Representation and Inference for Natural Language:

A First Course in Computational Semantics

Lecture notes for ESSLLI97,
Aix–en–Provence,
August 18–22, 1997.

Patrick Blackburn & Johