \text{LOVE}(x,y))$

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 $\text{LOVE}(x,y)$, 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 arranging them into complete formulas. What is left underspecified is which of these readings is the "right" reading of a specific utterance of the sentence.

5.2.4 The Masterplan

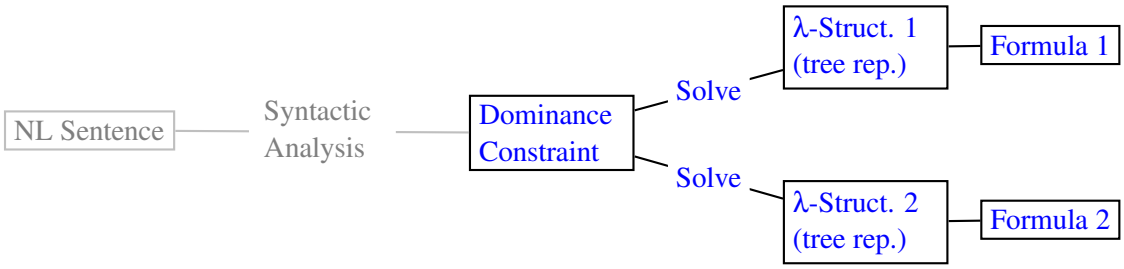
In the rest of this chapter and in the next one, we will go into the details of underspecification. What exactly are we going to do? We will first give you an intuition of what the formalism to be presented does, and then make this intuition more formal. Here's how we will proceed:

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 5.2.5), we'll explain how to do this.

2. Then (in Section 5.2.6) we will introduce a formalism that allows us to *describe* trees (and thereby formulas and λ-expressions). This formalism is called *normal dominance constraint* s. 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 5.2.7) 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? In our formalism of normal dominance constraints, 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 5.2.8). In the next chapter, we will then continue our discussion by formulating an algorithm that incorporates this intuition. Section 6.1 introduces the basic concepts used in that algorithm. In Section 6.2 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 5.2.1. 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 6.4), 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.

5.2.5 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!

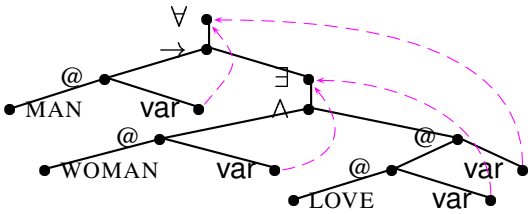
Finally, on the level of atomic formulas, we shall from now on write application of predicates to arguments with the binary symbol @. (We have already seen this: We indicate applications in λ-calculus the same way). Here's one of our standard example formulae in this notation (if you're interested in the motivation behind our decision to use this new notation, read the sidetrack (page 92) at the end of this chapter):

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

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

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

Testtitel



?- 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.

5.2.6 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 5.1.2, and repeat them here:

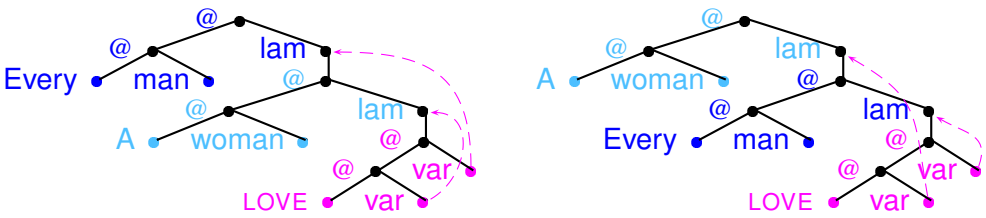
i. $(\mathsf{Every}@\lambda v.(\textsc{man}@v))@(\lambda x(\mathsf{A}@\lambda w.(\textsc{woman}@w))@(\lambda y.((\textsc{love}@y)@x)))$

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...

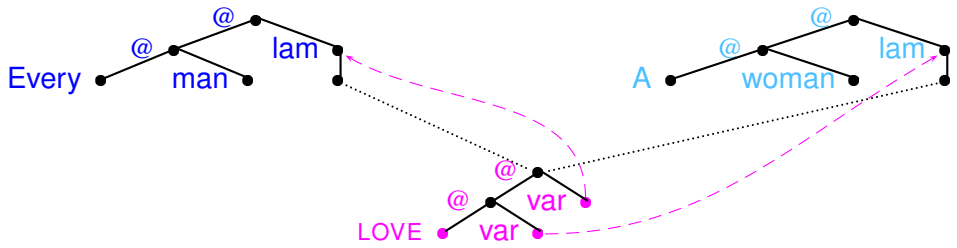
If we represent the two readings of our example as λ-structures, we can identify the three formula fragments relevant for the scope ambiguity we're interested in as three tree fragments. We have given them different colours in the picture below.



Exercise 5.9 *Write down (at least) one of the two readings as a "normal" λ-expression of the form* $(\lambda x.\textsc{walk}@x)@\textsc{john}$. *Remember that you have to "invent" a variable for each* lam*! If you do not feel familiar with this notation yet, replace the @s by bracketting, e.g.* $(\lambda x.\textsc{walk}(x))(\textsc{john})$. *Compare the way the transitive verb is combined with its arguments here with the way this was done in λ-calculus. Give a short comment on what you think is the main difference between the two approaches.*

...but only one constraint graph

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



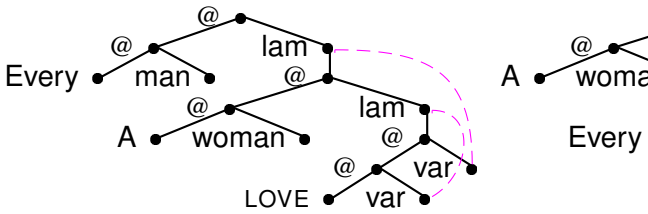
We call a graph as in this picture a *constraint graph* . A constraint graph is a directed graph that has node labels and three kinds of edges: ordinary solid edges, dotted *dominance edge* s, and purple arrow *binding edge* s. It consists of several little tree fragments which are internally connected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

5.2.7 From Lambda-Expressions to an Underspecified Description

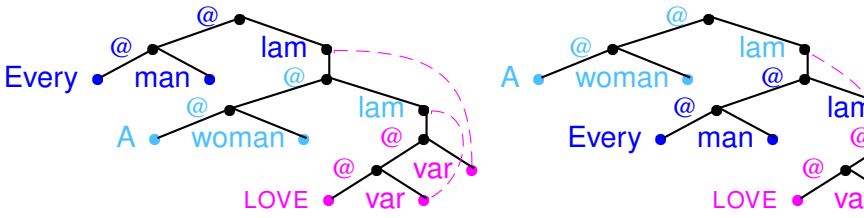
Let us look at our example once more and go through *step by step* how we have constructed the constraint graph describing our two λ-expressions. We had to take four steps:

1. We wrote down all (two) readings of the sentence, as λ-expressions:1. $(\mathsf{Every}@\lambda x.\textsc{man}(x))@(\lambda x.(\mathsf{A}$
$(\mathsf{A}@\lambda y.\textsc{woman}(y))@\lambda y.(\mathsf{Every}@\lambda x.\textsc{man}(x))@(\lambda x.\textsc{love}(x,y))$

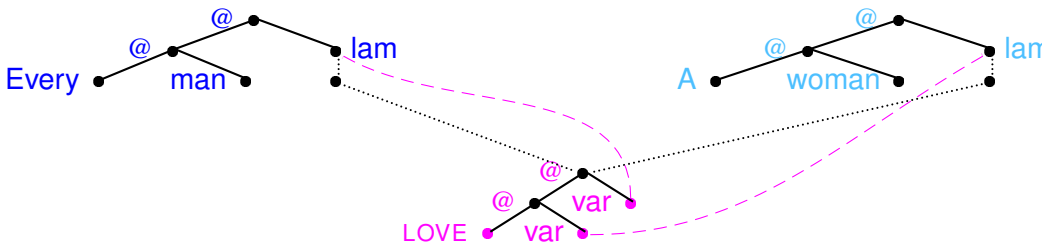
2. We converted the readings into λ-structures:



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 85). 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.

Exercise 5.10 *Write down a constraint graph that describes the five readings of the sentence 'Every owner of a Siamese cat loves a therapist.'*

5.2.8 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 arrange 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.

Exercise 5.11 *Convince yourself that the constraint graph you've seen in the example really describes* both *of our λ-structures. Check each of the points in the above definition.*

5.2.9 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.

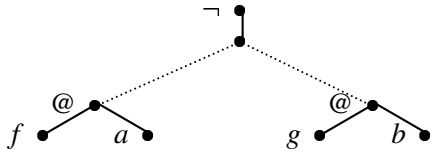
More Flexibility

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. Arranging

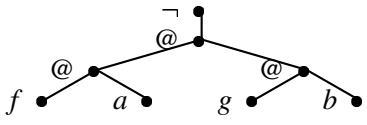
Here we will be ignoring this point, and eventually it will turn out that we'll never need to invent any additional material to solve the constraint graphs we get for scope ambiguities anyway. It is safe to think of the process of solving a constraint as *arranging* the fragments into a bigger tree.

The difference between embedding and arranging 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 arrange 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 arrange them by attaching the other fragment to this leaf. Otherwise, it is impossible to arrange them; this is the case in the example. If we take an example of larger scale, of course, we might be able to start with the arrangement process, but then notice somewhere down the line that we have plugged fragments together in the wrong order. That's why computation becomes much harder when we restrict ourselves to arrangement 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 constraint* s. In fact, if you look at the literature on dominance constraints, you'll find that the logical formulas are always the first concept to be defined, constraint graphs being then derived from them. If you'd like to know the *true* true story about constraint graphs, you can ask[1] us for literature on dominance constraints.

5.2.10 Sidetrack: Predicates versus Functions

When we introduced our tree notation for formulas (in Section 5.2.5) 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

As you may remember from the sidetrack on typed λ-calculus, 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

[1] mailto:koller@coli.uni-sb.de

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 recall what we've said about the interpretation of λ-expressions in Section 3.4.4, 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:

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

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:

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

We will make use of this simplification in our implementation of CLLS (see in particular Section 6.3.1).

6

Constraint Solving

6.1 Constraint Solving

6.1.1 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 5.2.9 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.

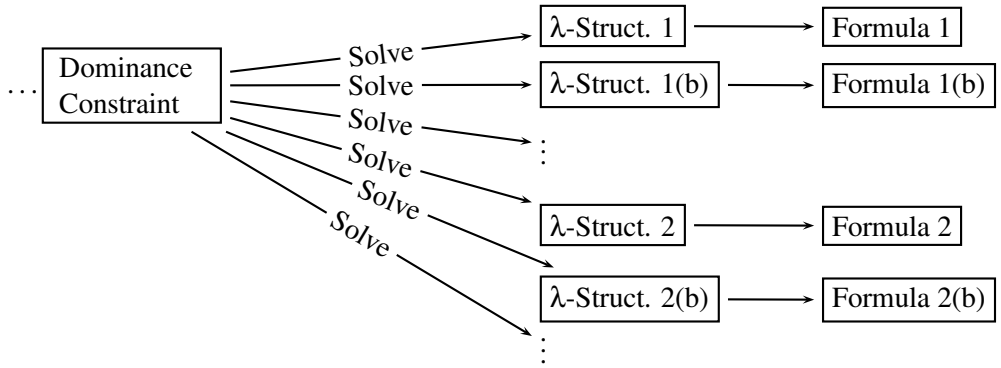
It is clear that the first problem is simpler than the second one. Whenever you have an algorithm with which you can enumerate all solutions, you can just stop when you have found the first solution, and say that the constraint is indeed satisfiable. And if it turns out that you just can't find a solution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

6.1.2 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:



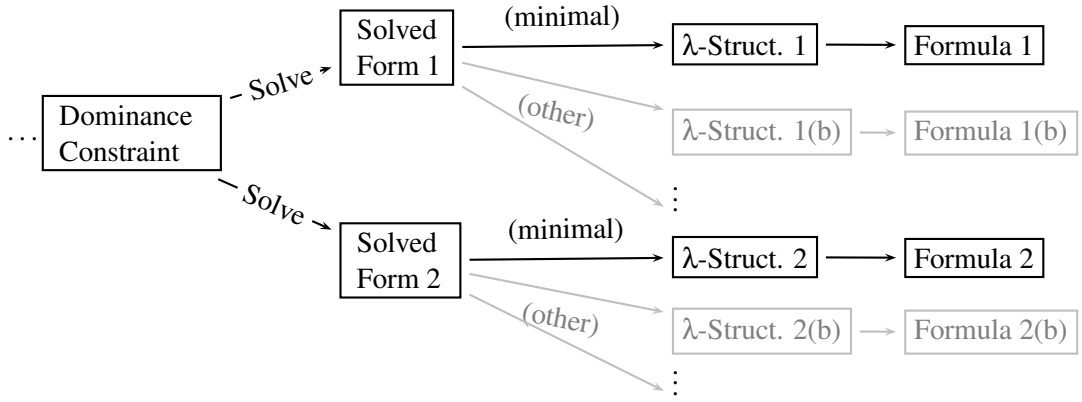
In this example, we might only be interested in the two solutions 1 and 2 while the "variants" 1(b) and 2(b) as well as all other variants with additional material inbetween are pointless to us.

...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.

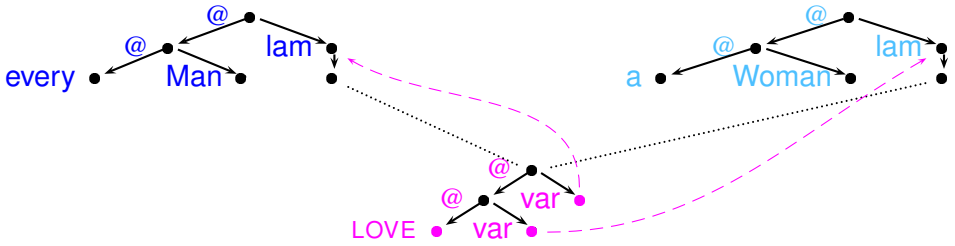
Before we look at an example, let's refine the scheme of our architecture (page 85) once more:



6.1.3 Solved Forms: An Example

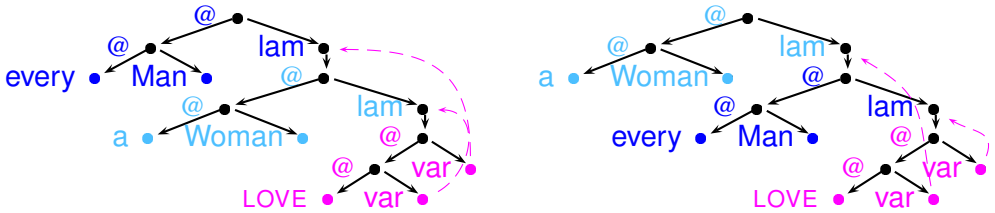
A constraint graph...

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).



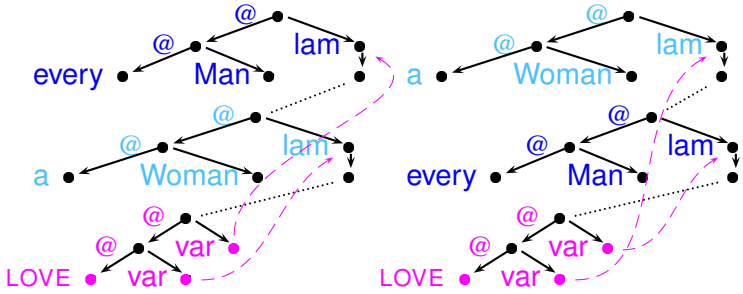
...two of its solutions...

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.

6.1.4 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:

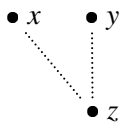
1. If we're lucky, the solved form will not contain any nodes with two outgoing dominance edges anyway. In this case, we can simply identify the endpoints of each dominance edge, and we obtain a minimal λ-structure that satisfies the solved form. As it happens, all of the solved forms we'll get for underspecified semantics will belong to this class.
2. Otherwise, we could do the same operation as in Section 5.2.9: 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.

6.2 An Algorithm For Solving Constraints

6.2.1 The Choice Rule

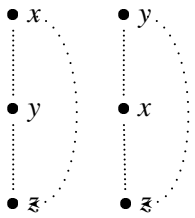
The key insight that we exploit in the algorithm is the following. Suppose you have a node with two incoming dominance edges:



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.

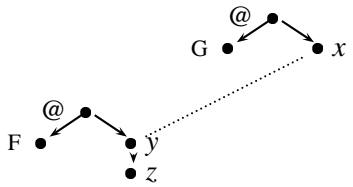
6.2.2 Normalization

Cleaning up

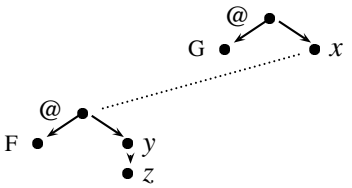
The Choice Rule is the driving force behind the enumeration process: It resolves one node that keeps the constraint from being in solved form by adding additional dominance edges. However, the Choice Rule can't operate on its own. It needs some helpers that clean up after it has done its job. This cleaning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



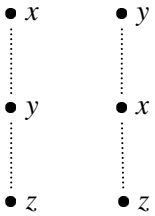
Fortunately, we can resolve this configuration easily. The key observation is that X and Y cannot be mapped to the same node in a solution, as their parents must be different. Thus we can infer that X must dominate not just Y, but the parent of Y:



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?

6.2.3 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.

6.3 Constraint Solving in Prolog

6.3.1 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`.

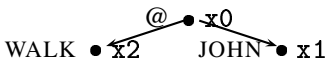
Dominances and Bindings

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],[],[]).
```

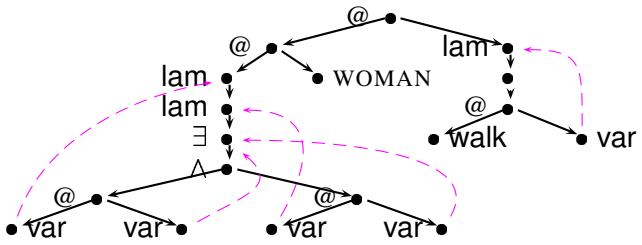
In the more familiar tree representation:



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 5.2.10 to see why this is possible.

6.3.2 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 6.2.3. 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 98)). 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.

6.3.3 Distribute

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

The predicate `distribute/3` (from `solveConstraint.pl`) implements the Distribution Rule (or Choice Rule (page 98)). 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,[dom(
        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)`.

6.3.4 (Parent) Normalization

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

What remains to be implemented now is the normalization of constraints (see Section 6.2.2). 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)`.

In !!!UNEXPECTED PTR TO EX_EX.EX.CLLS.MEMBER4SELECT!!!, you can try to reimplement this predicate using a simple member check instead of `select`.

6.3.5 Redundancy Elimination

The second normalization step described in Section 6.2.2 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.

6.4 Semantics Construction for Underspecified Semantics

6.4.1 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.

6.4.1.1 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 6.3.1 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.

6.4.1.2 Macros for the Determiners

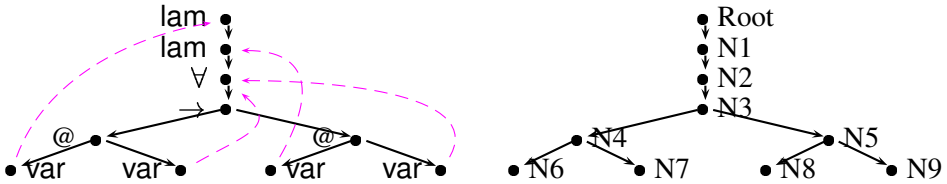
***See file* `clls.pl`.**

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

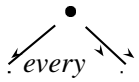
```
detSem(uni,usr([Root,N1,N2,N3,N4,N5,N6,N7,N8,N9],
        [Root:lambda(N1),N1:lambda(N2),N2:forall(N3),N3:(N4 > N5),
         N4:(N6@N7),N5:(N8@N9),N6:var,N7:var,N8:var,N9:var],
        [],
        [bind(N6,Root),bind(N7,N2),bind(N8,N1),bind(N9,N2)])).
```

```
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 $\lambda P \lambda Q. \forall x(P@x \rightarrow Q@x)$. 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).



!!!UNEXPECTED PTR TO EX_EX.CLLS.QUANTUSR!!! lets you test whether you understood these representations. 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. In !!!UNEXPECTED PTR TO EX_EX.CLLSEX1!!! you are asked to simplify the semantic macros in our implementation dramatically using the same trick.

6.4.2 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`. Best you do !!!UNEXPECTED PTR TO EX_EX.CLLS.MERGEUSR!!! right away in order to become familiar with this predicate.

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` → `PN`:

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

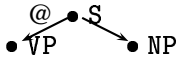
We present the more complex `combine`-rules below, except for one (namely N → N Adj). We leave the formulation of this rule to you as !!!UNEXPECTED PTR TO EX_CLLSEX2!!!.

6.4.2.1 S → 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.



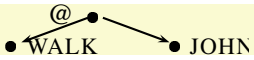
As you can see, the new constraint graph describes λ-structures in which the VP meaning is applied to the NP meaning. This is basically like our very first naive analysis of how NPs and VPs are combined semantically. It still works because the trick that we developed in order to get a uniform treatment 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, $walk(john)$.

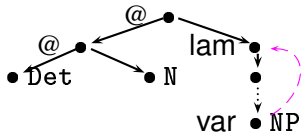
6.4.2.2 NP → Det N

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).
```

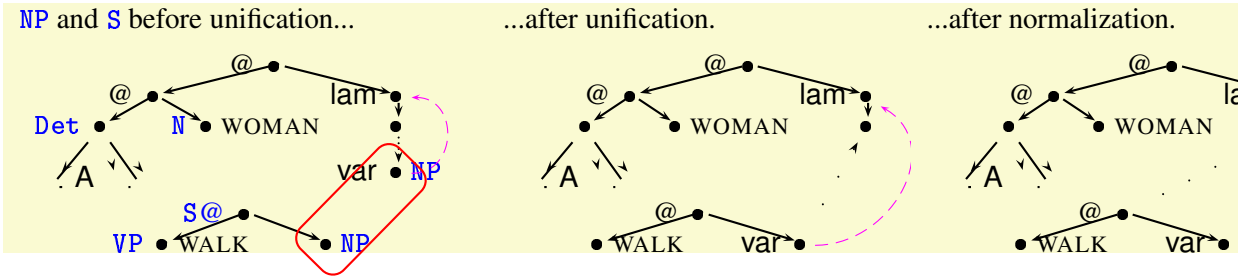
Again, this becomes more readable when written as a constraint graph:



If you doubt that the clause of `combine/2` given above really corresponds to the tree representation we presented, you should do !!!UNEXPECTED PTR TO EX_EX.CLLS.COMBINE!!! right now.

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 6.4.2.1). 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:

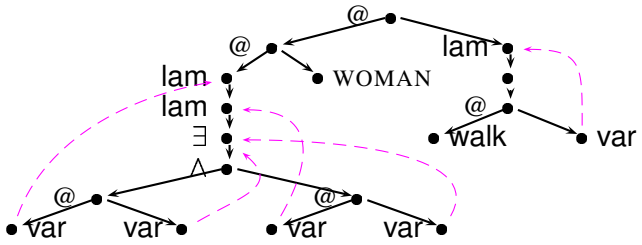
$$\mathsf{A}@\textsc{woman}@(\lambda x.\textsc{walk}@x)$$

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 6.4.2.1. 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.

6.4.2.3 Example: 'A woman walks'

No Comment

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. This is the first part of !!!UNEXPECTED PTR TO EX_EX.CLLS.BEAUTY!!!.

6.4.2.4 `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 6.4.2.1). 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),N
        [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 6.4.1.2)!

6.5 Running CLLS

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 4.5. 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 forms in
See file `englishGrammar.pl`.	The DCG-rules and the lexicon (using module See file `englishL`
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.

6.6 Exercises

Exercise 6.1 *Make sure that the clause of* `combine/2` *given in Section 6.4.2.2 really corresponds to the tree representation given there: Print out the tree and try to decorate its nodes with all respective Prolog variables for the nodes.*

Exercise 6.2 *1. Write down the λ-expression that corresponds to the graph given in Section 6.4.2.3. Reduce this λ-expression to a first order formula.*

Hint: the right half translates into $\lambda x.\text{WALK}@x$. In the left half, there are two lam*-nodes. Each translates into λP_i where P_i is a new variable. The existential quantifier has to be treated likewise.* var*-nodes translate into the variables bound by their binder.*

2. Look at our implementation of the predicate `translate/4` *(see* `cllsLib.pl`*) that translates solved constraints into λ-terms. Explain how the bindings are handled!*

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

This predicate converts solved constraints directly to λ-terms. Note that we only compute one (minmal) solution for one solved form. But as we have discussed (page 95) solved forms stand for classes of solutions. Which clause of our predicate would we have to change in order to compute non-minimal solutions?

Exercise 6.3 *Test whether you understand the semantic macro for the indefinite determiner* a*. The "naked" tree given in the figure in Section 6.4.1.2 on the right is already decorated with the PROLOG variables from the semantic macro* `detSem(indef,...)`*. Print out this tree or draw it on a piece of paper. Go through the semantic macro and add the node labels and the binding edges to the tree.*

Exercise 6.4 ***See file*** **`cllsLib.pl`.**

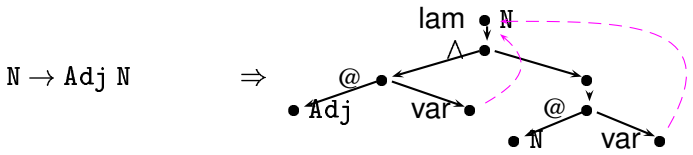
Look at the two clauses of `mergeUSR/2 (cllsLib.pl)` *and make sure that in the resulting constraint, the root node of the leftmost input constraint really ends up in front of all other nodes. Remember that we always interpret the first node of a constraint as its root node. This means we have to be able to formulate our calls of* `mergeUSR/2` *such that we can determine which node will finally be the root node.*

Design a test call

```
USR_A = usr([nodeA1,nodeA2,...],[nodeA1:lA1,...]...),
USR_B = usr(nodeB1,...),
USR_C = usr(...),
mergeUSR(merge(merge(USR_A,USR_B),USR_C),Result).
```

to practically test this.

Exercise 6.5 *Add a treatment of adjectives to the implementation of clls. The semantic macro for an adjective is a USR for a simple labeled node. Procede as follows: First, complete the semantic macro* `adjSem(_,_)` *for adjectives (to be found at the bottom of* `clls.pl`*). Second, add a clause of* `combine` *to* `clls.pl` *according to the following scheme.*



Exercise 6.6 ***See file*** **`solveConstraint.pl`.**

The implementation of `liftDominanceConstraints/2` *we presented immediately removes the old, lifted dominance edge with* `select/3` *(see Section 6.3.4).*

There is a problem if we use `member/2` *instead of* `select/3`*? Try it out! Why does this problem occur? Can you think of an elegant solution to it?*

Hint: look what predicates we've already got for cleaning up constraints!

Exercise 6.7 *[optional]*

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

In the implementation of clls we presented, the meaning of a determiner as given in the semantic macros is a relatively complex λ*-structure (tree). It represents e.g. the* λ*-term* $\lambda P \lambda Q. \forall x(P(x) \rightarrow Q(x))$ *in the case of the universal quantifier. The only part of such trees that is relevant for the semantics construction process is its root node. The rest of the tree remains stable during the whole construction process: There is no place inside these trees where further material can be inserted. To enhance the readability of the underspecified structures, one can for example represent the meaning of an universal quantifier as a node labeled* every *during the semantics construction process. Then,*

before the final β*-reduction is done, these abbreviations can be be expanded into the well-known* λ*-terms like* $\lambda P \lambda Q. \forall x(P@x \rightarrow Q@x)$.

Try to change the implementation of clls accordingly. You may proceed as follows. First, replace the entries of the determiners in the semantic macros by simple labeled nodes (like the entries of nouns for example). Then, build a PROLOG module `ExpandQuantifiers` *that exports a predicate* `ExpandQuantifiersList/2`. *This predicate should expand a list of* λ*-terms containing abbreviations like* `[every@man@lambda(A,walk@A)]` *into the list* `[lambda(P, lambda(Q, exists(X,(P@X)&(Q@X))))@man@lambda(A,walk@A)]` *Finally, integrate the predicate* `ExpandQuantifiersList/2` *into the predicate* `clls`.

Hint: use the predicate `compose/3` *(*`comsemLib.pl`*) to decompose terms like* `every@man@lambda(A,walk(`
.

Exercise 6.8 *[Mid-Term Project]*

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

Our implementation of `solve/2` *is a straightforward implementation of the enumeration algorithm. It is an example of* imperative *programming. For example, the solutions of two distributed constraints are computed at once and then they are solved recursively. Finally, the results are appended.*

The same could have been achieved in a more declarative *fashion with backtracking and two clauses of* `distribute`. *Reimplement the predicate in a more declarative manner!*

Hint: use Prolog search (like `bagof`*) to enumerate all solutions.*

7

Inference in Computational Semantics

7.1 What is Inference, and how do we use it in Computational Semantics?

7.1.1 What we already know about Logics

We have already learned a lot about the syntax and semantics of first-order logic. But logical systems usually have a third component - a *calculus*. Before we look in detail at this third component, we briefly recap the key logical concepts that we have seen so far.

Formula	sequence/tree of symbols	$x, y, f, g, p, 1, \pi, \in, \neg, \wedge, \forall, \exists$
Model	something we understand	natural numbers or sets
Interpretation	maps formulae into models	$[\![\text{three plus five}]\!] = 8$
logical consequence	$A \models B$, iff $\mathcal{M} \models B$ for all $\mathcal{M} \models A$	

Why is logic useful for us?

Logic studies formal languages and their relation to the world. This task is closely related to our task in computational semantics, i.e. computing the meaning of natural language utterances. Two of the advantages of using logic for this task are that logics are are mathematically precise and relatively simple.

Consider the following points where we have already gained a lot from this precision and simplification:

Formulae of formal languages (such as the language defined by the syntax of first-order logic (page 9)) simplify sentences of natural languages: Problems of grammaticality no longer arise. Furthermore, well-formedness can in general be decided by a simple recursive procedure.

Models are what we use as a simplification of real-world situations. Models (page 6) simplify the real world by concentrating on mathematically well-understood structures, namely sets and relations. They allow us to make predictions about truth conditions of natural language sentences. Moreover they make it possible to precisely define semantic notions, such as logical consequence (page 15).

7.1.2 Calculi

From a theoretical perspective, talking about models (as we did it in Chapter 1) enables us to *define* neatly e.g. which sentences follow from a particular sentence. 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.

Assumptions, Inference, Conclusions

Therefore, logics typically have a third part, an *inference system* (also called a *calculus*), which is a syntactic counterpart to the semantic notions. By saying that it is syntactic, we mean that such a system works without recourse to meaning, by considering only the *syntactic structure* of formulae. Hence a calculus gives us the syntactic counterpart to the semantic concept of logical consequence (page 15). Formally, a calculus is a set of rules that transform (sets of) formulae into other (sets of) formulae. The formulae given as input are called *assumption* s, the resulting formulae are called *conclusion* s. In what follows, we will use a *tableaux calculus* to infer what formulae follow from some given other ones.

Proof

But before that, let us look at the syntactic counterpart of validity (page 14). A sequence of rule applications that transform the empty set of assumptions into a formula $\mathbf{A}$, is called a *proof* of $\mathbf{A}$. To make this clear, let us turn to a very simple example.

7.1.3 A simple Logical System: Propositional Logic with Hilbert-Calculus

In this section we will discuss the simplest non-trivial fragment of first-order logic imaginable: A formal language which has only one constant symbol, and where atomic formulae (page 9) have been abbreviated to propositional variables. We will call this logic PH.

We have chosen this logic so that talking about it is simple, not that it covers any interesting fragment of natural language. We want to make clear certain concepts that will become important later, but we don't want to go through the exercise of formulating a calculus for a more expressive logic yet. Still you needn't worry, we will do so soon!

As discussed above, a logic has three components:

Syntax

The set of well-formed formulae of PH is built from *propositional variables*: $P, Q, R, \ldots$ and the logical operator of implication: $\Rightarrow$

Semantics

The semantics of PH is determined by the following rules:

- $[\![P]\!]^{\mathcal{M}}_g = g(P)$
- $[\![\mathbf{A} \Rightarrow \mathbf{B}]\!]^{\mathcal{M}}_g = \mathtt{T}$, iff $[\![\mathbf{A}]\!]^{\mathcal{M}}_g = \mathtt{F}$ or $[\![\mathbf{B}]\!]^{\mathcal{M}}_g = \mathtt{T}$.

Calculus

The calculus is given by specifying its inference rules. Inference rules are generally written as schemata of the following form:

$$\frac{\mathbf{A}_1 \quad \cdots \quad \mathbf{A}_n}{\mathbf{C}} \mathit{Name}$$

Here the $\mathbf{A}_i$ and $\mathbf{C}$ are (schematic) formulae and *Name* is the name of the inference rule. The rule signifies that whenever all of the formulae $\mathbf{A}_i$ (the so-called called *antecedent* s or *premise* s) have already been derived, the formula $\mathbf{C}$ (the *succedent* or *conclusion*) can also be derived.

Here are the four inference rules of our calculus:

$$\frac{}{P \Rightarrow (Q \Rightarrow P)} K \qquad \frac{}{(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))} S$$

$$\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}} MP \qquad \frac{\mathbf{A}}{[\mathbf{B}/P]\mathbf{A}} \mathit{Subst}$$

The first two inference rules are special cases, where the antecedents are empty. Such inference rules are commonly called *axiom* s. What makes them special is that their conclusions can be derived without assumptions, that is at *any* time. The rule MP is sometimes called *modus ponens* ; it codifies our intuition about the implication operator. The rule *Subst* allows to substitute arbitrary well-formed formulae for propositional variables. We will use the notation $[\mathbf{A}/X]\mathbf{B}$, for the application of the substitution $[\mathbf{A}/X]$, to the formula $\mathbf{B}$. This replaces all occurrences of the propositional variable X in $\mathbf{B}$ with the formula $\mathbf{A}$. For example $[(P \Rightarrow Q)/R](R \Rightarrow R)$ is $(P \Rightarrow Q) \Rightarrow (P \Rightarrow Q)$.

7.1.4 Proofs in Hilbert Calculus

Proof

An occurence of a rule schema where all schematic formulae (like $\mathbf{A}$, $\mathbf{B}$ etc.) have been replaced by actual formulae of the language is called an *instance* of that schema. A *proof* with our little calculus is simply a sequence of instances of the inference rule schema, such that the antecedents of each rule are succedents of rules earlier in the sequence. If we additionally allow formulae from a set $\mathcal{H}$ into the sequence, we speak of a proof from the assumptions in $\mathcal{H}$. Any formula that occurs in a proof generated from our Hilbert calculus is *provable* (or provable from the assumptions in $\mathcal{H}$ if it occurs in a proof from these assumptions). So intuitively, if we want to prove a formula, we take a (possibly empty) set of assumptions and apply the inference rules until we have produced our formula as the conclusion of some rule application.

Theorem

We will write $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{A}$ if the formula $\mathbf{A}$ is provable from the assumptions in $\mathcal{H}$ in the calculus $\mathcal{C}$. We will call the relation $\vdash_{\mathcal{C}}$ the *derivability* or *provability* relation for $\mathcal{C}$.

We will simply write $\vdash_{\mathcal{C}} \mathbf{A}$ if $\mathbf{A}$ is provable from the assumptions in $\mathcal{H}$ (so $\vdash_{\mathcal{C}} \mathbf{A}$ a shorthand for $\emptyset \vdash_{\mathcal{C}} \mathbf{A}$). A formula that is provable without any further assumptions is called a *theorem* .

An Example Proof

Let us look at an example to fortify our intuition. We will prove the theorem $\mathbf{A} \Rightarrow \mathbf{A}$.

1	$(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$	S
2	$(\mathbf{A} \Rightarrow (Q \Rightarrow R)) \Rightarrow ((\mathbf{A} \Rightarrow Q) \Rightarrow (\mathbf{A} \Rightarrow R))$	$Subst(1)$ with $[\mathbf{A}/P]$
3	$(\mathbf{A} \Rightarrow ((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow R)) \Rightarrow ((\mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A})) \Rightarrow (\mathbf{A} \Rightarrow R))$	$Subst(2)$ with $[(\mathbf{A} \Rightarrow \mathbf{A})/Q]$
4	$(\mathbf{A} \Rightarrow ((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A})) \Rightarrow ((\mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A})) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A}))$	$Subst(3)$ with $[\mathbf{A}/R]$
5	$P \Rightarrow (Q \Rightarrow P)$	K
6	$\mathbf{A} \Rightarrow (Q \Rightarrow \mathbf{A})$	$Subst(5)$ with $[\mathbf{A}/P]$
7	$\mathbf{A} \Rightarrow ((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A})$	$Subst(6)$ with $[(\mathbf{A} \Rightarrow \mathbf{A})/Q]$
8	$(\mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A})) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A})$	$MP(4,7)$
9	$\mathbf{A} \Rightarrow (Q \Rightarrow \mathbf{A})$	$Subst(5)$ with $[\mathbf{A}/P]$
10	$\mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{A})$	$Subst(9)$ with $[\mathbf{A}/Q]$
11	$\mathbf{A} \Rightarrow \mathbf{A}$	$MP(8,10)$

There are two things to note about this proof.

- The inference rules can be applied in a purely schematic and syntactic fashion. It does not play a role what the formulae that are manipulated actually mean.
- The actual proof proof is long, tedious, and in this particular case very unintuitive.

The last observation is negative. But it mainly has to do with the particular calculus we're using. We will discuss a more intuitive calculus (called semantic tableaux) later (page 121).

The Axiomatic Method

The first observation in contrast is a positive one. Indeed it is the great pro of using calculi (or the *axiomatic method* , as one also says): Instead of thinking about infinitely many models for *validity* , we can consider *provability* , which can be exhibited by pure syntactic formula manipulation. The only question that remains is how provability and validity relate to each other. In other words: We want to *justify* our calculus in terms of semantics again.

7.1.5 Properties of Calculi (Theoretical Logic)

In this section we will examine the relation between provability ($\vdash$) and validity ($\models$) (and more generally between derivability and logical consequence). Both are relations on formulae. Ideally, they would co-incide, but there is no a-priori guarantee for that. The two key notions with respect to this are:

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

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

If a calculus is correct and complete, provability and validity coincide ($\vdash \mathbf{A}$ iff $\models \mathbf{A}$). As a consequence, syntactic formula manipulation can totally replace semantic considerations.

The calculus presented in Section 7.1.3 is in fact correct and complete. Correctness is easy to determine with the methods from Section 1.1.2. We just need to establish that the succedent is a logical consequence of the antecedents for each inference rule. For the axioms K and S this means that they have to be valid formulae (for all P, Q, R), and MP and *Subst* that all interpretations that make the antecedents true also make the succedent true. In fact, this is the case. Establishing completeness is much more involved, and is beyond the scope of this course.

7.1.6 Sidetrack: Calculemus!

Greek Philosophers like Aristotle already realized that the validity of arguments at a certain level does not depend on any particular circumstances of the world. For example, the argument

'Socrates is human'. 'All humans are mortal'. 'Therefore Socrates will die'.

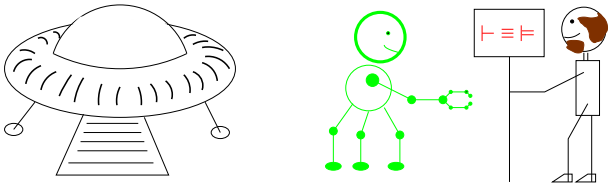
is valid irrespective of facts such as aging or the inadvisability of accepting drinks from your enemies. As we would put it, this is due to the coincidence between logical consequence and derivability. The only thing that matters is the syntactic form of the argument. Whenever we know '**A** is $\mathcal{P}$.' and 'All $\mathcal{P}$ are $\mathcal{Q}$.' then we can conclude '**A** is $\mathcal{Q}$.'

Calculemus!

Later, in the 18^{th} century the German philosopher Gottfried Leibniz (getting tired about the endless debates of his colleagues about questions like "how many angels can dance on a pinpoint") dreamed of a formal language that could express all of natural language (*lingua universalis*) and a calculus (*calculus ratiocinator*) that could compute all truths of this language. He wanted to be able to call out to his colleagues *Calculemus!* (let us just calculate who is right).

Today the discovery that syntactic and semantic determination of truth conditions can be equivalent under certain conditions is still considered as one of the great achievements of the human mind.

In fact if one day a committee of aliens lands on planet Earth to determine whether they should just blast it away to make way for a new intergalactic hyper-way, or if the human race is intellectually advanced enough to be worth saving, then we should try to bring this fact to their attention in order to save our planet (provided we're on the welcome comittee).

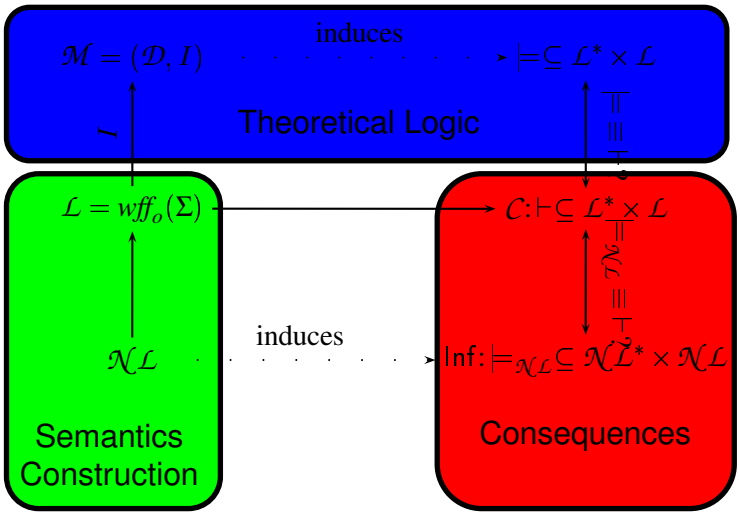


7.1.7 Natural Language Semantics

Let us now relate all of this to what we've learned so far about natural language semantics. Our overall aim is to develop a set of methods to determine the meaning of natural language. If we look back, all we did was semantic construction. We established translations from natural languages into formal languages like first-order logic (and that is all you will find in most semantics papers and textbooks). Now, we have just tried to convince you that formulae of these formal languages are themselves again nothing but syntactic entities that can be processed on syntactic grounds in calculi. So, *where is the semantics?* In fact, we have got it once we have a correct and complete calculus.

Why is this true? Basically, our argument goes as follows: The semantics of a sentence is reflected in what follows from it. The translation methods we already have give us first-order logical formulas for sentences. Now we said above that we take the *logical consequence* relation between such first-order formulae as a simplified version of the 'follows from' relation between the sentences they translate. Now, a complete and correct calculus captures this logical consequence relation between formulae - in terms of their syntax. So it also captures the 'follows from' relation between the corresponding sentences, and thus their semantics, in syntactic terms.

Let us consider the following diagram to shed more light on this situation:



As we mentioned, the green area is the one generally covered by natural language semantics (and all that we covered in the preceding chapters). In the semantics construction process, the natural language utterances (viewed here as 'formulae' of a language $\mathcal{NL}$) are translated to a formal language $\mathcal{L}$ (in our case first-order logic). Now the argument given above shows that this is all that is needed to recapture the semantics of $\mathcal{NL}$. Even though it is not immediately obvious at first: Theoretical logic gives us the missing pieces.

Let us reformulate our argument in the terminology used in the diagram. Since $\mathcal{L}$ is a formal language of a logical system, it comes with a notion of model and an interpretation function I that translates $\mathcal{L}$ formulae into objects of such models. Models induce

a notion of logical consequence[1] as explained in Section 1.1.2. The formal language $\mathcal{L}$ also comes with a calculus $\mathcal{C}$ acting on $\mathcal{L}$-formulae, which (if we are lucky) is correct and complete (then the mappings in the upper rectangle commute[2]).

In natural language semantics we are interested in 'follows from' relations on natural language utterances, which we have denoted by $\models_{\mathcal{NL}}$. If the calculus $\mathcal{C}$ of the logic $\mathcal{L}$ is correct and complete, then it is a model of the relation $\models_{\mathcal{NL}}$. In this case we really have a formal handle on the 'follows from' relation for natural language utterances if we only specify our semantics construction (the green part) method and the matching calculus.

7.2 Tableaux Calculi

7.2.1 Tableaux for Theorem Proving

Tableaux are data structures used in refutation-based procedures for automated theorem proving. Tableaux are trees labeled with formulae. They are constructed starting from a single (root) node, which is labeled with an input formula. From this node, a tree is constructed by recursively decomposing complex formulae along the lines of their logical connectives. Finally, the literals (page 10) on each branch of this tree tell us what has to be true (and what has to be false) in some class of models of the input formula, and (this is an important point) all branches of the tableaux taken together represent *all* models of the input formula.

Closed Tableaux: No Model

If a branch of a tableaux contains a contradiction (e.g. the literals q and $\neg q$), this branch is called *closed* . Branches that are not closed are called *open* . No models correspond to a closed branch (because two contradictory literals cannot be true in any model). So the nice thing is that if *all* branches of a tableaux are closed, we're sure that there is *no* model at all for its input formula. This is a direct consequence of the fact that a tableaux enumerates all models of the input formula.

Here's how we can make use of this fact in theorem proving: When trying to prove that a formula is a theorem, we want to show that the formula is true in *all* models. But we needn't do that directly. As you know, it is a common technique in mathematics to prove something by showing that its negation cannot be true. This is exactly how tableaux proofs work: If we want to show that a formula is a theorem, we prove that there is no way to make it false. So in order show that a given input formula is true in all models, we will show that there is *no model for its negation*.

Constructing a Tableaux Proof

Now what does all of this mean in terms of tableaux? We start with the negation of our suspected theorem, and construct a tableaux by decomposing it into subformulas step

[1] Relations on a set S are subsets of the Cartesian product of S, so we use $R \in S^* \times S$ to signify that R is a (n-ary) relation.

[2] We say that arrows in a diagram commute, if we can compose the mappings on any path and always arrive at the same result.

by step. To prove that it really is a theorem if we have to arrive at a closed tableaux at some point.

But how exactly do we construct the tableaux? Here's an example. Let's try to show that the formula $\neg((p \vee q) \wedge (\neg p \wedge \neg q))$ is a theorem. As explained above, we start with only a single node containing the negation of this initial formula. Instead of using the $\neg$ sign here, we indicate the top level negation that we've added to our input formula by *signing* the formula with an F:

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

Signed Formulae

All formulae in our tableaux calculus will be *signed* in that way as either T or F. The signs are best thought of as instructions that tell us that we have to make a formula true or false, respectively. Of course these signs were not covered by our original definitions in Chapter 1. Extending the syntax accordingly, it makes sense to adjust the definition of 'literal': Literals are signed atomic formulae. This means that (in contrast to our old definition) literals never have any connectives; we will no longer regard formulae with a negation symbol ($\neg$) as literals, even if the scope of the negation is atomic.

Next let's see how we can make our input formula *false* (following its sign). 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:

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

Conjunctive Expansion

We mark formulae that we have already expanded by $\checkmark$. 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* of a formula respectively a branch):

$$(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{\mathtt{F}}\checkmark$$
$$\downarrow$$
$$1 : ((p \vee q) \wedge (\neg p \wedge \neg q))^{\mathtt{T}}\checkmark$$
$$\downarrow$$
$$2 : (p \vee q)^{\mathtt{T}}$$
$$\downarrow$$
$$3 : (\neg p \wedge \neg q)^{\mathtt{T}}$$

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 certainly won't. So let us further expand these formulae!

7.2.2 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):

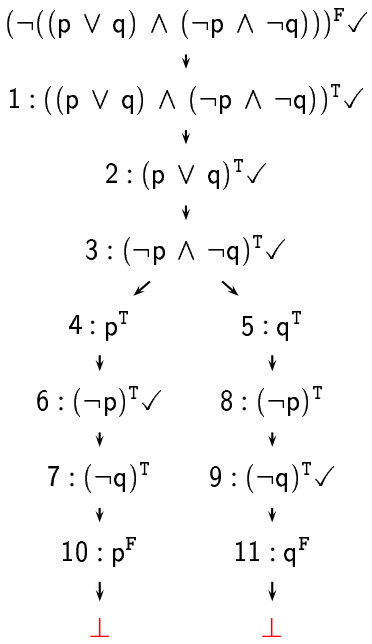
$$
\begin{array}{l}
(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark \\
\quad\downarrow \\
1:((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark \\
\quad\downarrow \\
\quad 2:(p \vee q)^{T}\checkmark \\
\quad\downarrow \\
\quad 3:(\neg p \wedge \neg q)^{T} \\
\quad\swarrow\quad\searrow \\
4:p^{T} \qquad 5:q^{T}
\end{array}
$$

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 again.

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.

$$
\begin{array}{l}
(\neg((p \vee q) \wedge (\neg p \wedge \neg q)))^{F}\checkmark \\
\quad\downarrow \\
1:((p \vee q) \wedge (\neg p \wedge \neg q))^{T}\checkmark \\
\quad\downarrow \\
\quad 2:(p \vee q)^{T}\checkmark \\
\quad\downarrow \\
\quad 3:(\neg p \wedge \neg q)^{T}\checkmark \\
\quad\swarrow\quad\searrow \\
4:p^{T} \qquad 5:q^{T} \\
\quad\downarrow \qquad\quad\downarrow \\
6:(\neg p)^{T} \qquad 8:(\neg p)^{T} \\
\quad\downarrow \qquad\quad\downarrow \\
7:(\neg q)^{T} \qquad 9:(\neg q)^{T}
\end{array}
$$

This time there are four expansions 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 could in principle choose any of them. But remember that we want to have a closed tableaux, i.e. we want each branch to be closed. So it's a good idea to expand the 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. These two expansions give us a closed tableaux (we indicate closed branches by a $\bot$)!



Signed Branch Closure

We call a branch closed iff there are two occurrences of the same atom with different signs on this branch (like a^F and a^T). So, we are done: We have generated a tableaux proof for the theorem $\neg((p \vee q) \wedge (\neg p \wedge \neg q))$!

Exercise 7.1 *Add annotations to the nodes of the tableaux above. The annotations should indicate from where each node resulted, and why. For example, node 1 would be annotated "from Input", and node 2 something like "from 1, making a conjunction true". Additionally, the closure nodes ($\bot$) should be annotated with the numbers of the respective contradictory nodes.*

7.2.3 Analytical Tableaux: A more formal Account

Let us now sum up what we've done in the example with the help of a few more concise definitions.

Signed Formulae

The calculus of analytical tableaux analyzes a formula in a tree that represents a set of case distinctions for satisfiability. The calculus we will use acts on *signed* formulae, i.e. formulae decorated with an intended truth value. A formula $\mathbf{A}^T$ signifies that the calculus tries to satisfy the formula $\mathbf{A}$, whereas $\mathbf{A}^F$ shows that the calculus tries to refute it.

Initial Tableaux

The tableaux proving process starts with an *initial tableaux* that contains only one node with one signed formula.

Tableaux Inference Rules

The tableaux is then generated by applying the following inference rules

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

For the moment we have only given the rules for conjunction and disjunctions, since the other connectives can be defined in terms of these ($\mathbf{A}\vee\mathbf{B}\equiv\neg(\neg\mathbf{A}\wedge\neg\mathbf{B})$, $\mathbf{A}\Rightarrow\mathbf{B}\equiv\neg\mathbf{A}\vee\mathbf{B}\equiv\neg(\mathbf{A}\wedge\neg\mathbf{B})$). We will extend the calculus later (page 130).

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 branch closing rule that adds the special symbol $\bot$ (for "closed") to branches that contain contradictory literals.

Open, Closed, Saturated

We will call a branch in a tableaux *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 no rule can be applied to it, sticking to this convention.

Termination

The convention helps us ensure that the tableaux construction process always terminates (at least for the quasi propositional logic PLNQ that we will introduce in a minute (page 126)). The inference rules just given 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 and the formulae on all branches have been reduced to literals (in other words, when 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.

A stronger version of our convention allows rule application only if it adds new material (i.e. if the result does not already occur on the branch). In Section 10.2.6, we will need this stronger control mechanism when we deal with first-order tableaux expansion.

Tableaux Proof

As we have discussed already, 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 for $\mathbf{A}^{\alpha}$ means building up an exhaustive case analysis of what is necessary to give $\mathbf{A}$ the truth value α. If all branches are closed, this means that all cases in this analysis lead to contradictions, and so $\mathbf{A}$ cannot have the truth value α.

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 $\mathbf{A}$. It

refutes the possibility of finding a model where **A** evaluates to F. Thus **A** must evaluate to T in all models, which is just our definition of validity.

Positive vs. Negative Calculi

So the tableaux procedure gives us a notion of proof and can thus be used as a calculus for proving theorems. But as we have seen it does not prove a theorem directly by deriving it from a set of axioms (like the calculus in Section 7.1.3 does). Instead it proves it by refuting its negation. A proof that works indirectly like this is also called a *refutation proof* . A calculus that leads to a refutation proof is called *negative* or *test calculus* . Generally negative calculi have computational advantages over positive ones, since they have a built-in sense of direction.

7.2.4 Using Tableaux to test Truth Conditions

Let us turn to some examples to back up the theoretical considerations in the last section (page 124).

The Logic PLNQ

To make things more interesting, we will use our reasoning procedure with a fragment (which we call PLNQ) of first-order predicate logic that allows us to express simple natural language sentences, without introducing the whole complications of first-order inference. PLNQ ("Predicate Logic with No Quantifiers") is a fragment of first-order logic (page 9) without variables and quantifiers[3]

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

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

which is equivalent to

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

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

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

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

[3] This logic is equivalent to propositional logic in expressivity: atomic formulae (page 9) take the role of propositional variables.

?- 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. $(\text{LOVE}(\text{MARY},\text{BILL}) \vee \text{LOVE}(\text{JOHN},\text{MARY})) \Rightarrow \text{LOVE}(\text{JOHN},\text{MARY})$
3. $\neg(\neg(\neg\text{LOVE}(\text{MARY},\text{BILL}) \wedge \neg\text{LOVE}(\text{JOHN},\text{MARY})) \wedge \neg\text{LOVE}(\text{JOHN},\text{MARY}))$

To prove the entailment in (1 (page 127)) we represent it as an implication (2 (page 127)). Recall that the deduction theorem (page 16) allows us to do so. We then eliminate the implication, arriving at (3 (page 127)).

Intuitively, (1 (page 127)) 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:

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

By now we've discussed thoroughly that a tableaux proof for a theorem is a *closed* tableaux for its negation. Yet obviously, 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.

In Section 9.1.3 we will look closer at open tableaux. We will study their relation to models, and basically see that if a tableaux has any branches that are open *and* saturated, we can read off a model for its input formula from each of them. Making the following observation now will help us understand why: 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). In other words: They state the 'minimal requirements' on a model for the (negated) input formula.

7.2.5 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. 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 reading 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. Intentions and assumptions 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.

7.2.6 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 $\neg(\psi_1 \wedge \ldots \wedge \psi_n \wedge \varphi)^{\mathbf{F}}$ 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'):

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

This is equivalent to:

$$\big(\neg\big(\big(\neg\big(\text{SIAMESECAT(MUTZ)} \wedge \neg\text{LIKE(MARY,MUTZ)}\big)\big) \wedge \text{SIAMESECAT(MUTZ)} \wedge \neg\text{LIKE(MARY,MUTZ)}\big)\big)^{\mathbf{F}}$$

As an exercise (page 132), convince yourself that this formula really yields a closed tableaux.

7.2.7 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:

$$\neg\big((\psi_1 \wedge \ldots \wedge \psi_n) \wedge \neg\varphi\big)^{\mathbf{F}}$$

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.

As an exercise (page 132), we ask you to use PLNQ to check whether 'Mary has a husband.' is informative in the context 'If Mary is married then she has a husband. She is married.' (treating the pronouns as in our example above).

?- Discussion!

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

Let us emphasize again 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?

7.2.8 Sidetrack: Practical Enhancements for Tableaux

Now that we've seen a first application, let's make a practical improvement to our tableaux calculus. So far, we have only given tableaux expansion rules for the connectives $\wedge$ and $\neg$. While it is possible to get by with rules for only these two connectives in PLNQ, it is a bit unnatural and tedious: we always have to eliminate the other connectives. In this section, we will make the calculus less frugal by adding rules for them.

Derived Inference Rules

We add such new rules to our calculus as *derived rule* s, i.e. inference rules that only abbreviate deductions in the original calculus. Generally, adding derived inference rules does not change the logical behaviour of a calculus and is therefore a safe thing to do. In particular, we will add the following three rules for the connective $\Rightarrow$to our tableaux system.

$$\frac{\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{F}}\mid\mathbf{B}^{\mathsf{T}}}\qquad\frac{\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\\mathbf{B}^{\mathsf{F}}\end{array}}\qquad\frac{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}\end{array}}{\mathbf{B}^{\mathsf{T}}}$$

Chaining

We will now convince ourselves that theses rules are derived rules. Take for instance the third rule. It is so useful that we give it a name: We call it the *chaining rule* . The green formulae on the tableaux are the ones we take over into our derived inference rule.

$$\frac{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\\mathbf{A}\Rightarrow\mathbf{B}^{\mathsf{T}}\end{array}}{\mathbf{B}^{\mathsf{T}}}\qquad\text{abbreviates}\qquad\begin{array}{c}\mathbf{A}^{\mathsf{T}}\\(\mathbf{A}\Rightarrow\mathbf{B})^{\mathsf{T}}\\(\neg(\mathbf{A}\wedge\neg\mathbf{B}))^{\mathsf{T}}\\(\mathbf{A}\wedge\neg\mathbf{B})^{\mathsf{F}}\\\mathbf{A}^{\mathsf{F}}\mid\neg\mathbf{B}^{\mathsf{F}}\\\bot\mid\mathbf{B}^{\mathsf{T}}\end{array}$$

In a similar way we derive rules for other common connectives $\vee$ and $\Leftrightarrow$. By the way, it is a worthwhile exercise to spell out the abbreviated tableaux for some of the derived rules.

$$\frac{\mathbf{A}\vee\mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{T}}\mid\mathbf{B}^{\mathsf{T}}}\qquad\frac{\mathbf{A}\vee\mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{F}}\\\mathbf{B}^{\mathsf{F}}\end{array}}\qquad\frac{\mathbf{A}\Leftrightarrow\mathbf{B}^{\mathsf{T}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\mid\mathbf{A}^{\mathsf{F}}\\\mathbf{B}^{\mathsf{T}}\mid\mathbf{B}^{\mathsf{F}}\end{array}}\qquad\frac{\mathbf{A}\Leftrightarrow\mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}}\mid\mathbf{A}^{\mathsf{F}}\\\mathbf{B}^{\mathsf{F}}\mid\mathbf{B}^{\mathsf{T}}\end{array}}$$

With these rules, the tableaux we have seen before (page **??**) has the following simpler form:

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

7.3 Tableaux Web-Interface

Click here![4]

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

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[5]. 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. Later (page 164) in the context of first order tableaux, we will learn what this argument is good for.

Try this!

Take for example the tableaux we have seen in Section 7.2.4 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.

7.4 Exercises

In !!!UNEXPECTED PTR TO EX_EX.SEC_INFERENCE.TABLEAUX1!!!, you are asked to add annotations to a given tableaux.

Exercise 7.2 *Re-prove the valid formula* $\mathbf{A} \Rightarrow \mathbf{A}$ *whose proof in the Hilbert calculus (page 116) we have studied in Section 7.1.4 in our tableaux calculus. Think about the difference between positive and negative calculi (page 126).*

Exercise 7.3 *[You have to read the sidetrack on derived inference rules (page 130) to do this exercise.] To see the value of the derived inference rules, prove or refute the formula* $(p \iff q) \iff (q \iff p)$.

Exercise 7.4 *In Section 7.2.6 we claim that the utterance 'Mary doesn't like Mutz' is inconsistent with the discourse 'If Mutz is a Siamese cat, then Mary likes her. Mutz is a Siamese cat.' because we can construct a closed tableaux for the formula:*

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

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

resp. the equivalent:

$$\big(\neg\big(\neg\big(\text{SIAMESECAT}(\text{MUTZ}) \wedge \neg\text{LIKE}(\text{MARY,MUTZ})\big)\big) \wedge \text{SIAMESECAT}(\text{MUTZ}) \wedge \neg\text{LIKE}(\text{MARY,MUTZ})\big)^{\mathbf{F}}$$

Construct this tableaux.

Then construct the tableaux needed to check whether 'Mary has a husband.' is informative in the context 'If Mary is married then she has a husband. She is married.' (treating the pronouns as in the example above).

For both examples, you may either use the derived inference rules from Section 7.2.8, or start of with a formula that uses only $\wedge$ and $\neg$.

8

Tableaux Implemented

8.1 Implementing PLNQ

8.1.1 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 Chapter 9 we will learn how to use tableaux for a task called *model generation*. 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 recusrive 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 (T/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?

8.1.2 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 125).

```
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 8.1.5 we illustrate all this by an example.

8.1.3 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 125). 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.

8.1.4 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 $\mathcal{T}(\wedge)^{\mathbf{F}}$ (page 125) 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 $\mathcal{T}(\wedge)^{\mathbf{F}}$-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 $\mathcal{T}(\wedge)^{\mathbf{F}}$-rule. Hence the whole subtableaux for the negative conjunction is closed.

8.1.5 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 $\neg\big(\big(\text{RUN}(\text{JOHN}) \wedge \neg\text{SLEEP}(\text{MARY})\big) \wedge \big(\text{SLEEP}(\text{MARY})\big)\big)$, and suppose we want to make it false. This should result in a closed one-branch tableaux, containing the contradiction $\text{SLEEP}(\text{MARY})^{\mathbf{T}}$ vs. $\text{SLEEP}(\text{MARY})^{\mathbf{F}}$. Now let's see how our program finds this tableaux.

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.

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.

What is the result?

Now we know that our input formula (page 140) 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.

8.1.8 Two Connectives

Notice that we've only discussed (and in fact that we've only implemented) tableaux rules for the connectives $\neg$ and $\wedge$ 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 $\neg$ and $\wedge$. 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.

8.2 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: `modGen/3`
See file `comsemOperators.pl`.	Our usual operator definitions

8.3 Exercises

Exercise 8.1 *Implement the derived inference rules from Section 7.2.8 using the ideas in this chapter. Include in your solution the examples that you used to test your implementation.*

Exercise 8.2 *Draw a tableaux for, and run the implemenation with the input:*

$$(\neg(\neg(\text{WALKS(JOHN)} \wedge \text{WALKS(MARY)}) \wedge \neg(\text{WALKS(MARY)} \wedge \text{RUN(MARY)}))) \wedge (\neg\text{WALKS(JOHN)} \wedge \neg\text{RUN}$$

If you expand the first conjunct first (as our implementation does), the second conjunct will have to be expanded on two branches.

Make sure you understand how this is done in our implementation. You can either perform a `trace` *or draw a computation tree.*

Exercise 8.3 *In Section 7.2.1 we used little check marks (✓) to signify that a formula had been expanded. In Section 7.2.3, we said that we do not expand formulae twice in order to ensured termination of the tableaux construction process. Now there's no obvious equivalent to such a technique in our implemenation. It terminates although it does not keep any explicit record of which formulae have already been expanded. Why doesn't it keep on expanding the same formulae for ever? In other words: How does our program know when a tableaux is saturated?*

9

[Sidetrack] Model Generation

9.1 Using Model Generation for Natural Language Interpretation

9.1.1 Why Model Generation?

In the following we will use a model generation procedure to model discourse understanding. This approach takes us one step beyond what we've done so far. Take for example the sentence 'John loves Mary and Mary hates John'. Up to now we would have said that understanding this sentence (at least for a computer) consisted in constructing the formula LOVE(JOHN,MARY) $\wedge$ HATE(MARY,JOHN). From now on, we say that understanding the sentence also involves deriving the literals $\{$LOVE(JOHN,MARY),HATE(MARY,JOHN)$\}$.

Why should we be so keen on deriving $\{$LOVE(JOHN,MARY),HATE(MARY,JOHN)$\}$ although we've already got their conjunction LOVE(JOHN,MARY) $\wedge$ HATE(MARY,JOHN). What's so special about a set of implied literals compared to a complex formula that implies them? The answer is that they represent the truth conditional content of the sentence in its basic form. This makes them interesting in many respects:

Formally, a set of literals derived from a formula specifies a model for that formula (and that's why we call the process of deriving it *model generation*). We discussed the relation between sets of literals and models in Section 2.1.4. In our example, the literals $\{$LOVE(JOHN,MARY),HATE(MARY,JOHN)$\}$ specify a model with a domain consisting of JOHN and MARY, where John loves Mary and Mary hates John. We shall even sloppily say that the literals *are* this model.

Technically, a flat list of literals is a nicer and more easily accessible data structure than a formula with recursive structure. We can view model generation as a normalization-like postprocessing step after semantics construction. Additionally we already have a tool that works on input of that form: Our model checker from Chapter 2. We could give it the model `[love(john,mary), hate(mary,john)]` and have it decide whether `forall(X, love(X,mary))` is true in this model or not.

Conceptually, the literals derived by a model generation procedure are more fundamental than the complex formula they are derived from. Ideally they stand for basic, logically independent facts that characterize a real-world situation where the complex formula would be true. By logically independent, we mean that the truth of each such basic fact depends only on the way the real world is, not on the truth of any other facts.

Complex sentences are in general *not* logically independent of one another. For example, the truth of the complex sentence 'John loves Mary and Mary hates John' interferes with that of 'John loves Mary or Mary hates John'. In contrast the basic fact 'John loves Mary' may or may not hold in a situation, independently from the equally basic 'Mary hates John' (and independently from all other facts): There can be two situations that only differ in whether John loves Mary or not.

Of course it is a strong idealization to assume that all literals that we derive from the semantic representations we've discussed are really logically independent. For instance the two literals LOVE(JOHN, MARY) and HATE(JOHN, MARY) obviously should not be said to be independent - either John loves Mary or he hates her, but normally not both at the same time. The problem here is that our semantic treatment of single words is by far not fine-grained enough. We basically translate each verb or noun into one single predicate symbol of its own. But there exist various meaning relations between single words. The area of *lexical semantic* s takes such relations as one starting point for working out elaborate models of the internal structure of word meanings.

Cognitively, a set of literals can serve as an approximation of what is called a *mental model* in the psychological literature. It is assumed that mental models are what constitutes believes and knowledge about the real world in human minds. Communication by natural language is then viewed as a process of transporting parts of the mental model of the speaker into the the mental model of the hearer. Therefore, the interpretation process on the part of the hearer is a process of integrating the meaning of the utterances of the speaker into his mental model.

We can take (sets of) literals as the currency for the information transferred in this process. What makes this choice of currency particularily interesting is that sets of literals are well-defined and have enough internal structure to allow us to formulate empirically testable hypotheses. For example they give us a means of claiming that we transfer more information by uttering 'John loves Mary and Mary hates John' than by just saying 'Mary hates John'. In the first case the hearer potentially has to integrate two literals (i.e. two 'pieces of information') into her mental model, compared to only one in the latter case. We could thus e.g. predict that understanding the first sentence would consume more resources than understanding the second one.

9.1.2 Tableaux for Model Generation with PLNQ

So for the above reasons we would like to have a technique that derives all literals that a formula implies, that is, we would like to have a *model generation procedure* . And in fact, we already have one. Look again at the tableaux (page **??**) by means of which we have attempted to prove the entailment 'Mary loves Bill or John loves Mary' $\models$ 'John loves Mary'.

The tableaux is saturated and still open. This tells us that our proof failed, which is correct. But the tableaux contains further useful information: The (green marked) literals on the open branch give us a counter-model for the entailment, namely $\{\text{LOVE}(\text{MARY}, \text{BILL})^{\text{T}}, \text{LOVE}(\text{JOHN},$
So, as a by-product of the proof, the tableaux has generated a model for its input formula (the negation of the entailment to be proven). We will now use this property of our tableaux calculus directly for model generation. Note that models from now on are signed. The model from above may equivalently be written $\{\text{LOVE}(\text{MARY}, \text{BILL}), \neg\text{LOVE}(\text{JOHN}, \text{MARY})\}$.

To find models for a formula, we will simply build a tableaux for this formula with positive polarity (T). In the end, each (open) branch of that tableaux will tell us one way of making the formula true - in other words it will contain a model for it.

Let us look at an example, the (admittedly a little unnatural) sentence 'John doesn't love Mary and if John doesn't love Mary then Mary loves Bill', which we represent by the formula:

$$\neg\text{LOVE(JOHN,MARY)} \wedge \big(\neg\text{LOVE(JOHN,MARY)} \Rightarrow \text{LOVE(MARY,BILL)}\big)$$

which is equivalent to

$$\neg\text{LOVE(JOHN,MARY)} \wedge \big(\neg\big(\neg\text{LOVE(JOHN,MARY)} \wedge \neg\text{LOVE(MARY,BILL)}\big)\big).$$

Now, which models does this formula have? Before we run our tableaux algorithm, let us give an intuitive formulation of how the models for our formula should look. The first part of our input sentence tells us that John does not love Mary. So, this fact must definitely be reflected in every model. The second part alone (the implication'if...then...') is true in two kinds of model: Either in a model where John loves Mary or in one where she loves Bill. Now since we already know that John doesn't love Mary, we know that only the second alternative is possible. So there is only one model for our sentence, namely the one where John doesn't love Mary but Mary loves Bill.

Here's the tableaux:

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

This tableaux is saturated and has one open branch, representing a model for the input formula. As you can see the tableaux once had *two* open branches while it was being constructed. At that time these two branches represented two different *models under construction* for the input formula. But our calculus found out that the literals on the left branch are contradictory. Hence this branch was discarded as it does not represent a model. So now there remains one open branch in our tableaux, and its literals taken together represent exactly the model we described above:

$$\big\{\text{LOVE(JOHN,MARY)}^{\mathtt{F}}, \text{LOVE(MARY,BILL)}^{\mathtt{T}}\big\}$$

Our tableaux calculus made use of the information given in the first part of the sentence while processing the rest. It then automatically made the right decision in discarding the left branch. So it arrived at the same result as we did in our intuitive considerations.

?- Question!

Look at the tableaux again. Which formula expansion resulted in the branching? In terms of models, why does this expansion have to introduce a branching? And why is the closure of the left branch appropriate?

9.1.3 Tableaux Branches and Herbrand Models

In this section we explore the relation between literals on saturated branches and models in more detail. Put in more professional terms, there's one general property of tableaux calculi of which we take advantage in model generation: The set of literals on an open saturated tableaux branch corresponds to a Herbrand model (page 22) for the input formula of the tableaux.

A Second Look at our Example

To fortify our intuition, we will study our example above, where the set of literals on the open branch is

$$\{\text{LOVE}(\text{JOHN},\text{MARY})^{\mathtt{F}},\text{LOVE}(\text{MARY},\text{BILL})^{\mathtt{T}}\}$$

and build such a model.

Let us recap: A Herbrand model $\mathcal{H}$ is a model (a pair $(\mathcal{D},I)$ of a domain and an interpretation function) where individual constants are interpreted "as themselves". Our formula mentions "Mary", "John", and "Bill", so let us take $\mathcal{D}:=\{\text{MARY},\text{JOHN},\text{BILL}\}$ and fix $I(\text{MARY})=\text{MARY}$, $I(\text{BILL})=\text{BILL}$, $I(\text{JOHN})=\text{JOHN}$.

So the remaining task is to find a suitable interpretation for the predicate LOVE that makes $\text{LOVE}(\text{JOHN},\text{MARY})$ false and $\text{LOVE}(\text{MARY},\text{BILL})$ true. This is simple: we just take $I(\text{LOVE})=\{\langle\text{MARY},\text{BILL}\rangle\}$. Indeed we then have $[\![\text{LOVE}(\text{MARY},\text{BILL})\vee\text{LOVE}(\text{JOHN},\text{MARY})]\!]^{\mathcal{M}}=\mathtt{T}$ and $[\![\text{LOVE}(\text{JOHN},\text{MARY})]\!]^{\mathcal{M}}=\mathtt{F}$ according to the definition of the interpretation function (page 6).

More Formally

Now let us generalize what we have just done for our example to arbitrary sets of literals coming from a saturated open tableaux branch. Let $\mathcal{LIT}$ be such a set of literals. Note that $\mathcal{LIT}$ necessarily is *contradiction-free* : It cannot contain literals $\mathbf{A}^{\mathtt{T}}$ and $\mathbf{A}^{\mathtt{F}}$ at the same time; otherwise the branch would be closed.

Fixing the Domain

We take the domain of interpretation to be isomorphic to the set $\mathcal{C}(\mathcal{LIT})$ of individual constants occurring in $\mathcal{LIT}$. The simplest thing to achieve this isomorphism, is just to take $\mathcal{D}:=\mathcal{C}(\mathcal{LIT})$ and take I to be the identity function on individual constants $I=\mathbf{Id}_{\mathcal{C}}:\mathcal{C}\to\mathcal{D}$. This means that all constant symbols will simply denote themselves. Note that choosing $\mathcal{D}$ this way is the most general possibility as long as we assume all individuals to be distinct. If we wanted to be able to express equality between induividuals (say between Mary and Bill, in which case we would know that he/she loves him/herself), we would have to add more structure to our domain, shifting to normal models (page 16).

Herbrand models

Now we can give a formal definition of Herbrand models. We will call a model a *Herbrand model* , iff $\mathcal{D}$ is a set $\mathcal{C}$ of individual constants and I is the identity on $\mathcal{C}$.

Interpreting Predicates

The only thing still to be done now is fixing the interpretation of predicates so that they satisfy the literals in $\mathcal{LIT}$. Again, we make the choice that commits us least. So we choose the extension of our predicates that makes exactly the positive literals in $\mathcal{LIT}$ true. That is we take

$$I(p) = \{(c_1, \ldots, c_n) | p(c_1, \ldots, c_n)^{\mathtt{T}} \in \mathcal{LIT}\}$$

for every n-ary predicate p that occurs in $\mathcal{LIT}$. Note that $I(p)$ is well-defined, since $\mathcal{LIT}$ does not contain contradictions. In result, each predicate will be interpreted as the set of those tuples of constant symbols that appear as arguments in positive literals for the respective predicate in $\mathcal{LIT}$.

What have we achieved?

Now we have fully specified a model $\mathcal{M} = (\mathcal{D}, I)$, such that $[\![\mathbf{A}]\!]^{\mathcal{M}} = \alpha$ for each $\mathcal{A}^{\alpha} \in \mathcal{LIT}$. In other words, we have found a model that satisfies all literals in $\mathcal{LIT}$.

On the other hand, for any model $\mathcal{M}$, the set

$$\mathcal{LIT}(\mathcal{M}, \Sigma) := \{\mathbf{A}^{\alpha} | \mathbf{A} \text{ atomic}, \ \alpha = [\![\mathbf{A}]\!]^{\mathcal{M}}\}$$

of literals is contradiction-free, since $[\![\cdot]\!]^{\mathcal{M}}$ is a function. So contradiction-free sets of literals are representations of models. Note that this is why we sloppily say that some set of literals *is* a (Herbrand) model. If this sounds familiar, that's no wonder. We have done the same argumentation when we had to represent models (page 21) in Prolog.

9.1.4 Tableaux generate Herbrand Models

The literals on an open saturated tableaux branch $\mathcal{B}$ are a Herbrand model $\mathcal{H}$, as we have convinced ourselves above. Is this all we wanted to show? No! We still have to argue that $\mathcal{H}$ is also a model for the input formula of the tableaux.

From the Literals to the Input Formula

The argument goes as follows: The tableaux inference rules are made up in a way that whenever a model satisfies the succedent, it also satisfies the antecedent (which is immediate from the semantics of the principal connectives). So, if $\mathcal{H}$ is a model of the literals of $\mathcal{B}$, then it is also a model of the complex formulae on $\mathcal{B}$ which were decomposed into these literals during the tableaux construction. Transitively, this argument shows that $\mathcal{H}$ is a model of *all* formulae on $\mathcal{B}$.

In particular, $\mathcal{H}$ is a model for the input formula of the tableaux, which is on $\mathcal{B}$ by construction. So the tableaux procedure is also a procedure that generates explicit (Herbrand) models for the input formula of the tableaux. Every branch of the tableaux corresponds to a (possibly) different Herbrand model.

Finite Model Property

As we have already seen (page 125), tableaux saturation always terminates for PLNQ. As a consequence, every Herbrand model we generate is represented by a finite set of literals. But if we look at the construction of such a Herbrand model, we see that it is also finite in a different sense: its domain is finite. Because all of the Herbrand models generated for PLNQ formulae by our tableaux procedure are finite, and our tableaux procedure is complete, we can be sure that satisfiable PLNQ formulae always have a finite model. This property of a logic is called the *finite model property* .

Decidability

Another consequence of termination (page 125) of tableaux saturation together with the fact that tableaux enumerate *all* possible models is that model generation is a *decision procedure* for satisfiability (page 12) in PLNQ, i.e. an algorithm that will determine the satisfiability of any PLNQ formula in a finite time.

9.2 Discourse understanding

9.2.1 Building Discourse Models

World Knowledge

One big advantage of tableaux is that they come with an in-built possibility to keep around knowledge that's assumed to be 'already there'. Be it world knowledge, the content of previous sentences, or a new sentence - simply put it on the tableaux, and you can use it. Putting this formally, let's generalize the notion of an initial tableau (page 124). Instead of allowing only one initial signed formula at the root node, let's allow a linear tree whose nodes are labeled with signed formulae. We will for instance start model generation for the sentence 'If Mutz is a cat, Mutz likes Hansi.' with the following tableaux, already containing the world knowledge that Hansi is a canary and Mutz is a cat (we've enclosed the formula for the actual input *sentence* in a box):

$$\begin{array}{c} \textsc{canary}(\textsc{Hansi})^{\mathtt{T}} \\ \textsc{cat}(\textsc{Mutz})^{\mathtt{T}} \\ \boxed{\textsc{cat}(\textsc{Mutz}) \Rightarrow \textsc{like}(\textsc{Mutz}, \textsc{Hansi})^{\mathtt{T}}} \end{array}$$

Online Process

Now what exactly are we going to do with tableaux to model discourse understanding? We think of the hearer (or reader) in a discourse as internally maintaining a tableaux that represents the current set of alternative models for that discourse. Now the hearer's understanding is an online process that receives as input the logical forms of the sentences of the discourse one by one. The hearer has a mechanism for choosing a *preferred model* (i.e. branch) from the open branches of his internal tableaux. He also maintains a set of *deferred* branches that can be re-visited if the preferred branch gets closed in the course of further processing.

Append, Saturate, Choose

Upon input, the given logical form is appended to the tableaux as a leaf to all branches. The hearer then saturates the current (i.e. the preferred) tableaux branch, exploring the set of possible models for the sequence of input sentences. If the subtableaux generated by this saturation process contains open branches, then he chooses one of them as the new preferred model, marks some of the other open branches as deferred, and waits for further input. If the saturation yields a closed sub-tableaux, he backtracks. This means that he selects a new preferred branch from the deferred ones, and continues the tableaux expansion on this branch. Backtracking may be repeated until successful, or until some termination criterion is met. In this case discourse processing fails altogether.

9.2.2 A first Example

Let us now look at an example of tableaux-based discourse processing. Assume that we hear the sentence 'If Mary smokes, Mary looks pale and feels sick.' Let us further assume that we process this sentence without any further knowledge whatsoever. This is cognitively implausible, but it facilitates the presentation: We simply start with an empty tableaux. Understanding this sentence gives us the following tableaux:

$$\boxed{\text{SMOKE}(\text{MARY}) \Rightarrow (\text{PALE}(\text{MARY}) \wedge \text{SICK}(\text{MARY})^{\text{T}})}$$

$\text{SMOKE}(\text{MARY})^{\text{F}}$	$\text{PALE}(\text{MARY}) \wedge \text{SICK}(\text{MARY})^{\text{T}}$
	$\text{PALE}(\text{MARY})^{\text{T}}$
	$\text{SICK}(\text{MARY})^{\text{T}}$

This tableaux has two branches, both saturated and open. So, there are two models that make this discourse true. In other words, there are two situations that are described by this sentence. Either Mary does not smoke, or Mary is pale and feels sick. As we argued above, we now have to decide which model we prefer.

?- Question!

It is an open question what kind of information influences such decisions. Do you have any ideas? (We encourage you to speculate!)

Let's take the second model (on the right branch) to be our preferred model (without any compelling reasons), and assume that the sentence 'Mary does not feel sick, Mary is hungry.' is uttered next. We proceed as follows: First, we add the corresponding formula to all branches of our tableaux:

$$\text{SMOKE}(\text{MARY}) \Rightarrow (\text{PALE}(\text{MARY}) \wedge \text{SICK}(\text{MARY})^{\text{T}})$$

$\text{SMOKE}(\text{MARY})^{\text{F}}$	$\text{PALE}(\text{MARY}) \wedge \text{SICK}(\text{MARY})^{\text{T}}$
$\boxed{\neg\text{SICK}(\text{MARY}) \wedge \text{HUNGRY}(\text{MARY})^{\text{T}}}$	$\text{PALE}(\text{MARY})^{\text{T}}$
	$\text{SICK}(\text{MARY})^{\text{T}}$
	$\boxed{\neg\text{SICK}(\text{MARY}) \wedge \text{HUNGRY}(\text{MARY})^{\text{T}}}$

We then start further expanding our preferred branch:

$$\begin{array}{l|l}
\text{SMOKE(MARY)} \Rightarrow (\text{PALE(MARY)} \wedge \text{SICK(MARY)}^{\text{T}}) & \\
\text{SMOKE(MARY)}^{\text{F}} & \text{PALE(MARY)} \wedge \text{SICK(MARY)}^{\text{T}} \\
\boxed{\neg\text{SICK(MARY)} \wedge \text{HUNGRY(MARY)}^{\text{T}}} & \text{PALE(MARY)}^{\text{T}} \\
 & \text{SICK(MARY)}^{\text{T}} \\
 & \boxed{\neg\text{SICK(MARY)} \wedge \text{HUNGRY(MARY)}^{\text{T}}} \\
 & \neg\text{SICK(MARY)}^{\text{T}} \\
 & \text{HUNGRY(MARY)}^{\text{T}} \\
 & \text{SICK(MARY)}^{\text{F}} \\
 & \mathcal{T}(\bot)
\end{array}$$

But what has happened? Our preferred branch has become inconsistent. So, we backtrack and expand the other branch (which is still open). As a result, we get the following tableaux:

$$\begin{array}{l|l}
\text{SMOKE(MARY)} \Rightarrow (\text{PALE(MARY)} \wedge \text{SICK(MARY)}^{\text{T}}) & \\
\text{SMOKE(MARY)}^{\text{F}} & \vdots \\
\boxed{\neg\text{SICK(MARY)} \wedge \text{HUNGRY(MARY)}^{\text{T}}} & \mathcal{T}(\bot) \\
\neg\text{SICK(MARY)}^{\text{T}} & \\
\text{HUNGRY(MARY)}^{\text{T}} & \\
\text{SICK(MARY)}^{\text{F}} &
\end{array}$$

The left branch (which is the new preferred branch) is saturated. So we've 'understood' our little discourse by building a model where Mary is hungry, and where she's neither smoking nor feeling sick. Note that the tableaux would close altogether if it would turn out later in the discourse that Mary is indeed smoking. In this case, there would be no model for the discourse at all, and we would know that at least one of the previous utterances must have been false.

9.2.3 A Second Example

The example before has shown that tableaux offer a data structure rich enough for maintaining different ways of understanding (parts of) a discourse. By organising tableaux expansion the way we do, in particular by concentrating on the expansion of one branch at a time, we can model the process of incrementally understanding a discourse. As we have seen in the example, this involves discarding one branch (i.e. partial model) and starting the model building process anew based on another branch if we encounter an inconsistency.

But using the ability to detect inconsistencies built into our model generation approach, we can do much more than we've seen so far. We can deal with lots of other kinds of ambiguities in the sentences we process. As our next case study, let's use inconsitency detection to resolve a syntactic ambiguity. The sentence in (Sentence 1 (page 152)) has two syntactic readings (2 (page 152)) and (3 (page 152))

1. Sentence 1:'Peter loves Mary and Mary sleeps or Peter snores.'
2. *Reading 1*: $(\text{LOVES}(\text{PETER}, \text{MARY}) \wedge \text{SLEEPS}(\text{MARY})) \vee \text{SNORES}(\text{PETER})$
3. *Reading 2*: $\text{LOVES}(\text{PETER}, \text{MARY}) \wedge (\text{SLEEPS}(\text{MARY}) \vee \text{SNORES}(\text{PETER}))$

Of course normally such an ambiguity would be resolved in the syntax- or semantic construction-component, e.g. on the base of prosodic information. But for the sake of

our example, let's assume our system isn't that clever. Thus at first, we will have to consider both of the above readings in parallel. We can do so by simply building two tableaux, one per reading.

Let us first look at Reading 2 (page 152).

$$\boxed{\text{LOVES}(\text{PETER},\text{MARY}) \wedge (\text{SLEEPS}(\text{MARY}) \vee \text{SNORES}(\text{PETER}))^{\text{T}}}$$
$$\text{LOVES}(\text{PETER},\text{MARY})^{\text{T}}$$
$$(\text{SLEEPS}(\text{MARY}) \vee \text{SNORES}(\text{PETER}))^{\text{T}}$$
$$\text{SLEEPS}(\text{MARY})^{\text{T}} \mid \text{SNORES}(\text{PETER})^{\text{T}}$$

We see that model generation gives us two models. In both, Peter loves Mary. But in the first one, Mary sleeps, while in the second one Peter snores. If we take the logically different input (Reading 1 (page 152)), we obtain different models:

$$\boxed{(\text{LOVES}(\text{PETER},\text{MARY}) \wedge \text{SLEEPS}(\text{MARY})) \vee \text{SNORES}(\text{PETER})^{\text{T}}}$$
$$\text{LOVES}(\text{PETER},\text{MARY}) \wedge \text{SLEEPS}(\text{MARY})^{\text{T}} \mid \text{SNORES}(\text{PETER})^{\text{T}}$$
$$\text{LOVES}(\text{PETER},\text{MARY})^{\text{T}}$$
$$\text{SLEEPS}(\text{MARY})^{\text{T}}$$

Let's continue the discourse with:

- Sentence 2:'Peter does not love Mary.'

We have to extend the second tableaux to:

$$(\text{LOVES}(\text{PETER},\text{MARY}) \wedge \text{SLEEPS}(\text{MARY})) \vee \text{SNORES}(\text{PETER})^{\text{T}}$$
$$\text{LOVES}(\text{PETER},\text{MARY}) \wedge \text{SLEEPS}(\text{MARY})^{\text{T}} \mid \text{SNORES}(\text{PETER})^{\text{T}}$$
$$\text{LOVES}(\text{PETER},\text{MARY})^{\text{T}} \mid \boxed{\neg\text{LOVES}(\text{PETER},\text{MARY})^{\text{T}}}$$
$$\text{SLEEPS}(\text{MARY})^{\text{T}} \mid \text{LOVES}(\text{PETER},\text{MARY})^{\text{F}}$$
$$\boxed{\neg\text{LOVES}(\text{PETER},\text{MARY})^{\text{T}}} \mid \bot$$
$$\text{LOVES}(\text{PETER},\text{MARY})^{\text{F}}$$
$$\bot$$

On this tableaux, we now have exactly one model. And the first tableaux closes altogether when extended with our second sentence.

?- Question

Check this claim!

This means that we've resolved all ambiguities. The choice of models has been reduced to one, which constitutes the intuitively correct reading of the discourse (Sentence 1 (page 152) and Sentence 2 (page 153)).

9.3 Wrapping it up (Model Generation)

World Knowledge

You can think of *world knowledge* as a collection of formulae stating facts about the world. Typically, there will be basic facts like SIAMESECAT(MUTZ) and rule-like facts like $\forall x.\text{SIAMESECAT}(\text{X}) \rightarrow \exists y.\text{OWN}(\text{Y},\text{X})$.

```
modGen(Formula,WKNumber,Model) :-
        wk(WKNumber,WK),
        toconj([Formula|WK],Conjunction),
        naonly(Conjunction,Converted),
        tabl(true(Converted),[],Model).
```

We have to give the world knowledge as a list of formulas. This list has to be put in the database together with a number, as a term `wk(N,Formulas)`. Our predicate `modGen/3` accesses the world knowledge by the number given as second argument, then conjoins it with the input formula (using `toconj/2`, which we won't discuss here). It then calls `tabl/3` to make the resulting conjunction true.

To give an example, we have added the toy world knowledge `wk(1,[man(john),love(john,mary)]).` You can try the following test call `modGen(love(john,mary)>woman(mary),1,M).`

A listing of all necessary files can be found in Section 8.2

9.4 Exercises

Exercise 9.1 *1. Construct a model generation tableaux to represent the following discourse: 'Fido is the dog and Mimi is the cat. Jane owns the dog or Jane owns the cat. Jane does not own Fido.'*

2. *Now show, using the model generation calculus, that adding the sentence 'Jane does not own Mimi' to the discourse above leads to a contradiction.*

Clarification*: We assume that 'the dog' is mapped to a constant d, where the information* $\mathrm{DOG}(d)$ *is in the world knowledge. Likewise for the cat.*

Exercise 9.2 *Look again at the example in Section 9.1.2. Instead of the somewhat clumsy sentence quoted there, let us now consider a small discourse similar in meaning: 'If John doesn't love Mary then Mary loves Bill. John doesn't love Mary.'. How would our tableaux approach process this discourse? Construct the tableaux.*

Compare your tableaux to the one you would build to validate the intuitively correct argument 'If John doesn't love Mary then Mary loves Bill. John doesn't love Mary. Thus Mary loves Bill.'

Exercise 9.3 **Project!**

[This is a project (mid- or end-term)]

Modify our model generation procedure (page 143) for PLNQ such that it can process discourse (i.e. multiple sentences) as sketched in Section 9.2.

You may (as in our implementation for single sentences) abstract over the question which is the preferred *model (branch). This means that you simply take the first model the implementation generates as the preferred one. In this case Prolog's search strategy, and in particular the order of the two disjunctive clauses of* `tab/3`*, effectively determine which model is prefered. The only thing* you *have to work out is how to store the alternative branches so that they can be backtracked to if subsequent expansions lead to branch closure on the preferred one.*

Alternatively, you can re-implement the tableaux procedure such that the tableaux is explicitely represented. There are various ways of doing this: you may e.g. represent a tableaux branch as a list of formulae (and hence a tableaux as a list of lists (branches)). This is what's done in the famous textbook by Melvin Fitting. Or you may assert tableaux nodes to the database and establish the connection between the nodes by specifying the children of each node. Any such explicit representation will give you more freedom in controling the tableaux expansion.

10

First-Order Inference

10.1 The Step to First Order

10.1.1 Why First-Order Inference?

As a logician, it's just a normal thing to ask if it's possible to extend what you've done for propositional logic to first-order logic. But apart from this there're especially good reasons to do so if you're interested in computational semantics: As a matter of fact, natural language contains lots of quantified NPs, which are formalized using quantifiers. Thus if we want to be able to infer from such sentences, we must have a treatment of quantifiers in our calculus.

Quantification in natural language...

In PLNQ we could only formulate (and therefore only infer from) particular statements about named entities. Up to now we could infer from the premisses

- 'Mary sleeps.' and 'If Mary sleeps then Mary snores.'

that Mary snores. But we could for instance neither express the generalization 'Every man snores.', nor could we infer from this and 'John is a man.' that John snores. In order to express the premiss of this argument, we have to let quantifiers into PLNQ, thus arriving at the language of full first-order logic. And in parallel our calculus has to be able to work with them in order to validate such arguments.

...and in world knowledge

But the omnipresence of quantification in natural language is not the only reason why we want to have first order inference. We are particularily interested in deriving the consequences that follow from our *world knowledge* together with some particular natural language sentence that's been uttered. Remember we argued above (page 150) that it is a great advantage of our tableaux based model generation method that we can readily infer from any kind of background knowledge by simply including it on the (initial) tableaux. But world knowledge characteristically consists of generalizations, such as 'Every cat is furry.' or 'Every human has a father.'. So we need quantified formulae to formalize world knowledge (such as $\forall X \text{CAT}(X) \Rightarrow \text{FURRY}(X)$ and $\forall X \text{HUMAN}(X) \Rightarrow \exists Y \text{FATHER}(Y,X)$). This means that inference becomes essentially first-order if we take world knowledge into account.

In fact both of the above arguments are also small lies. There's a lot of teritorry that lies between propositional and first-order logic, and it may well be that we don't really need full first-order logic and inference for the described tasks. There's a lot of research going on that tries to answer two questions:

- How much of first order logic do we need in order to formalize *exactly* those cases of quantification that really occur in natural language and world knowledge, no more and no less?
- How can we use clever formalizations and representations to support just the kinds of inferences we need, without bothering about the full complexities of first-order inference.

We will see in this chapter where we get using full first order logic. One thing we'll learn this way is to appreciate why it is so important to study what's in between propositional and first-order.

10.1.2 Extending our Calculus: The Universal Rule

In this section we extend our calculus to first-order logic with quantifiers. To do so, we have to add a treatment of quantification to the inference processes discussed so far. This means that we have to give rules for the quantifiers under both polarities.

New Rules: $\mathcal{T}(\forall)^{\mathtt{T}}$, and $\mathcal{T}(\forall)^{\mathtt{F}}$

Remember we confined ourselves to a minimal set of propositional connectives (only $\neg$ and $\wedge$), treating the others as defined. Likewise we shall confine ourselves here to giving rules for the universal quantifier $\forall$ and think of the existential $\exists$ as defined in terms of the equivalence $\exists XP \Leftrightarrow \neg\forall X\neg P$. Due to this equivalence, the rule for the universal quantifier with negative polarity ($\mathcal{T}(\forall)^{\mathtt{F}}$) is in essence a rule for existential quantification. We will sometimes call it 'the existential rule' (in contrast to $\mathcal{T}(\forall)^{\mathtt{T}}$, 'the universal rule'). We will later (page 172) give derived rules that deal with the existential quantifier directly.

The Universal Rule

Let's first have a look at the universal rule. Whenever we have a generalized sentence, we know that we can put it concretely about every single individual we know of. When we have $\forall x.A$, then we must also have A for any individual that we ever come to know of. Let's think of 'we have A for an individual named c' as 'we know A, with all occurences of the variable x in A replaced by the constant c'. Then, the following rule is a straightforward formalization of our considerations:

$$\frac{\begin{array}{c}\forall x.\mathbf{A}^{\mathtt{T}} \\ \vdots \\ [c/x]\mathbf{A}^{\mathtt{T}}\end{array} \quad c \in \mathcal{H}}{\mathcal{T}(\forall)^{\mathtt{T}}}$$

Multiple Applications

To understand this rule, we have to introduce the concept of a *Herbrand base* ($\mathcal{H}$) of a tableaux branch. The Herbrand base of a tableaux branch is the set of constant symbols occuring on that branch. The rule $\mathcal{T}(\forall)^{\mathtt{T}}$ allows to instantiate the scope of the quantifier with *any* constant symbol of the Herbrand base (the notation $[c/x]\mathbf{A}$ means that we substitute c for all occurences of x in $\mathbf{A}$). It may be necessary to apply the $\mathcal{T}(\forall)^{\mathtt{T}}$ rule to one and the same formula A with multiple or even all constants on a branch. To achieve a complete calculus we thus have (in contrast to all other rules) to allow for *multiple application* of this rule to one formula.

Up to now, we have only made sure that the rule is justified if we read it from the antecedent to the succedent. But remember (page 149) that it is crucial that the rules are also justified read in the other direction, i.e. the truth of the succedent guarantees the truth of the antecedent. But in fact, this is also the case here: when we have A for any individual that we ever come to know of, then we must also have $\forall x.A$. It can be shown that we can safely stick to cases where all individuals are named by some constant for tableaux-inference.

Let us now look at an example where one universal formula has to be instantiated twice to close a tableau. We will construct a tableaux proof for $\neg(\neg(\textsc{run}(\textsc{john}) \wedge \textsc{run}(\textsc{mary})) \wedge \forall X\,\textsc{run}(\mathrm{X}))$. Intuitively, we would expect this to be a theorem. It states that it's impossible that John and Mary don't run, but still everyone runs. This is clearly a truism. And in fact with our new rule, we can have a tableaux proof of this formula as follows (we've coloured the premiss and results of the $\mathcal{T}(\forall)^{\mathtt{T}}$-rule pink):

Using it

$$
\begin{array}{c}
\neg(\neg(\textsc{run}(\textsc{john}) \wedge \textsc{run}(\textsc{mary})) \wedge \forall x.\textsc{run}(\mathrm{X}))^{\mathtt{F}} \\
\neg(\textsc{run}(\textsc{john}) \wedge \textsc{run}(\textsc{mary})) \wedge \forall x.\textsc{run}(\mathrm{X})^{\mathtt{T}} \\
\neg(\textsc{run}(\textsc{john}) \wedge \textsc{run}(\textsc{mary}))^{\mathtt{T}} \\
\forall x.\textsc{run}(\mathrm{X})^{\mathtt{T}} \\
\textsc{run}(\textsc{john}) \wedge \textsc{run}(\textsc{mary})^{\mathtt{F}} \\
\begin{array}{c|c}
\textsc{run}(\textsc{john})^{\mathtt{F}} & \textsc{run}(\textsc{mary})^{\mathtt{F}} \\
\textsc{run}(\textsc{john})^{\mathtt{T}} & \textsc{run}(\textsc{john})^{\mathtt{T}} \\
\bot & \textsc{run}(\textsc{mary})^{\mathtt{T}} \\
 & \bot
\end{array}
\end{array}
$$

We achieve the closed tableaux by first instantiating the universal quantification with JOHN (closing the left branch) and then with MARY (closing the right branch). Intuitively, to show that neither John nor Mary runs, we have to derive this fact once for each of them. One instantiation alone would not suffice.

?- Question!

As we will learn, the possibility to re-apply the $\mathcal{T}(\forall)^{\mathtt{T}}$-rule represents the main computational problem of first order inference. Can you already see why it is so problematic?

10.1.3 The Existential Rule

What does an existential (respectively the equivalent negative universal) statement tell us? Intuitively, if we know that 'Somebody smokes' (respectively that 'It's not the case that everybody doesn't smoke'), then all we know is that there is *somebody* who

smokes, but we have no clue *who* it is. It may be someone we know of (say John or Bill), but it may just as well be someone we don't know yet. Therefore, the existential statement does not warrant the conclusions 'John smokes' or 'Bill smokes' - claiming any of these we would commit ourselves too much. John and Bill may both be non-smokers and 'Somebody smokes' still be true. But what we can do is give the unknown smoker a name that we invent only for him (we might call him 'N.N.'). So we can be sure that we don't say anything wrong about someone we already know of; we can let Bill and John remain non-smokers in a world where nevertheless *somebody* smokes. This idea is formalized in the following rule:

Existential Rule

$$\frac{\forall x.\mathbf{A}^{\mathtt{F}}}{[w_{new}/x]\mathbf{A}^{\mathtt{F}}}\,\mathcal{T}(\forall)^{\mathtt{F}}$$

Introducing a Witness

We take the scope of a negative universal quantification and substitute a brand new constant w_{new} for the quantified variable. The new constant is also called a *witness* or *skolem constant* . It is like the new name that we decided to invent for the smoker in our example. Because that witness constant is not contained in any of the formulae on the tableaux so far, the tableaux cannot close on a direct contradiction between any of these formulae and the coclusion of the $\mathcal{T}(\forall)^{\mathtt{F}}$-rule. Let's look at an example:

$$\begin{array}{r}\vdots\\ \textsc{smoke}(\textsc{John})^{\mathtt{F}}\\ \textsc{smoke}(\textsc{Bill})^{\mathtt{F}}\\ \forall x.(\neg\textsc{smoke}(x))^{\mathtt{F}}\\ \neg\textsc{smoke}(w_0)^{\mathtt{F}}\\ \textsc{smoke}(w_0)^{\mathtt{T}}\end{array}$$

In the second but last line, we have invented the witness w_0 for the scope of the existential quantification. We have generated a model in which John and Bill do *not* smoke, but nevertheless *somebody* (whom we call w_0) smokes.

The fact that our new constant cannot *immediately* be involved in closing the tableau does not imply that it cannot ever be involved in closing it. Contradictions may arise with consequences of universally quantified formulae, because once a witness constant has become available, it can be used with the $\mathcal{T}(\forall)^{\mathtt{T}}$-rule. Let us look at a tableaux where this happens. The formula $\forall x.\textsc{man}(\mathrm{x}) \vee \neg\textsc{man}(\mathrm{x})$ is obviously valid. It is equivalent to $\forall x.\neg(\textsc{man}(\mathrm{x}) \wedge \neg\textsc{man}(\mathrm{x}))$ ('it is not the case that there exists a man that isn't a man.'), for which we now give a tableaux proof:

$$\begin{array}{r}\forall x.\neg(\textsc{man}(x) \wedge \neg\textsc{man}(x))^{\mathtt{F}}\\ \neg(\textsc{man}(w_1) \wedge \neg\textsc{man}(w_1))^{\mathtt{F}}\\ \textsc{man}(w_1) \wedge \neg\textsc{man}(w_1)^{\mathtt{T}}\\ \textsc{man}(w_1)^{\mathtt{T}}\\ \neg\textsc{man}(w_1)^{\mathtt{T}}\\ \textsc{man}(w_1)^{\mathtt{F}}\\ \bot\end{array}$$

In the first step, we again invent a witness (w_1, 'the man that is no man') for the scope of the existential quantification. We then derive a contradiction for this witness,

about which we know nothing else and did not make any further assumptions. Yet of course we could have derived that contradiction with any other choice of constant instead of w_1, too. So the fact that we could derive it does not mean that our policy of inventing new constants has failed. On the contrary we have to be able to derive such contradictions involving universal quantification for our calculus to be complete.

10.1.4 Rule-like World Knowledge and Computational Nightmares

Something we could not deal with using the methods from Section 7.2 was knowledge like 'Every man has a father'. In full first-order logic, such knowledge is very easy to write down:

$$\forall x.\mathrm{MAN}(x) \Rightarrow \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,x)$$

But in a calculus such formulas are as dangerous as they're useful. To understand why, let's look at a tableaux constructed from our father-son-rule and the fact $\mathrm{MAN}(\mathrm{JOHN})^{\mathrm{T}}$.

$$\begin{array}{l|l}
\forall x.\mathrm{MAN}(x) \Rightarrow \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,x)^{\mathrm{T}} & \\
\mathrm{MAN}(\mathrm{JOHN})^{\mathrm{T}} & \\
\mathrm{MAN}(\mathrm{JOHN}) \Rightarrow \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,\mathrm{JOHN})^{\mathrm{T}} & \\
\mathrm{MAN}(\mathrm{JOHN})^{\mathrm{F}} & \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,\mathrm{JOHN})^{\mathrm{T}} \\
\bot & \mathrm{MAN}(c_0) \wedge \mathrm{FATHER}(c_0, \mathit{john})^{\mathrm{T}} \\
 & \mathrm{MAN}(c_0)^{\mathrm{T}} \\
 & \mathrm{FATHER}(c_0,\mathrm{JOHN})^{\mathrm{T}} \\
 & \mathrm{MAN}(c_0) \Rightarrow \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,c_0)^{\mathrm{T}} \\
 & \mathrm{MAN}(c_0)^{\mathrm{F}} \quad \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,c_0)^{\mathrm{T}} \\
 & \bot \quad\quad \mathrm{MAN}(c_1) \wedge \mathrm{FATHER}(c_1,c_0)^{\mathrm{T}} \\
 & \qquad\quad \mathrm{MAN}(c_1)^{\mathrm{T}} \\
 & \qquad\quad \mathrm{FATHER}(c_1,c_0)^{\mathrm{T}} \\
 & \qquad\quad \mathrm{MAN}(c_1) \Rightarrow \exists y.\mathrm{MAN}(y) \wedge \mathrm{FATHER}(y,c_1)^{\mathrm{T}} \\
 & \qquad\quad \vdots
\end{array}$$

$\forall\exists$-Constellation

What's happening here? Intuitively, our tableaux is generating and endless chain of fathers: John is a man, therefore he has a father. The father is a man as well and therefore he also has father who is a man again... . More formally, the pattern is as follows: The universal quantification is instantiated. This directly results in an implication, which is expanded next. While one of the resulting branches closes immediately, we get an existential quantification on the other one. And here we get into trouble. We have to introduce a new constant - but now we have got all it takes to produce exactly the same constellation again. We can re-instantiate the universal quantification (with our new constant) and finally get a new existential quantification, giving us a third constant etc. The crucial point is that we can go on like this forever. We can always produce new constants, allowing us to re-apply the universal rule indefinitely. We call such a pattern a $\forall\exists$-*constellation* .

10.1.4.1 Consequences for Model Generation

We did not start our tableaux with a (negated) theorem in the example before. This means what we hoped for was to arrive at a saturated tableaux, containing a model for our input. But we obviously never will.

Infinite Model

Nonetheless there is of course a model for our input. To convince ourselves of this, let us just take the set of natural numbers as the domain and choose $I(\text{MAN})$ as the set of natural numbers $I(\text{JOHN}) = 0$, and $I(\text{FATHER})$ to be the set of pairs $(n, n+1)$, where n is a natural number. Then 'Every man has a father' just means 'Every natural number has a successor', which is of course true in the natural numbers. The set of natural numbers is infinite and this is why our tableaux couldn't generate it. But the fact that our model generation procedure couldn't find a model based on the natural numbers does not contradict the claim that it would have found a *finite* model if there was one.

Finite Model Property

In essence, we have just shown by an example that full first-order logic no longer has the finite model property (page 150) (in contrast to PLNQ). Up to now, models have always been finite. With rule-like formulae such as the one we have been looking at, it is now possible to force models to be infinite. Of course the finite model property has now become highly desirable for our application of model generation in discourse interpretation - losing it is a considerable disadvantage.

Model Human Behaviour

But this consideration is largely theoretical, since we humans also face the same infiniteness problems; We could derive infinite chains of ancestors with the same world knowledge as well, yet we stop short of this in practice. To model this behaviour the model generation procedure would have to be equipped with a criterion when to stop saturation, even if it is not (theoretically) complete. Defining such a criterion is a current topic in research. In our implementation (as in most other implementations), we tackle the problem of infiniteness by simply fixing the maximal number of instantiations of an existential quantifier. If you allow 10 instantiations, our system will generate exactly 10 fathers for the example above and halt.

10.1.4.2 Consequences for Theorem Proving

As we've already said, the presence of quantifiers gives a huge boost in expressivity. We were happy to benefit from this fact. But with the loss of the finite model property we can now see the dark side of it. The consequences also extend to the use of tableaux for theorem proving. We can already easily see that if quantifier alternations like the $\forall\exists$-constellation in our example can keep us busy forever, we must be careful that they do not keep us from finding contradictions that would close our tableaux.

Loosing Decidability

In short, all our problems relate to the fact that first-order satisfiability is *no longer decidable*. So we have bought the added representational power with a loss of computational tractability. But we know what the problems are, and with this in mind, we can continue our theoretical considerations. Still for the implementation of our calculus, we really have lost something essential. Recall (page 149) that decidability of PLNQ, termination of the tableaux calculus for it, and termination of our implementation were all closely connected. Now that we've lost decidability, we cannot have a complete and correct calculus that terminates in general. Of course we have to keep this in our

mind when we extend our implementation - we will see that termination comes only at the cost of completeness. We will always have to invest a good deal of thought and work to arrive at an implementation that terminates but still gives useful results in many interesting cases. This is what we will do in the next section.

?- Question!

Things could be worse: At least first order satisfiability is *semi-decidable* . This means that whenver a formula is *not* satifiable, we can find this out by a *finite* computation - a matter of great importance to tableaux based theorem proving. Why is it so important?

10.2 Implementing First-Order Tableaux

10.2.1 The Existential Rule

Let us recapitulate our two new rules. Let us first look at the $\mathcal{T}(\forall)^{\mathbf{F}}$-rule. In Prolog, we write negative universal formulas as `false(forall(x,Scope))`. When we encounter an existential formula, we first introduce a new constant and then generate a new formula by substituting the new constant for the quantified variable `x` in `Scope`. E.g. `false(forall(x,walk(x)))` will license `false(walk(c7))` where `c7` is a new constant.

***See file* `foTabl.pl`.**

If we ignore the additional arguments of `tabl/6` for a moment, our implemented $\mathcal{T}(\forall)^{\mathbf{F}}$-rule (to be found in `foTabl.pl`) should be relatively easy to understand:

```
tabl(false(forall(X,Scope)),InBranch,OutBranch,QuantsIn,QuantsOut,Qdepth) :-
        !,newconst(NewConst),
        subst([X:NewConst],Scope,NewFormula),
        tabl(false(NewFormula),InBranch,OutBranch,QuantsIn,QuantsOut,Qdepth).
```

***See file* `signature.pl`. *See file* `substitute.pl`.**

The call to `newconst/1` (from `signature.pl`) creates a new constant of the form `c1,c2...` (using a counter). The predicate `subst/3` (from `substitute.pl`) in the case at hand does the same as the `substitute/3` we used in earlier chapters. It takes a substitution, e.g. `[x:c7,y:c5]` and a formula as input and returns the result of applying the substitution to the formula.

10.2.2 Universal Rule: Which Individuals to use?

As we have seen, our implementation of the $\mathcal{T}(\forall)^{\mathbf{F}}$-rule does not differ essentially from the propositional rules: We decompose an input formula and continue the tableaux building process with a less complex output formula. This means, that we practically throw away the input formula once we have decomposed it.

Quantifier must see *all* individuals

Can we proceed like this in the $\mathcal{T}(\forall)^{\mathtt{T}}$ case as well? We can't. Recall that (in contrast to all other rules) we have to be able to apply the $\mathcal{T}(\forall)^{\mathtt{T}}$ once with every individual. Suppose that a branch stands for the model $\{\text{MAN(JOHN)},\text{WOMAN(MARY)}\}$ at some time. If we now analyse a quantified formula like $\forall x.\text{LIKE}(x,\text{TWEETY})$, it's not enough to apply $\mathcal{T}(\forall)^{\mathtt{T}}$ only once, say with JOHN. We have to be able to instantiate its scope with both individuals and get LIKE(JOHN,TWEETY) and LIKE(MARY,TWEETY). And moreover if later a new individual, say $c7$ is introduced, we have to be able to conclude LIKE(C7,TWEETY). So simply throwing away the quantified formula will never be possible. We have to keep it somewhere in case we have to re-apply $\mathcal{T}(\forall)^{\mathtt{T}}$ later.

Store the quantifier

How can we do this in our implemetation? Our solution is quite simple: Whenever we analyse a formula of the form `true(forall(X,Scope))`, we not only instantiate `Scope` with one of the currently known individuals. We additionally put the quantified formula on a list so that we can re-use it later on. This is why our `tabl/6` predicate now has the additional arguments `QuantsIn` and `QuantsOut`. They serve to store the quantified formulae thus collected. We will refer to the list represented by this pair of arguments together as *quants* in the next sections.

Passing through

Thus each part of a tableaux may add one or more quantifiers to our quants-list (much like it possibly extends the model (branch), (see Section 8.1.1)). If you look at the conjunctive clause with the new arguments, you can see that the second call of `tabl/6` gets the resulting quantifier list (`QuantsOut1`) of the first call of `tabl/6` as input:

```
tabl(true(A & B),InBranch,OutBranch,QuantsIn,QuantsOut,Qdepth) :-
        !,tabl(true(A),InBranch,K,QuantsIn,QuantsOut1,Qdepth),
        tabl(true(B),K,L,QuantsOut1,QuantsOut,Qdepth),
        list_to_set(L,OutBranch).
```

The second call can thus see the universally quantified formulae found in the first call (and possibly re-apply $\mathcal{T}(\forall)^{\mathtt{T}}$ to them).

10.2.3 Restricting the Application of the Universal Rule

There may be infinitely many individuals.

But we immediately run into the next trouble. The interplay between $\exists$- and $\forall$-quantifiers may introduce infinitely many new individuals to a tableaux. Consequently the two input formulae $\forall x.\text{MAN}(x) \rightarrow \exists y.(\text{MAN}(y) \wedge \text{FATHER}(y,x))$ and MAN(JOHN) would be enough to let our program (in its present form) go off forever generating fathers MAN(JOHN),MAN(C1,FATHER(C1,JOHN)),MAN(C2,FATHER(C2,C1)) ... . We have to control when and how often $\mathcal{T}(\forall)^{\mathtt{T}}$ is (re-)applied .

Limiting instantiations

The solution to this is simple: We allow each quantifier only a limited number of instantiations. This is what the third new argument of `tabl/6` (`Qdepth`) is for: It formulates the upper bound on the number of instantiations.

Completeness lost

It is important to understand that this solution makes our algorithm *incomplete*: It may always happen that we only allow for n instantiations, where the (n+1)th instantiation would have closed our tableaux. So, we might fail to find out that some formula is a theorem only because we have stopped our tableau construction process too early. Still there is one thing we can do: We can use *iterative-deepening* . That means we can increase our `Qdepth` and re-run the tableaux construction (e.g. during lunchtime) to get a more exhaustive result. Iterating this process, we can approximate completeness, and this is all we can hope for. As we've discussed, there's a price we have to pay for the expressivity of first order logic: Satisfiability (and validity) is no longer decidable. Therefore no terminating implementation of a first-order calculus can ever be complete.

New instantiations

It is clever to make sure that a *new* instantiation is generated each time we expand a universal quantifier. One does not want to waste Qdepth on generating the same instantiations twice or more. We ensure this by attaching a list to each quantifier on our quants-list that records which individuals have already been used with some particular quantifier. So, each quantifier is put on the quants-list in a tuple of the form:

```
(forall(X,Scope),Inds)
```

The list `Inds` tells us to which individuals have already been used with the quantifier. At the same time we can read off the list how often it has been instantiated in total.

10.2.4 Universal Rule: The Prolog Clause

See file `foTabl.pl`.

We are now prepared to look at the implementation of the $\mathcal{T}(\forall)^{\mathtt{T}}$ clause of `tabl/6`. This clause generates the first instantiation of a $\forall$-quantifier-scope and puts the quantifier on the quants-list. In a minute, we will turn to the more subtle matter of when and how to do the subsequent instantiations with other individuals.

```
tabl(true(forall(X,Scope)),InBranch,OutBranch,QuantsIn,QuantsOut,Qdepth) :-
        !,hbaseList(InBranch,HBase),
        member(Ind,HBase),
        subst([X:Ind],Scope,NewInstantiation),
        tabl(true(NewInstantiation),InBranch,OutBranch,[(forall(X,Scope),[Ind]
```

See file `comsemLib.pl`.

First, `hbaseList/2` (from `comsemLib.pl`) generates the current Herbrand base from the current model. Then `member/2` picks some individual from the Herbrand base and `subst/3` generates the first (and therefore trivially new) instantiation of the quantifier's scope. For example, if our formula at hand is `true(forall(x,walk(x)))` and the current model (`InBranch`) is `[true(woman(mary)),true(man(john))]`, `HBase` will be instantiated to `[mary,john]`. The new instantiation will then be `true(walk(mary))`.

In the recursive call to `tabl/6`, the `QuantsIn`-list is extended by a tuple of the form `(forall(X,Scope),[Ind])`. In our example, this is the tuple `(forall(x,walk(x)),[mary])`. So, if we want to instantiate the quantifier again later on, we know that we have already instantiated it once before, namely with `mary`.

Who is *?

By definition, the universe of a model must not be empty. This means we must never have an empty Herbrand base. In order to avoid this, `hbaseList/2` is designed such that it never returns an empty list. If there is no constant on the branch so far (i.e. in the 'developing' model), we put in a joker individual `*` instead. One result of this is that we always have a constant for the first instantiation of a quantifier.

10.2.5 Universal Rule: Subsequent Instantiations

In the previous section, we have seen how we store a universal quantifier. What remains to be discussed is how to re-use the quantifiers from our quants-list. This happens in the base case of `tabl/6` (whenever we arrive at a literal). Before we look at the details, let us consider again the corresponding part of the implementation for PLNQ.

One propositional base clause

In the implementation of PLNQ we have the following (base) clause for literals:

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

Reaching this clause, the only thing to do is to add the new literal to the current model. If there is a contradiction, the predicate fails and we know we are on a closed branch. Otherwise we are...

1. ...either inside some tableau branch (if this was the last step of the first call of a conjunctive expansion). In this case, the second call takes over, or we are...
2. ... at the leaf of a saturated branch. In this case, we are done with this branch.

Four first-order base clauses

In the case of first-order expansions, the situation is different. If we reach an atomic formula, there may be quantifiers on our list that still can (and must) be applied. This gives us some choices. Therefore we now have four clauses ((a)-(d)) instead of our former single base clause. The first clause is harmless - it is almost identical to the base clause of our old implementation and does not have to care about quantifiers at all. It looks for a contradiction on our branch including the new literal. If it it finds one, it signals this using `fail`, and we know that the branch is closed (like in the old implementation).

***See file* `foTabl.pl`.**

```
% (a)

tabl(F,InBranch,OutBranch,_,_,_) :-
        OutBranch = [F|InBranch],
        clash(OutBranch),!,fail.
```

We do not want to (in fact we mustn't) consider any of the other clauses of our extended base case once we have found a clash. Hence the cut immediately after the clash test. But if the `clash/1`-test itself fails, we are not yet behind the cut. We can (and must) thus try the next clauses.

10.2.6 Subsequent Instantiations: Instantiate

Fairness

The rest of our four clauses together manage the successive fair instantiation of the quantifiers collected on the quants-list. In short, our strategy is to always make one instantiation of one quantifier every time we raech an atomic formula. Then we continue our tableaux with the instantiated scope. The next time we derive an atomic formula, we shift to the next quantifier, instantiate it, go on with the instance etc. If we shift past the end of our quants, we start again at the beginning. Rotating like this, we ensure *fairness* - every member of quants will get its share.

Enforcing termination

As far as described, our strategy would keep on rotating and instantiating the quants indefinitely. Of course we have to control this somehow. This is where the `Qdepth` argument comes in: We instantiate each quantifier at most `Qdepth`-times. Additionally, as we argued above, we will only make instantiations if they contribute something new.

The second base clause does all this. It takes a quantifier from the quants list, checks the side conditions just discussed, and eventually generates a new instance. Plus - after all it's a clause for literals - it extends the branch we're on by its input literal.

```
% (b)


tabl(F,InBranch,OutBranch,[(forall(X,Scope),SoFarInds)|RestQuants],QuantsOut,Q
        length(SoFarInds,LengthInds),
        LengthInds =< Qdepth,
        hbaseList([F|InBranch],HBase),
        member(Ind,HBase),
        \+ member(Ind,SoFarInds),!,
        subst([X:Ind],Scope,NewInstantiation),
        append(RestQuants,[(forall(X,Scope),[Ind|SoFarInds])],NewQuants),
        tabl(true(NewInstantiation),[F|InBranch],OutBranch,NewQuants,QuantsOut,
```

Let us go through the code in more detail. First, we use Prolog arithmetics to check if the given `Qdepth` is not exceeded. Second, with the help of member checks, we look for an individual that has not been used with the quantifier yet (i.e. we check if any instantiation would contribute anything new). If there is one, we take it (and the cut commits us to stick to the clause we are in). We then generate the new instantiation with our individual as explained earlier.

Finally, there is the recursive call with our new formula (the instantiated scope), where we have added the input literal `F` to our model. We know that there is no clash with that formula so far, because otherwise clause (a) would have applied. Note that we append

the quantifier tuple we have considered to the back of the quants-list. This is how we implement 'rotating' the quants-list.

The clause we've seen will fail once it arrives at a quantifier that has either used up it's full `Qdepth` or that has already been instantiated with all the individuals currently known. Next, we will see what happens in clauses (c) and (d), which handle these cases.

What are all the cuts for?

But before we go on, let us comment some more on the cuts in the clauses we're looking at: Note that we make double use of Prolog failure. On the one hand the explicit `fail` in clause (a) means as much as 'This branch is closed'. Backtracking should in this case directly lead to the next branch of the tableaux. But on the other hand we also use failure and backtracking to navigate between the three clauses (b), (c) and (d). As we shall see, one of these clauses will always succeed. The cut in clause (a) expresses the 'difference in meaning' between these two uses of fail.

We are right now talking about what happens if we do not reach the cut clause (a), that is if our branch is not closed and we must backtrack to clauses (b)-(d). As we have seen above, clause (b) generates new instantiations of the first universial quantifier on the quants list if possible. Next we will see what clauses (c) and (d) do in case clause (b) fails.

10.2.7 Subsequent Instantiations: If we can't instantiate

What do we have to do if the two base clauses explained before fail? In this case, we have reached an atomic formula, there is no clash and (more interestingly) we could not instantiate the first quantifier of our list. Hence we must be in one of two situations:

1. We have just finished expanding the first recursive call of a conjunctive expansion (i.e. the expansion of the first conjunct). In this case we just pass through our complete quants-list to the second conjunct and let it care for the subsequent instantiations. Maybe the second conjunct will produce more individuals. This may allow us to re-instantiate some of the quantifiers that could not find any new individuals before.
2. Or we are at the leaf of an otherwise saturated branch. In this case we will recursively try to instantiate all quantifiers on quants as often as possible.

Clause (c) handles the first of the discussed cases. It passes on the list of quantifiers:

```
% (c)

tabl(F,InBranch,[F|InBranch],Quants,Quants,_) :- !.
```

This leaves us with the second case: We're at a (current) leaf and want to work through the quantifiers that remain on our quants-list. Notice that we know that the first quantifier on the list can never be used again. Either it has used up its `Qdepth` or it has already been used with all the individuals there are. We know this because it's the reason why

we got here - otherwise the heading quantifier would have been instantiated in the last call to clause (b).

Clause (d) thus throws away the now useless quantifier and starts a recursive call with the rest of our quants-list:

```
% (d)

tabl(F,InBranch,OutBranch,[_|RestQuants],QuantsOut,Qdepth) :-
        tabl(F,InBranch,OutBranch,RestQuants,QuantsOut,Qdepth).
```

10.000$ Question!

We have claimed that clause (d) applies if and only if its input formula `F` is at a current leaf of the tableaux. Now the 10.000$ question is: 'Why? How do we know when we're at a leaf, and how do we enforce that clause (d) is applied then?'. The answer to this is a little tricky. First of all, we have to ensure that clause (d) is applied at all. To do so, we instantiate `QuantsOut` with `[]` when we call our `tabl/6`-predicate at top-level. Otherwise, our program would always use clause (c) when it reaches a quantifier that it cannot instantiate in clause (b). The quantifier would 'block' all others and our program would happily produce a full list of quantifiers on `QuantsOut` (headed by the 'blocking' quantifier), most of which could probably still be instantiated. By requiring that `QuantsOut` (finally) be empty, we force our program to apply clause (d) to remove the 'blocking' quantifier. In effect, we ensure that all quantifiers are instantiated as often as their `Qdepth` and the Herbrand base of the branch permit.

The answer

Now here is the central point of the answer to our question: If we reach a leaf (which is the last step in the construction of a branch), coindexing tells us that `QuantsOut` has to be the empty list - due to our top-level call. If there are quantifiers left on the `QuantsIn` list, we cannot enter the non-recursive clause (c) because the `...,Quants,Quants, ...` in its head clashes. So we correctly choose clause (d).

So much for the'if'-direction in our claim that clause (d) is applied *if and only if* its input formula is at a current leaf. Now for the *only if*: The cut in clause (c) takes care of this. We will always choose clause (c) instead of clause (d) whenever we're not at a current leaf (i.e. we are expanding some 'first conjunct'). Clause (c) comes first in the program, and no constraints are put on the value of `QuantsOut` at inner nodes of a tableaux branch. Only at a leaf, the head of clause (c) won't match because we've set the empty list as `QuantsOut` at top-level. Now the cut in clause (c) ensures that none of our *previous* choices to enter (c) is backtracked on this clash and redone entering (d). In effect clause (d) can only be used to empty a quants list that has been built before, not to build another one instead.

10.3 Running First-Order Tableaux

See file `fo.pl`.

We now provide driver predicates and give you a listing of all files you need in order to experiment with first-order theorem proving or model generation. The driver predicates (found in `fo.pl`) aren't very exciting:

```
modGen(Formula,Model,Qdepth) :-
        naonly(Formula,ConvFormula),
        tabl(true(ConvFormula),[],Model,[],[],Qdepth).


modGen(_,_,_) :-
        format("~nNo model found!~n",[]).
```

We have added a clause for existentially quantified formulae to `naonly/2` that re-writes formulae of the form $\exists x.\mathcal{A}$ into the equivalent $\neg\forall x.\neg\mathcal{A}$. The call to `tabl/6` requires the outgoing quantifier list to be empty. We argued in the last section that this is important for the control of the quantifier instantiations.

Try it!

`modGenWrite/3` is a version of `modGen/3` producing formatted output. Try it out: `modGenWrite((forall(x,man(x)>(exists(y,man(y) & father(y,x)))) & man(john)),M,10).` In this example, you can see that we really have to control the application of the quantifiers. For this input, our system might generate fathers till the end of time. Only our limited `Qdepth` ensures that it halts as soon as it has generated 10 fathers.

The driver for theorem proving (also found in `fo.pl`) is even simpler:

```
theorem(Formula,Qdepth) :-
        naonly(Formula,ConvFormula),
        \+ tabl(false(ConvFormula),[],_,[],[],Qdepth).
```

You can test it like this: `theorem((forall(x,man(x)v~man(x))),10).`

Modules

See file `fo.pl`.	The drivers for model generation and theorem proving.
See file `foTabl.pl`.	The tableaux itself: `tabl/6`
See file `comsemLib.pl`.	`naonly/2`, `hbaseList/2`
See file `comsemOperators.pl`.	Operator definitions.
See file `substitute.pl`.	`subst/3`
See file `signature.pl`.	`newconst/1`

10.4 Model Generation with Quantifiers

10.4.1 A New Problem

How do our rules fare if we want to generate models with our tableaux calculus? There's no problem with the $\mathcal{T}(\forall)^{\mathtt{T}}$-rule. Obviously, a model for a universal quantification may - in fact even has to - contain all the literals that come from any instance of the scope. And our rule allows to derive all of them (one after another). Let us look at a tableaux for 'Mutz is a siamese cat and Tiger is a siamese cat', where we know that 'Every siamese cat purrs' (to get a smaller tableaux, we use the derived inference rule for implication from Section 7.2.8):

A model that we like

$$\begin{array}{c}
\forall X(\text{SIAMESECAT}(X) \Rightarrow \text{PURR}(X))^{\mathtt{T}} \\
\text{SIAMESECAT}(\text{MUTZ})^{\mathtt{T}} \\
\text{SIAMESECAT}(\text{TIGER})^{\mathtt{T}} \\
\text{SIAMESECAT}(\text{MUTZ}) \Rightarrow \text{PURR}(\text{MUTZ})^{\mathtt{T}} \\
\text{SIAMESECAT}(\text{TIGER}) \Rightarrow \text{PURR}(\text{TIGER})^{\mathtt{T}} \\
\text{SIAMESECAT}(\text{MUTZ})^{\mathtt{F}} \mid \text{PURR}(\text{MUTZ})^{\mathtt{T}} \\
\bot \qquad\qquad \text{SIAMESECAT}(\text{TIGER})^{\mathtt{F}} \mid \text{PURR}(\text{TIGER})^{\mathtt{T}} \\
\qquad\qquad\quad \bot
\end{array}$$

The only branch that finally remains open on this tableaux contains a model where our world knowledge about purring siamese cats has been applied to Mutz as well as to Tiger: Both purr.

But now let's turn to the $\mathcal{T}(\forall)^{\mathtt{F}}$-rule. Let's look at a tableaux for the discourse 'Mutz is asleep. Tiger isn't purring. There is a siamese cat that's purring.' We get the following tableaux:

A model that we don't like

$$\begin{array}{c}
\text{SLEEP}(\text{MUTZ})^{\mathtt{T}} \\
\neg\text{PURR}(\text{TIGER})^{\mathtt{T}} \\
\neg\forall X\neg(\text{SIAMESECAT}(X) \wedge \text{PURR}(X))^{\mathtt{T}} \\
\text{PURR}(\text{TIGER})^{\mathtt{F}} \\
\forall X\neg(\text{SIAMESECAT}(X) \wedge \text{PURR}(X))^{\mathtt{F}} \\
\neg(\text{SIAMESECAT}(w_{new}) \wedge \text{PURR}(w_{new}))^{\mathtt{F}} \\
\text{SIAMESECAT}(w_{new}) \wedge \text{PURR}(w_{new})^{\mathtt{T}} \\
\text{SIAMESECAT}(w_{new})^{\mathtt{T}} \\
\text{PURR}(w_{new})^{\mathtt{T}}
\end{array}$$

This is a little disappointing. Of course we have generated a model for our input formula, but we have learned nothing new about Mutz or Tiger. Instead our calculus has invented a brand new siamese cat, and tells us that this cat purrs.

The problem

The core problem is that our $\mathcal{T}(\forall)^{\mathtt{F}}$-rule is designed to introduce a new constant every time it is applied. In effect it only generates models that have a new individual for each existential quantification. Such models are often strongly non-minimal. Non-minimality doesn't matter for theorem proving. But in model generation, we also want to get smaller models where existential quantifications are instantiated with individuals that are already there in the model. As we've convinced ourselves, we cannot achieve this by simply instantiating with one of the individuals that are already on the tableau. Otherwise we could incidentally say something wrong about that individual, introducing an unwarrranted contradiction (e.g. notice that if we had instantiated with TIGER on the above tableaux, we would have closed it without any good reason).

10.4.2 A special Rule for Model Generation

Still there's one thing that we can do to exploit the information in an existential quantification also with respect to the individuals that we already know of: We can *enumerate* all the individuals we know of, plus a new one, *disjunctively*. In terms of tableaux this means we will continue on a branch of its own with each of the constants that are

already there, and on yet another one with a newly invented constant. This is done by the following rule, which we will use instead of $\mathcal{T}(\forall)^{\mathbf{F}}$ when generating models:

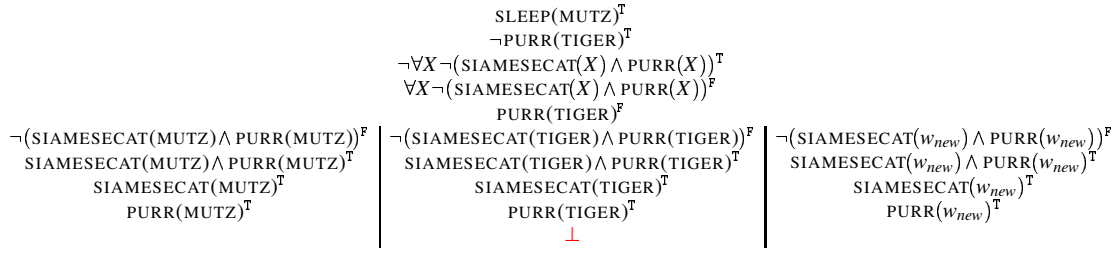
A new rule...

$$\frac{\forall x.\mathbf{A}^{\mathbf{F}} \quad \mathcal{H} = \{a_1, \ldots, a_n\}}{[a_1/X]\mathbf{A}^{\mathbf{F}} \mid \ldots \mid [a_n/x]\mathbf{A}^{\mathbf{F}} \mid [w_{new}/x]\mathbf{A}^{\mathbf{F}}}\,\mathcal{T}(\forall)^{\mathbf{F}}_{mg}$$

The rule makes a case distinction between the cases that the scope holds for one of the already known individuals (those in the Herbrand base) or a currently unknown one (for which it introduces a witness constant w_{new}).

Let us look at our example again, this time with the $\mathcal{T}(\forall)^{\mathbf{F}}_{mg}$-rule applied:

...fixes our problem



On the rightmost branch we get the model we could already generate with our old $\mathcal{T}(\forall)^{\mathbf{F}}$-rule. The middle branch, where we instantiated the quantified formula with TIGER, closes because we know that $\text{PURR(TIGER)}^{\mathbf{F}}$. But on the leftmost one we get a model that we couldn't generate before: Mutz purrs.

As an exercise (page 174), convince yourself that the rule $\mathcal{T}(\forall)^{\mathbf{F}}_{mg}$ is admissible!

10.5 Sidetrack: Derived Rules for the Existential Quantifier

$$\frac{\begin{array}{c}\exists x.\mathbf{A}^{\mathbf{F}} \\ \vdots \qquad c \in \mathcal{H} \end{array}}{[c/x]\mathbf{A}^{\mathbf{F}}}\,\mathcal{T}(\exists)^{\mathbf{F}} \qquad \frac{\exists x.\mathbf{A}^{\mathbf{T}}}{[w_{new}/x]\mathbf{A}^{\mathbf{T}}}\,\mathcal{T}(\exists)^{\mathbf{T}} \qquad \frac{\exists x.\mathbf{A}^{\mathbf{T}} \quad \mathcal{H} = \{a_1, \ldots, a_n\}}{[a_1/X]\mathbf{A}^{\mathbf{T}} \mid \ldots \mid [a_n/x]\mathbf{A}^{\mathbf{T}} \mid [w_{new}/x]\mathbf{A}^{\mathbf{T}}}\,\mathcal{T}(\exists)^{\mathbf{T}}_{mg}$$

In analogy to the $\mathcal{T}(\forall)$-rules, we will call the $\mathcal{T}(\exists)^{\mathbf{F}}$-rule *universal* and the $\mathcal{T}(\exists)^{\mathbf{T}}$-rules *existential* . It's probably obvious why our defined rules do what they should do.

10.6 Project: Adding Equality to our Calculus

logical constant

Generally, extending a logic with a new logical constant - equality is counted as a *logical constant* , since its semantics (page 16) is fixed in all models - involves extending all three components of the logical system: the language, semantics, and the calculus.

Two new rules

We have already considered the logical ramifications of this extension in Section 1.3 (we have just added a binary relation $=$ to the vocabulary, and $[\![a=b]\!]^{\mathcal{M}} = \mathtt{T}$, iff $[\![a]\!]^{\mathcal{M}} = [\![b]\!]^{\mathcal{M}}$). Thus we can concentrate on the calculus side here: We add two inference rules (a positive and a negative) for the new principal constant.

$$\frac{a \in \mathcal{H}}{a = a^{\mathtt{T}}} \qquad \frac{a = b^{\mathtt{T}} \quad \mathbf{A}^{\alpha}}{[b/a]\mathbf{A}^{\alpha}}$$

$\mathcal{H}\widehat{=}$ the Herbrand Base
the set of constants occurring on
the branch

An example

The following example shows how the second of the two rules works: If we simplify the translation of definite noun phrases, so that the phrase 'the teacher' translates to a concrete individual constant THE_TEACHER (of course one could think of more principled ways to treat definite noun phrases), then we can interpret (1 (page 173)) as (2 (page 173)).

1. 'Mary is the teacher. Peter likes the teacher'.
2. $\mathrm{MARY} = \mathrm{THE\_TEACHER}$ and $\mathrm{LIKES}(\mathrm{PETER}, \mathrm{THE\_TEACHER})$

Thus interpreting (1 (page 173)) leads to the following model generation tableau:

$$\begin{array}{c} \mathrm{MARY} = \mathrm{THE\_TEACHER}^{\mathtt{T}} \\ \mathrm{LIKES}(\mathrm{PETER}, \mathrm{THE\_TEACHER})^{\mathtt{T}} \\ \mathrm{LIKES}(\mathrm{PETER}, \mathrm{MARY})^{\mathtt{F}} \end{array}$$

In particular, we can test whether our two sentences entail that 'Peter likes Mary' using the method from the last section: the tableaux

$$\begin{array}{c} \mathrm{MARY} = \mathrm{THE\_TEACHER}^{\mathtt{T}} \\ \mathrm{LIKES}(\mathrm{PETER}, \mathrm{THE\_TEACHER})^{\mathtt{T}} \\ \mathrm{LIKES}(\mathrm{PETER}, \mathrm{MARY})^{\mathtt{F}} \\ \mathrm{LIKES}(\mathrm{PETER}, \mathrm{THE\_TEACHER})^{\mathtt{F}} \\ \bot \end{array}$$

confirms our intuition that Peter likes her.

A new closure rule

There's one small complication: An equality statement can be contradictory *in itself*. For example, there can be no model already for the literal $\neg\mathrm{MARY} = \mathrm{MARY}$ (respectively $\mathrm{MARY} = \mathrm{MARY}^{\mathtt{F}}$) alone. Thus we must close a tableaux branch on such a statement (and obviously we don't have to find a 'partner' for it, like normally when we look for contradictions). This is captured in the following rule:

$$\frac{\begin{array}{c} a \in \mathcal{H} \\ a = a^{\mathtt{F}} \end{array}}{\bot}$$

The project

Extend our implementation of first-order tableaux to include a treatment of equality as just discussed. [Hint: First of all you will of course have to decide what symbol to use for equality. You shouldn't use `=` or `==` for this purpose, because both already have a meaning in Prolog. You can e.g. use `===` or `eq` instead.]

10.7 Exercises

Exercise 10.1 *Run the implementation and perform some traces.*

Exercise 10.2 *Implement the existential rule for model generation (Section 10.4.2). [Hint: Mind where you put cuts. Look at the disjunctive rule.]*

Exercise 10.3 *[Theoretical]*

The way we use `hbaseList/2` *in the universal rule we strictly speaking do not get the full Herbrand base of the branch at the time of the call: We only consider the literals on the branch, hence we can only see constants that occur in literals. Why is this enough?*

Exercise 10.4 *The way we control the universal rule isn't fully fair (which means that even if we use iterative deepening (page 164), we won't truly approximate completeness with our implementation). To understand the problem, let's look at the input formula* $\forall x\text{MAN}(x) \wedge ((\forall y \exists z\text{FATHER}(z,y)) \wedge \forall u \exists v\text{MOTHER}(v,u))^{\mathtt{T}}$. *In this example our implementation will use up* any given `QDepth` *for* $\forall x\text{MAN}(x)$ *on the fathers generated from* $\forall y \exists z\text{FATHER}(z,y)$. *So it will* never *have any* `QDepth` *left to instantiate with mothers when it reaches* $\forall u \exists v\text{MOTHER}(v,u)$. *Try out the following call. It consists of two calls to* `modGenWrite/3`*, one with* `QDepth` *10 and one with 20 (plus some additional output):*

Try it!

```
modGenWrite(forall(x, man(x)) & (forall(y, exists(z, father(z,y))) & forall(u, exist
```

Thus with such input formulae, the `QDepth` *isn't distributed in a fair manner over new individuals coming from* all *conjuncts. Do you have an idea how to fix this? Can you implement your idea?*

Exercise 10.5 *[Theoretical]*

We say that an additional rule of a calculus is admissible *if it doesn't make the calculus incorrect or incomplete. This exercise deals with the admissibility of* $\mathcal{T}(\forall)^{\mathtt{F}}_{mg}$. *It is obvious that our tableaux calculus doesn't become incorrect if we exchange* $\mathcal{T}(\forall)^{\mathtt{F}}$ *for* $\mathcal{T}(\forall)^{\mathtt{F}}_{mg}$*: We cannot close any tableaux with* $\mathcal{T}(\forall)^{\mathtt{F}}_{mg}$ *that we couldn't close with* $\mathcal{T}(\forall)^{\mathtt{F}}$. *The reason is that whenver a tableaux remained open with the old* $\mathcal{T}(\forall)^{\mathtt{F}}$*-rule, the branch with the new constant remains open if we use the* $\mathcal{T}(\forall)^{\mathtt{F}}_{mg}$ *instead. But why does the calculus remain complete? In other words: Why don't we have more tableaux that remain open if we use* $\mathcal{T}(\forall)^{\mathtt{F}}_{mg}$ *instead of* $\mathcal{T}(\forall)^{\mathtt{F}}$*?*

Exercise 10.6 *Give the (sub)tableaux that the derived rules for the existential quantifier given in Section 10.5 abbreviate.*

11

Discourse Representation Theory

11.1 Discourse Phenomena

11.1.1 Anaphoric Pronouns

Pronouns are words that refer to objects in the text or situation in which they are uttered. We will focus on *anaphoric pronoun* s (pronouns that refer back to textual antecedents) and will mainly consider the personal pronouns 'he', 'she' and 'it'. Let's look at the following discourse with an anaphoric pronoun:

'A woman walks. She smokes.'

This discourse consists of two sentences. The second sentence contains the pronoun 'she' which refers to the noun phrase 'a woman' introduced by the first sentence. Let's try to translate this sentence into first-order logic, and let's try to do this in a systematic way. Recall from Chapter 1 that a sensible formula for the first sentence would be:

$$\exists x(\text{WOMAN}(x) \wedge \text{WALK}(x))$$

But what would be an appropriate first-order logic formula for the second sentence? One way to proceed is to translate the pronoun 'she' as a free variable:

$$\text{SMOKE}(x)$$

Now we can put together the translations of the two sentences together and get the following first-order formula for the entire discourse:

$$\exists x(\text{WOMAN}(x) \wedge \text{WALK}(x) \wedge \text{SMOKE}(x))$$

This formula is true in a model where there is an individual that has the properties of being a woman, walking, and smoking, and therefore correctly describes the meaning of our example discourse.

However, note that some mysterious operations took hold of the existential quantifier when we put the single translations of the two sentences together. After analysing the first sentence, the scope of the existential quantifier was restricted to $\text{WOMAN}(x) \wedge \text{WALK}(x)$. But its scope expanded after integrating the second sentence in order to include $\wedge\ \text{SMOKE}(x)$. In other words, we didn't construct this representation in a *systematic* way.

11.1.2 Donkey Sentences

A similar problem is also manifested in so-called *donkey sentence* s. (Incidentally, the example sentences that led to increased study of anaphoric pronouns in discourse in formal semantics, due to Peter T. Geach, staged as main characters donkeys and farmers.) One of the most famous donkey sentences is: ‘ Every farmer that owns a donkey beats it. ’

If we use our systematic method of translating into first-order logic and use free-variables to translate pronouns, we would get something similar to:

$$\forall x(\text{FARMER}(x) \wedge \exists y(\text{DONKEY}(y) \wedge \text{OWN}(x,y)) \rightarrow \text{BEAT}(x,y))$$

However, this is not a correct translation since there is an occurrence of a free variable (namely, the occurrence of y in BEAT(x,y)). In other words, it is not a logical sentence. So let’s try to repeat the trick that we applied in our previous example and extend the scope of the existential quantifier.

$$\forall x \exists y(\text{FARMER}(x) \wedge \text{DONKEY}(y) \wedge \text{OWN}(x,y) \rightarrow \text{BEAT}(x,y))$$

Unfortunately, our “scope-extending-trick” doesn’t work in this case. This formula does not assign the right truth conditions for the donkey sentence (Why is this so? Try to imagine a situation where there is a farmer owning a donkey and a pig, and not beating any of them. The above formula will be true in that situation, because for each farmer we need to find at least one object that either is not a donkey owned by this farmer, or is beaten by the farmer. Hence, if this object denotes the pig, the sentence will be true in that situation.).

A correct translation into first-order logic for the donkey sentence seems to be:

$$\forall x \forall y(\text{FARMER}(x) \wedge \text{DONKEY}(y) \wedge \text{OWN}(x,y) \rightarrow \text{BEAT}(x,y))$$

Recall from Chapter 3 that we translate determiners such as ‘every’ with universal quantifiers, and that the indefinite articles (such as ‘a’ and ‘some’) are translated into existential quantifiers. However, this seems not the case for donkey sentences. As the translation above shows, the indefinite noun phrase ‘a donkey’ is translated as a *universal* quantifier.

In other words, our approach about being systematic when it comes to translating natural language to semantic representations seems not to rhyme with a correct treatment of indefinite noun phrases: depending on the context, sometimes they are translated as existential quantifiers, and sometimes as universal quantifiers. But exactly when?

Exercise 11.1 *Give a first-order translation for the sentence ‘If a farmer owns a donkey, he beats it.’ Explain which quantifiers you chose to translate the indefinite noun phrase and motivate your choice.*

11.1.3 Another Puzzle

Before we propose a solution to deal with these unpleasant mismatches between natural language and first-order logic, let's have a look at a different, but somehow related, puzzle. Consider the following sentence:

'It's not the case that no woman walks.'

$$\neg\neg\exists x(woman(x) \wedge walk(x))$$

In first-order logic, this formula is equivalent with the translation for 'A woman walks'. However, although it is possible to continue this sentence with 'She smokes', it sounds ungrammatical for

' It's not the case that no woman walks. *She smokes. '

This observation leads to the conclusion that describing meaning in terms of truth-conditions alone might be appropriate for sentences, but certainly it is not enough to capture the meaning of discourses. There seems to be something else going on that blocks anaphoric references - something that we are unable to capture with our simple first-order translations for discourse.

11.2 Discourse Representation Structures

11.2.1 A First Example

Let's go through some examples to get a view of what it is to think in terms of these Discourse Representation Structures. Here is a first example:

As can be seen from this first example, a DRS is presented as a box-like structure, with so-called *discourse referent* s in the top part of the box and conditions upon these discourse referents in the lower part of the box. There are two discourse referents in this example (x and y), denoting 'a woman' and 'she', respectively. Discourse referents are entities mentioned in the discourse to which pronouns potentially can refer to. In our example, an anaphoric link has been established between 'she' and 'a woman' by virtue of the condition y=x.

One intuitive way of thinking of DRSs is to view them as partial descriptions of situations. In the example above we have a DRS describing a situation with two entities denoting the same object, an object which has the properties of being a woman, walking, and smoking.

11.2.2 Accessibility Constraints

The structure of DRSs plays a crucial role in pronoun resolution. In DRT, anaphoric pronouns are only allowed to refer to discourse referents that are *accessible*. As we will see shortly, DRSs are defined recursively, and accessibility is defined in terms of how the DRSs are nested into each other. Consider the following discourse and its DRS translation:

This DRSs contains one complex condition, where two DRSs are conjoined by the $\Rightarrow$ operator. This implicational condition can be interpreted as follows: if we are able to

extend the current situation with an entity having the property of being a woman, then this entity must walk. This complex condition is triggered by the use of the universal quantifier 'every', and has interesting consequences for pronoun resolution.

Now, what DRT stipulates is that discourse referents introduced by anaphoric pronouns can only establish links with *accessible* discourse referents. The discourse referent x introduced by 'Every woman' is not accessible from the viewpoint of discourse referent y, introduced by the pronoun 'She', because it is declared at a level deeper than the discourse referent of the pronoun. Hence DRT predicts that 'every woman' is not allowed as antecedent for the anaphoric pronoun 'she'. (And this is correct, hence the $*$ in the example).

But what exactly pins down this idea of accessibility? Before discussing some more examples, let's first give a formal definition of the syntax of DRSs and then define accessibility more precisely.

11.2.3 Syntax of DRSs and DRS-Conditions

1. If $x_1, \ldots, x_n$ are discourse referents ($n \geq 0$) and $C_1, \ldots, C_m$ ($m \geq 0$) are DRS-conditions then the following is a DRS:
$$\begin{array}{|c|}\hline x_1, \ldots, x_n \\ \hline C_1, \ldots, C_m \\ \hline \end{array}$$
2. If R is a relation symbol of arity n ($n > 0$), and $x_1, ..., x_n$ are some discourse referents, then $R(x_1, ..., x_n)$ is a DRS-Condition;
3. If τ_1 and τ_2 are first-order terms, then $\tau_1 = \tau_2$ is a DRS-Condition;
4. If B and B' are DRSs, then $B \Rightarrow B'$ and $B \vee B'$ are DRS-Conditions;
5. If B is a DRS, then $\neg B$ is a DRS-Condition;
6. Nothing is a DRS or DRS-Condition unless it can be shown to be so using clauses 1-5.

Here, first-order terms (clause 3) denote either discourse referents or constants. We sometimes refer to DRS-Conditions of the form licensed by clauses 2-3 as 'basic' conditions, while those licensed by clauses 3-5 are called *complex* conditions. Note that, in a way, DRSs bear a lot of similarities with the first-order logic formula syntax. As in first-order logic, we have the boolean connectors ($\Rightarrow$, $\vee$, $\neg$) to create nested boxes and express implication, disjunction, and negation. But unlike first-order logic, we don't have explicit conjunction, and we don't have explicit quantifiers. With a formal definition of the syntax of DRSs at our disposal, we are now in a good position to define *subordination* (a relation between DRSs), which then opens the way to define accessibility.

11.2.4 Subordination

Let B and B' be DRSs. Then B subordinates B' only if one of the following conditions hold:

1. B contains a DRS-condition of the form $\neg$ B ' ;

2. B contains a DRS-condition of the form $B' \Rightarrow B''$, for some DRS B'' ;
3. $B \Rightarrow B'$ is a DRS-condition of some DRS B'' ;
4. B contains a DRS-condition of the form $B' \vee B''$ or $B'' \vee B'$, for some DRS B'' ;
5. Some DRS B'' subordinates B', and B subordinates B''.

It is important to realise that subordination is defined differently for $\Rightarrow$ and $\vee$, although they are both two-place connectors in the DRS language.

11.2.5 Accessibility

Now, accessibility is informally defined as follows. Discourse referents of DRS B are accessible from DRS B' only if one of the following two conditions hold:

1. B subordinates B' ;
2. B and B' denote the same DRS.

Accessibility is by no means the only criterion for an antecedent to be classified as 'suitable' for an anaphoric pronoun. Many factors play a role in pronoun resolution, ranging from prosodic and syntactic information to topic-focus articulation and common-sense knowledge - discourse structure is just one of the factors that constrain resolution. Nevertheless, accessibility is a useful constraint, so let's look at some more examples supporting DRT.

11.2.6 Discourse Structure and Accessibility

Let's look at some more examples that show that DRS structure plays a central role for determining the possibility of anaphoric links between pronouns and their potential antecedents. Indefinite noun phrases normally introduce their discourse referents "locally" and hence are not accessible from outside a negation or implication.

First consider the following example:

Here we first construct a DRS for the first sentence, creating two discourse referents (x for John, and y for his watch). On interpreting the second sentence, the DRS gets extended and the pronouns 'he' and 'it' get resolved to John and his watch, respectively.

If we change this example slightly, using a conditional in the first sentence, we will get the following DRSs:

As we can see from this DRS, an 'if' connector introduces an implicational condition. Note that the indefinite noun phrase \nl{a valuable watch} is introduced in an embedded sentence (Further note that the DRS-conditions for the proper name \nl{John} are introduced in the main DRS. In a few moments we will explain why proper names behave this way.)

If we now attempt to continue the discourse with a pronoun referring to this watch, we fail to do so:

The reason why DRT correctly predicts this impossible anaphoric link runs as follows: The discourse referent v for the pronoun 'it' cannot be linked to y, because the DRS in

which v is introduced subordinates the DRS in which v is declared. Hence, the y is not accessible for v, and DRT correctly predicts this impossibility.

In this example an anaphoric link between 'it' and \nl{a milk shake} is allowed since the discourse referent y (denoting \nl{a milk shake}) is accessible for discourse referent v, introduced by the pronoun. In the following example, where we introduced negation, an anaphoric link is blocked, because here y is not accessible for v:

Similar observations can be made with respect to disjunction. The reader is asked to attempt the following exercise to find out what DRT predicts on anaphoric links to antecedents in disjunctive clauses.

Exercise 11.2 *Analyse the following examples and translate them into DRSs. Are anaphoric links between the pronoun 'it' and 'an apple' permitted?*

1. *'Bill eats an apple. It is delicious.'*
2. *Bill eats an apple or a pear. It is delicious.*

What do you think about this example?

- *Bill eats an apple or a pear. The apple is delicious.*

11.2.7 Proper Names

So far we have mainly discussed the relationship between anaphoric pronouns which have indefinite noun phrases as textual antecedents. We have seen that in contexts of negation, implication, and disjunction, DRT correctly predicts the impossibility of an anaphoric link between pronoun and antecedent. But matters are different with proper names.

Proper names always seem accessible for anaphoric links with pronouns, even if they occur inside a negation, implication or disjunction. DRT deals with the observation by promoting the discourse referents and conditions of proper names to the global DRS, or put differently, the outermost box.

Here is an example illustrating this point:

So, rather than introducing the discourse referent for 'Mary' in the consequent of the implicational condition (the DRS on the right-hand side of $\Rightarrow$), it is contributed to the top level of the DRS. Since y, the referent for 'Mary', is in the outermost box, it is accessible for any follow up pronouns. Therefore, the pronoun 'she' can pick up the discourse referent of 'Mary', and DRT correctly predicts that the discourse is felicitous.

11.2.8 Donkeys again

Let's return to the donkey sentences again and see how DRT analyses them. Consider the example 'Every farmer that owns a donkey beats it.' again:

Both 'every farmer' and 'a donkey' introduce a discourse referent in the antecedent DRS of the implicational condition. The pronoun 'it' introduces the discourse referent z in the consequent of the implicational condition. Following the definition of accessibility, the discourse referent y introduced by 'a donkey' is available as antecedent, a link that is established by the DRS-condition $z = y$.

11.2.9 Accessibility and Discourse Structure: Summary

The examples showed us how noun phrases introduce discourse referents and how pronouns co-refer to some of these noun phrases by referring to their discourse referents. Not all discourse referents are available for pronouns: the internal structure of discourse representations constrains the *accessibility* of discourse referents. The position of a discourse referent in the DRS determines whether it can be referred to by a pronoun, or put differently, whether it is *accessible* or not.

Accessibility itself is formally defined using the notion of *subordination* between DRSs. Informally, a DRS subordinates another DRS if the first encapsulates the second. A special case of subordination are implicational DRS-condition, where the antecedent DRS subordinates the consequent DRS. For a precise formalisation of subordination please see the definition (page 178).

11.3 Interpreting Discourse Representations

11.3.1 Embedding Semantics

The idea behind the "embedding semantics" is that DRSs are viewed as partial models. Now, what does this mean? Basically, it means that a DRS is true with respect to a model (for example one which describes the real world) if it can be *embedded* in that model. An embedding succeeds if the discourse referents of the DRS can be mapped onto entities of the model's domain in such a way that all the conditions of the DRSs are fulfilled in this model.

Recall from Chapter 1 that we define models as ordered pairs $< D, F >$, with D a non-empty finite domain of entities, and F an interpretation function mapping constants of the DRS language to elements or tuples of elements of D. As usual, we have $F_R \subseteq D^n_{\mathbf{M}}$ for an n-place predicate symbol R. Now consider the DRS for:

This DRS is true with respect to a model $< D, F >$ if we can find an assignment f such that $f(x) \in F(\text{WOMAN})$, $f(y) \in F(\text{DOG})$, and $\langle f(x), f(y)\rangle \in F(\text{WALK})$. This is the key intuition behind the partial match, and this will give us some idea as to how to define the model embedding semantics for DRSs in a formal way. Viewing assignments as partial functions from discourse referents to elements of $D_{\mathbf{M}}$, we come to the following definition:

11.3.2 Embedding Semantics for DRSs (Definition)

Assignment f verifies a DRS with discourse referents U and conditions C ($\langle U, C\rangle$) in a model $\mathbf{M} =< D, F >$ if there is an extension f' of f with the following properties:

1. f' is defined for U and for all discourse referents occurring in basic conditions in C;
2. If $R(x_1, ..., x_n)$ is in C then $\langle f'(x_1), ..., f'(x_n)\rangle \in F_{\mathbf{M}}(R)$;
3. If $\tau_1 = \tau_2$ is in C then $f'(\tau_1) = f'(\tau_2)$;

4. If $B^I \Rightarrow B^{II}$ is in C then every assignment that verifies B^I and agrees with f^I on all discourse referents that are not in $U_{B'}$, also verifies B^{II}.

5. If $B^I \vee B^{II}$ is in C then either there is an extension of f^I that verifies B^I in $\mathbf{M}$ or there is an extension of f^I that verifies B^{II} in $\mathbf{M}$;

6. If $\neg B^I$ is in C then no extension of f^I that is defined for all elements of $U_{B'}$, verifies B^I in $\mathbf{M}$.

According to the embedding semantics, two DRSs can have the same truth conditions while having different anaphoric potential. Put differently, the embedding semantics only describes the *logical* meaning of DRSs, not the *discourse* meaning. For this reason, the embedding semantics is sometimes referred to as a *static* semantics. Reformulations of the semantics of DRSs, where the interpretation of a DRS is described as a relation between assignment functions, capture the intuitions behind *discourse* meaning by describing meaning in terms of *context change potential*. Such approaches are known as *dynamic* semantics, and we will give a formal definition for dynamically interpreting DRSs.

11.3.3 Meaning as Context Change Potential

Defining DRS interpretation in terms of context change potential is done by relating each DRS with an ingoing and outgoing assignment function. But there are some other technical changes we make as well. First, we define an assignment f as a 'total' function (in other words, f is defined for any discourse referent used in a DRS). Second, the interpretation function $[\![\ ]\!]^{DRS}$ yields a set of assignment functions for conditions, and a set of pairs of assignment functions for DRSs. And finally, we need a device to express differences of assignment functions with respect to a set of discourse referents. We will write $f'[x_1,\ldots,x_n]f$ meaning that assignment f' differs at most from f in the values it assigns to the discourse referents $x_1,\ldots,x_n$.

11.3.4 Context Change Potential Semantics for DRSs (Definition)

1. $[\![\boxed{\begin{array}{|c|}\hline x_1,\ldots,x_n \\ \hline C_1,\ldots,C_m \\ \hline\end{array}}]\!]_{\mathbf{M}} = \{(f,f') | f'[x_1,\ldots,x_n]f \text{ and } f' \in [\![C_1]\!]_{\mathbf{M}},\ldots,f' \in [\![C_m]\!]_{\mathbf{M}}\}$;

2. $[\![R(x_1,\ldots,x_n)]\!]_{\mathbf{M}} = \{f | \langle f(x_1),\ldots,f(x_n)\rangle \in F_{\mathbf{M}}(R)\}$;

3. $[\![x{=}y]\!]_{\mathbf{M}} = \{f | f(x) = f(y)\}$;

4. $[\![B \Rightarrow B']\!]_{\mathbf{M}} =$
$\{f|$ if for all f': $(f,f') \in [\![B]\!]_{\mathbf{M}}$, then there is a f'' such that $(f',f'') \in [\![B']\!]_{\mathbf{M}}\}$

5. $[\![B \vee B']\!]_{\mathbf{M}} =$
$\{f|$ there is a f' such that $(f,f') \in [\![B]\!]_{\mathbf{M}}$ or there is a f' such that $(f,f') \in [\![B']\!]_{\mathbf{M}}\}$;

6. $[\![\neg B]\!]_{\mathbf{M}} = \{f|$ there is no f' such that $(f,f') \in [\![B]\!]_{\mathbf{M}}\}$.

In clause 1, the interpretation of a DRS is a *set of pairs* of assignment functions, where the second assignment function differs from the first assignment function possibly only with respect to the discourse referents in the domain of the DRS. The second assignment function must satisfy all conditions appearing in the DRS.

The intuition behind Clause 1 can be explained by viewing the ways a DRS is able to change the discourse context: the first embedding of the pair represents the current context, the second the change to the current context taking the meaning of the DRS into account. Hence, in this way, DRSs are not interpreted solely in terms of truth conditions (the logical meaning) but also in terms of their context change potential (discourse meaning).

The clauses 2–6 define the interpretation of DRS-conditions, denoting sets of assignment functions. Clauses 2 and 3, for instance, interpret basic conditions as the set of assignment functions that satisfy the basic condition with respect to the model. Clause 4 handles implicational conditions, clause 5 disjunction, and clause 6 negation.

11.3.5 Translations to First-Order Logic

Apart from giving a direct semantics for DRSs (as we formulated for the embedding semantics and for the context change potential semantics), it should be noted that it is also possible to translate the representations of DRT into first-order logic syntax. Once we are able to do that, we can just use the interpretation methods for first-order logic to give a semantics to DRSs.

The translation from DRSs to first-order logic is surprisingly simple, and can be accomplished by defining a translation function ($[.]^{fo}$, from DRS or DRS-conditions to first-order formulas) as follows.

$$[\langle\{\mathrm{x}_1,\ldots,\mathrm{x}_n\},\{\gamma_1,\ldots,\gamma_m\}\rangle]^{fo} \stackrel{\text{def}}{=} \exists\mathrm{x}_1\cdots\exists\mathrm{x}_n([\gamma_1]^{fo}\wedge\cdots\wedge[\gamma_m]^{fo})$$

This maps the discourse referents to existentially quantified variables, and recursively translates the conditions. The remaining clauses deal with the DRS-conditions. Basic conditions simply map to themselves, viewed as first-order atomic formulas:

$$[R(\mathrm{x}_1,\ldots,\mathrm{x}_n)]^{fo} \stackrel{\text{def}}{=} R(\mathrm{x}_1,\ldots,\mathrm{x}_n)[\tau_1=\tau_2]^{fo} \stackrel{\text{def}}{=} \tau_1=\tau_2$$

Moreover, complex conditions formed using $\neg$ and $\vee$ are also straightforwardly handled; we simply push the translation function in over the connective, leaving the connective unchanged:

$$[\neg B]^{fo} \stackrel{\text{def}}{=} \neg[B]^{fo}$$

$$[B_1\vee B_2]^{fo} \stackrel{\text{def}}{=} ([B_1]^{fo}\vee[B_2]^{fo})$$

Finally, complex conditions formed using $\Rightarrow$ are translated as follows.

$$[\langle\{\mathrm{x}_1,\ldots,\mathrm{x}_n\},\{\gamma_1,\ldots,\gamma_m\}\rangle\Rightarrow B]^{fo} \stackrel{\text{def}}{=} \forall\mathrm{x}_1\cdots\forall\mathrm{x}_n(([\gamma_1]^{fo}\wedge\cdots\wedge[\gamma_m]^{fo})\rightarrow[B]^{fo})$$

Translations as above are known to preserve the logical meaning. That is, given a model **M** and an assignment f, it can be shown that a DRS B is true in **M** with respect to f iff $[B]^{fo}$ is satisfied in **M** with respect to assignment f.

In the next section we will implement this translation in Prolog, and implement a way of building DRSs for English expressions in a systematic way.

11.4 Implementing DRT in Prolog

11.4.1 DRSs in Prolog

We will represent DRSs in Prolog as terms of the form `drs(D,C)`, where `D` is a list of terms representing the discourse referents, and `C` is a list of other terms representing the DRS conditions. To represent complex conditions we use the same operator definitions as for first-order logic (but note that we are not going to use the operator for conjunction, since we don't need it in DRT).

Let's consider an example, the DRS for:

And in Prolog we represent this DRS as:

```
drs([],[drs([X],[man(X)]) > drs([],[walk(X)])]).
```

The translation function discussed in the previous section (page 183) can now be implemented in Prolog as follows. We will use two predicates for this task: `drs2fol/2`, translating DRSs to first-order formulas, and `cond2fol/2`, translating DRS-conditions to first-order logic syntax.

```
drs2fol(drs([],[Cond]),Formula):-
   cond2fol(Cond,Formula).


drs2fol(drs([],[Cond1,Cond2|Conds]),Formula1 & Formula2):-
   cond2fol(Cond1,Formula1),
   drs2fol(drs([],[Cond2|Conds]),Formula2).


drs2fol(drs([X|Referents],Conds),exists(X,Formula)):-
   drs2fol(drs(Referents,Conds),Formula).
```

The way these three clauses recursively interact in the translation is by first working on the discourse referents, followed by translating the DRS-conditions. The third clause translates the discourse referents into existentially bound variables, the first clause translates exactly one DRS-condition, the second clause translates DRSs with at least two DRS-conditions.

```
cond2fol(~ Drs, ~ Formula):-
   drs2fol(Drs,Formula).
```

```
cond2fol(Drs1 v Drs2, Formula1 v Formula2):-
   drs2fol(Drs1,Formula1),
   drs2fol(Drs2,Formula2).


cond2fol(drs([],Conds) > Drs2, Formula1 > Formula2):-
   drs2fol(drs([],Conds),Formula1),
   drs2fol(Drs2,Formula2).


cond2fol(drs([X|Referents],Conds) > Drs2, forall(X,Formula)):-
   cond2fol(drs(Referents,Conds) > Drs2,Formula).


cond2fol(Condition,Formula):-
   \+ Condition = (~_),
   \+ Condition = (_ v _),
   \+ Condition = (_ > _),
   Formula=Condition.
```

Translating the DRS-condition is done by using the five clauses above for `cond2fol/2`. Straightforward are the first two clauses dealing with negation and disjunction. The third and fourth clause cover implicational conditions, and introduce universal quantifiers for discourse referents declared in the antecedent DRS of implications. The fifth clause, finally, deals with basic conditions.

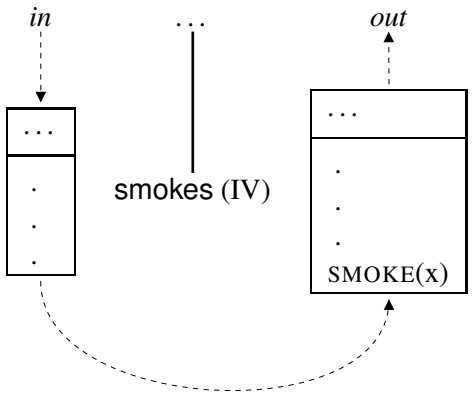
11.4.2 DRS Threading

Now that we are familiar with the syntax and semantics of the DRS language and with coding DRSs in Prolog, it is time to implement DRT for a fragment of English. Working with DRSs, as we will see shortly, is rather different than working with first-order formulas, and we will need to introduce some new machinery for constructing DRSs and performing pronoun resolution. Nevertheless, we will be able to reuse a lot of the software engineering work we did in the previous chapters - we will still be using semantic macros and semantic combination rules, and we will essentially use the same lexicon and grammar as before.

However, the way we will construct DRSs will be radically different from what we have seen before. We will use a method call *threading*, due to Johnson and Klein, a very intuitive approach to building semantic representations. Given a syntactic tree of an English expression, a DRS is threaded around the nodes of this tree in a left-to-right top-down way, thereby accumulating information in the DRS as it goes along.

11.4.3 A First Example

Let's look at a first example. How would threading work for intransitive verbs?



Note that the domain of the outgoing DRS is equal to the domain of the ingoing DRS, and that the condition set of the outgoing DRS consists of the conditions of the ingoing DRS 'plus' the basic condition introduced by the verb. It is clear that the node has simply made the expected semantic contribution as it was threaded through the DRS.

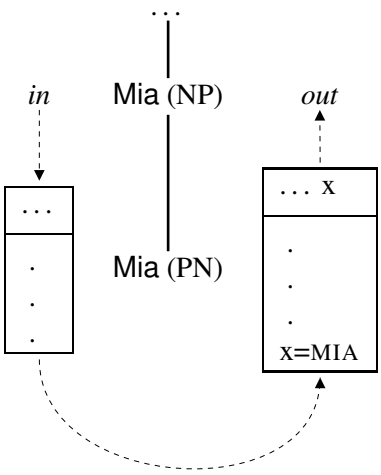
In the threading approach every node is associated with an 'ingoing' DRS, and an 'outgoing' DRS, and the difference between the ingoing and outgoing DRS is exactly the information that is contributed by the syntactic category. The good news is that thinking in terms of ingoing and outgoing DRSs can be coded in Prolog in a straightforward and declarative way. To capture the effect of the previous diagram, the semantic macro for an intransitive verb must be:

```
ivSem(Sym,[arg:Arg,drs:Drs,ana:[in:A,out:A]]):-
   compose(Cond,Sym,[X]),
   Arg = [subj:X,obj:_],
   Drs = [in:[drs(D,C)|S],
          out:[drs(D,[Cond|C])|S]].
```

There are three important points to be made at this point. First, note that a semantic representations is not a single unit, but a recursive feature structure. On the top level we have `arg`, for argument information, and `drs`, containing the in- and outgoing DRS, labelled by `in` and `out` respectively. The third top-level feature `ana` collects constraints on anaphoric bindings as we will see later.

Second, note that we are keeping track of the discourse referent `X` used as the argument of the verb. Thirdly, note that we are not threading a single DRS around nodes (as shown in the diagram), but a list of DRSs. The reason for this extra bookkeeping will become clearer shortly when we will be discussing more complex examples.

Let's now give a threading analysis of the sentence 'Mia dances'. First, we need to add a lexical entry for the proper name 'Mia'. Proper names introduce both a discourse referent and a condition. The following diagram shows what happens when a node labelled by 'Mia' is threaded through a DRS:

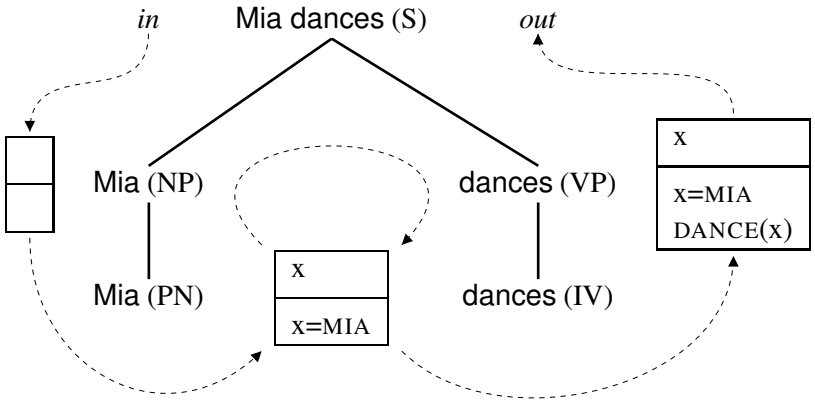


The following semantic macro turns this picture into a set of constraints:

```
pnSem(Sym,[arg:[index:X],drs:Drs,ana:[in:A,out:A]]):-
   Drs = [in:[drs(D,C)|S],
          out:Out,
          restr:_,
          scope:[in:[drs([X|D],[X=Sym|C])|S],
                 out:Out]].
```

As this macro shows, we introduced a number of new features here. Besides `in` and `out`, we also have `restr` and `scope`. These will play a role when we introduce quantified noun phrases (as we will explain in a moment), and the reason that we use them here for proper names is to keep the semantic representations in our grammar uniform and consistent.

That's the situation in the lexicon. So let's now see what happens as the DRS representing the previous discourse slides its way around the parse tree for 'Mia dances' (in the following diagram we assume that the initial DRS is empty):



The semantic rule that takes care of the threading of the NP and VP nodes is coded as follows:

```
combine(s1:S,[np2:NP,vp2:VP]):-
   S = [drs:Sem,ana:[in:A1,out:A3]],
   NP = [arg:[index:I],drs:Sem,ana:[in:A1,out:A2]],
   VP = [arg:[subj:I,obj:_],drs:Drs,ana:[in:A2,out:A3]],
   Sem = [in:_,out:_,restr:_,scope:Drs].
```

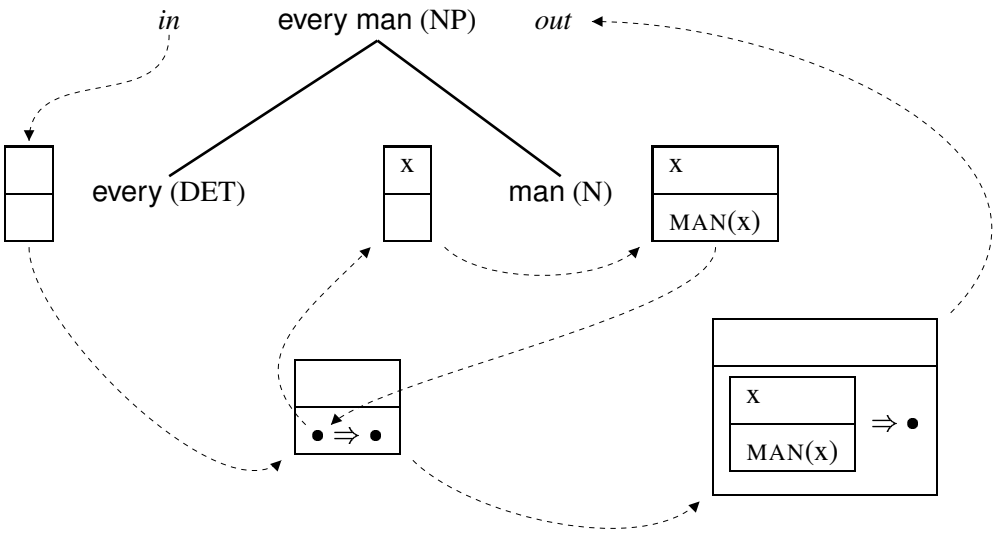
These constraints ensure that the information is packed into the DRS correctly. Note that the extra arguments percolated upwards by the noun phrase and verb phrase are unified; this ensures the correct bindings of the discourse referent with its conditions.

In Section 11.4.7 when we discuss pronon resolution, we will see how constraints on anaphoric bindings are collected in the `ana` feature. What you can see here already is that these constraints are passed through such that all constraints end up on the sentence level.

11.4.4 A second example (universal quantification)

How does DRS-threading extend to sentences containing determiners such as 'every'. Let's give a threading analysis of \nl{Every man runs}.

The semantic information associated with the the determiner 'every' is a complex structure: an implicational condition consisting of two DRSs. This is shown in the analysis for the noun phrase \nl{every man} (we assume that the ingoing DRS is empty):



The outgoing DRS for 'every' is the ingoing one to which an implicational condition (that is, an arrow linking two sub-DRSs) has been added. We don't yet know what the contents of these sub-DRSs are, so we represent each of them with a black hole •. Actually, we *do* know a little more: this implication is a universal quantification, hence the antecedent black hole must contain a discourse referent x; this is indicated by the third threading arrow. How is the condition set of the antecedent to be filled? By the contribution of the noun ('man' in this example), as the fourth threading arrow indicates. In short, once we've threaded every node in the noun phrase through the

initial DRS, we will have successfully filled in the antecedent black hole and now are ready to pass on the resulting incomplete DRS (incomplete because the consequent is still just a black hole) to the rest of the sentence for further threading. But before going any further, let's express this noun phrase analysis in Prolog. First, the semantic marcro determiners that introduce universal quantification:

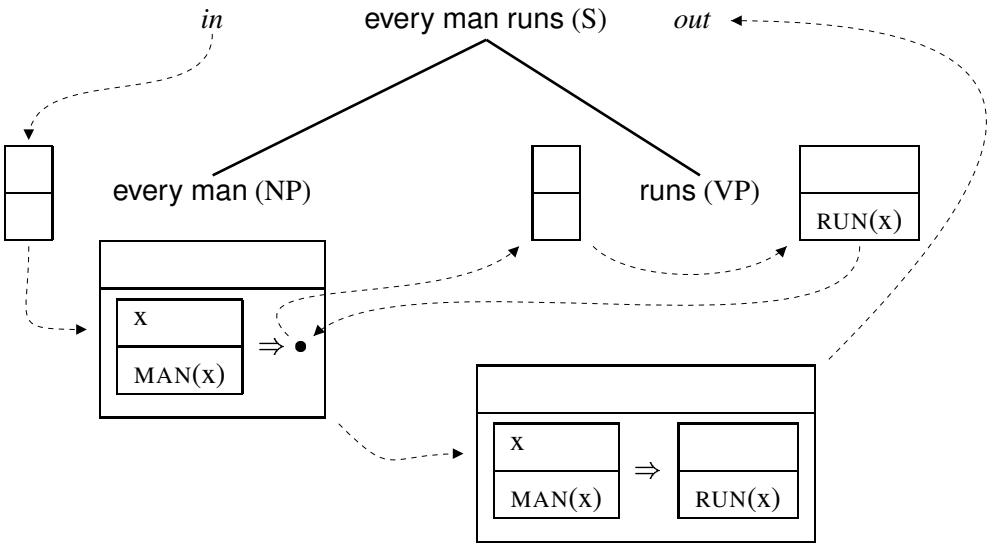
```
detSem(uni,[drs:Drs,ana:[in:A,out:A]]):-
   Drs = [in:[drs(D,C)|S],
          out:[drs(D,[Y>Z|C])|S],
          restr:[in:[drs([],[]),drs(D,C)|S],out:[Y|T]],
          scope:[in:[drs([],[]),Y|T],out:[Z|_]]].
```

Note that we have added two DRS-threading pairs: `restr`, which builds the DRS for the restriction (the first •, represented by the Prolog variable `Y`), and `scope` which builds the DRS for the nuclear scope (the second •, represented by `Z`). The outgoing DRS equals the ingoing DRS plus a new implicational condition.

Now for the combine rule that combines a determiner and a noun to form a noun phrase:

```
combine(np1:NP,[det:Det,n2:N]):-
   NP = [arg:Arg,drs:Drs,ana:[in:A1,out:A3]],
   Det = [drs:Drs,ana:[in:A1,out:A2]],
   N = [arg:Arg,drs:Restr,ana:[in:A2,out:A3]],
   Drs = [in:_,out:_,restr:Restr,scope:_].
```

Here `Drs` stands for the DRS-pair corresponding to the the overall DRS, and `Restr` for the DRS-pair that fills the first hole (the restriction of the quantifier). So, we now have an analysis for 'Every man', but this contains a black hole marking the missing nuclear scope information. How do we fill it? The verb phrase takes care of this, as the following analysis shows:



That is, we start threading the VP with an empty DRS and substitute the result for the black hole in the consequent of the implicational condition introduced by 'every'.

This completes our threading analysis of 'Every man runs', and all important the ingredients of DRS-threading have now been discussed.

11.4.5 Grammar rules for discourse

To finish off, we'll extend our grammar with some rules for discourse:

```
d2(D2)--> d1(D1), {combine(d2:D2,[d1:D1])}.


d1(D1)--> s2(S2), {combine(d1:D1,[s2:S2])}.
d1(D1)--> s2(S2), d1(D2), {combine(d1:D1,[s2:S2,conj,d1:D2])}.
d1(D1)--> s2(S2), {lexicon(coord,_,Word,Type)}, Word, d1(D2), {combine(d1:D1,[s
```

The first four rules allow us to make small discourses by sequencing sentences (we have introduced a new category D for 'discourse'). The second set of rules cover basic sentences and conditionals.

11.4.6 Driver predicate

It remains to design a driver predicate that gives the sentence to the DCG, outputs the DRS, translated the DRS into first-order logic and display the results as well.

```
drt:-
   readLine(Discourse),
   d2(DRS,Discourse,[]),
   printRepresentations([DRS]),
   drs2fol(DRS,FOL),
   printRepresentations([FOL]).
```

For instance, suppose we want to build the DRS for the discourse that contains of the sentence 'John walks' followed by \nl{A woman smokes}. To do this, consult the file `drt.pl`, start the DRT parser (by typing `drt.` at the Prolog prompt), and type in an example discourse:

```
?- drt.


> A man loves a woman. He smokes.


1 drs([A,B],[smoke(A),male(A),love(B,A),woman(A),man(B)])
1 exists(A,exists(B,smoke(A) & (male(A) & (love(B,A) & (woman(A) & man(B))))))
```

Note that the lists are built from right to left; that is, the most recent information comes first in the list.

Exercise 11.3 *Change the program in such a way that proper names are floated to the top DRS. There are two ways of doing this. The first method is using an explicit DRS that collects all proper names, which then will be merged with the DRS of the sentence once it is parsed. Another method, more elegant but also more technically demanding, is to extend the stack of DRSs that we already use for threading with a global DRS (first member of the stack) to collect all proper name information.*

11.4.7 Pronoun Resolution

The DRT implementation is able to resolve anaphoric pronouns as well, including those appearing in donkey sentences. Let's have a look at some examples (the second example is actually an instance of a donkey example):

```
> A man walks. He smokes.


1 drs([A],[smoke(A),male(A),walk(A),man(A)])
1 exists(A,smoke(A) & (male(A) & (walk(A) & man(A))))


> Every man that loves a woman likes her.


1 drs([],[drs([A,B],[love(B,A),woman(A),man(B)])>drs([],[like(B,A),female(A)])
1 forall(A,forall(B,love(B,A) & (woman(A) & man(B))>like(B,A) & female(A)))
```

How did we implement this? Let's have a closer look at the semantic macro for pronouns:

```
proSem(Sym,[arg:[index:X],drs:Drs,ana:Ana]):-
   Drs = [in:[drs(D,C)|S],
          out:Out,
          restr:_,
          scope:[in:[drs(D,[Cond|C])|S],out:Out]],
   compose(Cond,Sym,[X]),
   Ana = [in:A,
          out:[accessible(X,[drs(D,C)|S])|A]].
```

At first glance this semantic macros bears a great resemblance with the semantic macro for proper names. But there are two differences. The first difference is that there is no new discourse referent introduced. The second difference is that we add a contraint `accessible(X,...)` to our list of binding constraints. This constraint requires that there must be an accessible discourse referent for `X`in the incoming DRS.

11.4.8 Implementing accessibility

What's left to do is implementing accessibility. In the previous section we gave a nice and clean recursive definition of accessibility. Perhaps surprisingly, we will *not* follow

the formal definition to give its counterpart in Prolog. This is because the threading apparatus already gives us direct hands on accessibility. Recall that we don't just thread a single DRS along the nodes of the syntax tree - we use a list of DRSs. This list mirrors exactly the DRSs that are accessible from any given point in the syntax tree. A beautiful example of how this stacking of DRSs works was given in the DRS threading example for 'every man runs' above.

Let's return to the semantic macro for pronouns and the constraint on anaphoric binding `accessible(X,...)`. As we have seen, the constraints are handed through all `combine/2` predicates and end up on the sentence level and finally on the discourse level. The rest of the story is easily told. On the discourse (`d2`) level, a recursive predicate `checkConstraints/1` is called and handed over the list of constraints. They have the form of PROLOG predicate calls (e.g. `accessible(X,[drs(D,C)|S])`) and are simply called recursively:

```
combine(d2:Drs,[d1:D1]):-
   D1 = [drs:[in:[drs([],[])],out:[Drs]],ana:[in:[],out:Ana]],
   checkConstraints(Ana),
   bindingDrs(Drs).


checkConstraints([]).


checkConstraints([X|L]):-
   call(X),
   checkConstraints(L).
```

Finally, the predicate `accessible/2` looks in the space of DRSs, and then picks out one of the discourse referents (simply by using `member/2`). Because the threading rules for universal quantification, disjunction, and negation are set up, we will be sure that these discourse referents are accessible.

```
accessible(X,Space):-
   member(drs(D,_),Space),
   member(X,D).
```

11.4.9 Binding constraints

There are several constraints governing the resolution of pronouns to textual antecedents. The structure of discourse, a constraint which we implemented using the notion of accessibility, is just one of them. Let's have a look at the binding constraint, illustrated by the following sentences:

In the first example, the pronoun 'him' cannot have 'John' as antecedent. However, in the second example, the pronoun 'himself' can *only* have 'John' as antecedent. ('himself', 'herself', and so on, are called *reflexive* pronouns.)

Even though the intuitions beyond the binding constraint seem relatively easy to grasp, it is surprisingly hard to implement them. This is not only because there is an interplay between syntax and semantics, but also because there are subtle relaxations in the use of reflexive pronouns in other contexts (at least this holds for English). Consider for

instance the following example, where both the use of a normal pronoun or a reflexive pronoun are felicitous:

For these reasons, we will focus on ordinary pronouns in the scope of this chapter, and implement the binding constraint for pronouns in object position by checking the DRS for illegitimate DRS-conditions. Here the use of a simple heuristic will do: we won't allow basic DRS-conditions that have two identical arguments. This will prevent pronouns in object position of a transitive verb to refer to the subject noun phrase. The reader is asked to do the following exercise to get a full understanding of the implementation of this constraint in the DRS threading program.

Exercise 11.4 *The binding constraint heuristic is implemented with the help of the predicate* `bindingDrs/1` *in* `drt.pl`*. Check out the definition of this predicate, and find out when it is used. Think of cases where this heuristic might give the wrong predictions. Also think of ways to extend this heuristic to cover reflexive pronouns.*

11.4.10 Sortal Constraints

Another obvious constraint on resolving pronouns is of semantic nature. The English pronouns 'he' and 'him' refer to male objects, 'she' and 'her' to female objects, and 'it' normally to non-human objects. Our basic implementation ignores these kinds of sortal constraints, and happily identifies 'John' as an antecedent for the pronoun 'she':

```
> John walks. She smokes.


1 drs([A],[smoke(A),female(A),walk(A),A=john])
1 exists(A,smoke(A) & (female(A) & (walk(A) & A=john))
```

As our computer program doesn't have any information as to which John is male or female, it won't stumble over the fact that we refer to 'John' with the pronoun 'she'. So what would be a good way of introducing these sortal constraints in our DRT implementation?

First of all we need the sortal information of the pronoun at our disposal. Actually, this information is already present in the DRS, as we have specified this in the lexicon. For the example above, the pronoun 'she' introduced the DRS-condition `female(A)`.

Secondly, we need more information about other entities. We need to know, for instance, that John is a male, that women are female, the females are disjoint from males, and so on. In other words, we need a semantic ontology stipulating semantic relations between entities.

Finally, we need to use this ontology to filter out DRSs that violate sortal constraints. An interesting way of doing this is to translate the ontology into a first-order theory, use the first-order translation of the DRSs, and use a theorem prover (for instance the tableaux-based theorem prover we developed earlier in the course) to check for inconsistent information.

Exercise 11.5 *Extend the grammar and lexicon with the possessive pronouns 'his', 'her' and 'its'. Hint: think of these pronouns as determiners.*

11.5 Running DRT

The driver predicate is explained in Section 11.4.6.

Try this out: `drt([every,owner,of,a,siamese,cat,loves,john],SEM), printRepresentations(`

All files

See file `drt.pl`.	The drivers for DRS construction and the semantic macros.
See file `drs2fol.pl`.	The translation from DRT to predicate logic.
See file `discourseGrammar.pl`.	Grammar rules for discourse.

11.6 Compositional Approaches to DRT

The method of DRS-threading clearly has a lot going for it. This construction method is declarative and suggestive in the way to think about meaning in terms of context change potential. But on the other side, the construction rules get complicated and the level of complexity will further increase if one attempts to add other linguistic phenomena.

An alternative way of constructing DRSs is to use the lambda calculus as we have been doing in Chapter 4 for building first-order logic representations for English expressions - in other words, formulating a purely compositional account. This is a tempting direction to take, but applying the lambda-based techniques to DRSs will inevitably lead to several practical and theoretical problems. (And that's why we refrained from using the lambda calculus in this chapter). Let's identify the most important of these problems - the interested reading is referred to the "further reading" section at the end of this chapter for pointers to the literature.

First of all, one needs a new operator to merges two DRSs and a process that transforms merged DRSs into an ordinary DRS (merge-reduction). An example illustrating the merge operator ; and merge-reduction is the following:

This sounds straightforward, but a number of technical problems are lurking behind the corner. What, for example, if discourse referents with the same name are declared in two DRSs that are part of a merge. A case in point is the following DRS:

Does this mean these discourse referents denote the same object? Or do we want to rename one of them and then allow merge-reduction? Or shall we disallow merging in such cases (thereby closing off the scope of one discourse referent)?

Another obstacle that arises when combining the lambda calculus with a formalism based on dynamic semantics is the way variables are treated. Discourse referents can be viewed as objects that can bind variables out of their syntactic scope. This will lead to technical problems when combined with lambda-bound variables - variables that have a static semantics. As a result, we need to extend the process of β-conversion to avoid accidental bindings. Moreover, without postulating further constraints in the formalism, bindings can appear to be ambiguous. Consider, for instance the following example:

During β-conversion the argument will be duplicated and the two resulting occurrences of y will have different status: in the consequent of the implication y will be bound,

but in the disjunction, y will be free. This leads to a bizarre situation in the unreduced DRS expression above: is the occurrence of y in the argument free or bound?

Nevertheless, leaving some of these theoretical issues beside, using the lambda calculus can be of immediate practical importance when used as a pure "glue-language". In other words, we will use lambda-abstraction, functional application, and beta-conversion just as means to build discourse representations structures, and hence are not interested in semantically interpreting intermediate results. Probably the most convenient way of implementing the merge is to use a renaming operation on discourse referents to prevent clashes among variable names during the process of merge-reduction.

11.7 Further Reading

11.7.1 References

- Gamut, L.T.F. (1991): Logic, Language, and Meaning. Volume II. Intensional Logic and Logical Grammar. The University of Chicago Press.
- Kamp, H. (1981): A Theory of Truth and Semantic Representation. In Groenendijk, J., T.M.V. Janssen and M. Stokhof: Formal Methods in the Study of Language. p. 277-322. Mathematical Centre, Amsterdam.
- Kamp, H. and U. Reyle, Uwe (1993): From Discourse to Logic; An Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and DRT. Dordrecht, Kluwer.
- Kohlhase, M., S. Kuschert and M. Pinkal (1996): A Type-Theoretic Semantics for λ -DRT. In P. Dekker, and M. Stokhof, Proceedings of the Tenth Amsterdam Colloquium. Pages 479-498. ILLC/Department of Philosophy, University of Amsterdam.
- Kuschert, S. (1999): Dynamic Meaning and Accommodation. PhD Dissertation. Universit\"at des Saarlandes.
- Muskens, R. (1996): Combining Montague Semantics and Discourse Representation. Linguistics and Philosophy 19 (143-186).
- Van der Sandt, R. A. (1992): Presupposition Projection as Anaphora Resolution. Journal of Semantics 9 (333-377).
- Zeevat, H. (1989): A Compositional Approach to Discourse Representation Theory. Linguistics and Philosophy 12 (95-131).

12

The Proof of the Pudding is in the Eating

12.1 End term projects

The following exercise together are a suggestion for an end-term project.

Exercise 12.1 Model Inspection

Design a natural language-interface (using λ-calculus) to our revised model checker. The idea is that you can inspect some predefined model like

```
example(4,[bird(tweety),siamesecat(mutzi),woman(mary),man(john),man(miles),
        love(miles,mary),owner(mary),of(mary,mutzi),
        love(miles,tweety),love(john,mary),walk(mary),
        therapist(mary),eat(mutzi,tweety)]).
```

by asking questions like

- *'Does John love Mary?'*
- *'Which owner of a siamese cat walks?'*

Hint: For the time being, a cursory treatment of questions is enough for our purposes here. It should lead to the following representations of the questions given above:

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

and

$$\lambda x \exists y.\text{SIAMESECAT}(y) \wedge \text{OWNER}(x) \wedge of(x,y) \wedge \text{WALK}(x)$$

The expected answer to the first question is 'yes' in the example. The answer to the second question should consist of all values of x that make the scope of the abstraction true (in our case 'Mary').

Exercise 12.2 Model Extension

Use our tableaux-based model generation procedure to extend models like the one presented in !!!UNEXPECTED PTR TO EX_EX.PUTTING1!!!. An example run could look like this:

```
1 ?- extendModel(5,10).

 before: [bird(tweety), siamesecat(mutzi), woman(mary), man(john), man(miles)

> john loves every siamese cat.

 after: [eat(mutzi, tweety), therapist(mary), walk(mary), love(john, mary), lo
```

This example also gives you an impression of what direct, uncontrolled model generation does. First of all, if a speaker utters 'John loves every siamese cat.' in the situation described by our input model, one thing she probably wants to communicate is that 'John loves Mutzi'. But our system also has generated lots of other facts, such as `~siamesecat(john)`*, that are not contradictory (up to now) although they probably weren't intended by the speaker. Moreover, the system has blindly chosen one of many possible extensions of the input model. Another possible extension would e.g. contain* `siamesecat(john), love(john,john)` *instead of* `~siamesecat(john)`*. Now what exactly makes the extension of mental models in human language understanding so much more focussed than what our implementation does? Do you have any ideas (speculate, you don't have to implement...)?*

Exercise 12.3 Informativity Checking

Remember how we can use tableaux to check the informativity of a given formula with respect to some other formula(e) (see Section 7.2.5). Implement a little program that checks the informativity of a natural language sentence with respect to some world knowledge. For example suppose you have some world knowledge that tells you that 'Every human works', that 'John is a man' and that 'All men are human':

```
wk(man(john)
   & forall(x,human(x)>work(x))
   & forall(y,man(y)>human(y))
   & forall(z,woman(z)>human(z))).
```

Now, your system should check the informativity of input sentences like this:

```
1 ?- checkInf.

> john works.

*** Doesn´t interest me... ***

Yes
2 ?- checkInf.
```

```
> john does not work.

*** This is impossible! ***

Yes
3 ?- checkInf.

> tweety works.

*** Maybe... ***

Yes
```

Common Predicates

compose/3

See file **`comsemLib.pl`.**

The predicate `compose/3` (de-)composes complex terms out of (into) the functor and a list of its arguments. Possible usage:

```
compose(+Term,?Functor,?ArgsList)   compose(walk(mary),F,A)
                                    compose(1+2,F,A)
compose(-Term,+Functor,+ArgsList)   compose(Term,walk,[mary])
                                    compose(Term,'+',[1,2])
```

newvar/1

See file **`signature.pl`.**

This predicate returns in its argument a new variable according to our convention (page 20): `v1,v2,...,vn`.

```
newvar(V),write(V).
```

Each call of `resetVars/0` sets the variable counter back to 0.

printRepresentations/1

See file **`comsemLib.pl`.**

Prints all complex terms from its input *list* in a human readable fashion. Each term is put in a new (numbered) line. Second, the Prolog variables are displayed as capital letters (or the like, this is done by the Prolog built-in predicate `numbervars/3`).

Note that the variables are replaced for each term (from `Terms`) beginning from scratch. So, variable coindexing between terms (from line to line) is not possible (see the example below). Coindexing within one term is handled, of course.

```
printRepresentations([p(MyVar1),q(MyVar2),r(MyVar1,MyVar2),r(MyVar1,MyVar1),z(_,_)])
```

readLine/1

See file **`readLine.pl`.**

The auxiliary predicate `readLine/1` prompts the user to type in a sentence and returns our well-known list representation of the sentence. If the user types in “John walks”, `readLine/1` instantiates its argument with [john,walks].

substitute/4

***See file* `comsemLib.pl`.**

This predicate implements a version of the Sterling and Shapiro `substitute/4` predicate. It takes a term, a variable, and a formula as its first three arguments, and returns in its fourth argument the result of substituting the term for each free occurrence of the variable in the formula. Because this is an important predicate (we shall use it again when we implement a first-order theorem prover), we recommend that you look at its definition.

```
substitute(woman,P,lambda(Q, exists(X,P@X& Q@X)),Result), printRepresentations([Resu
```

vars2atoms/1

***See file* `signature.pl`.**

The predicate `vars2atoms/1` extracts all free (Prolog-) variables from the input term with the help of the Prlog in-built predicate `free_variables/2`. These variables then are instantiated with fresh and distinct variables using `newvar/1` (see Section 12.1).

```
vars2atoms(lambda(A, A@tweety)@lambda(B, smoke(B))),write(lambda(A, A@tweety)@lambda
```

Calling `resetVars` (see Section 12.1) resets the variable counter.

Code Index

Index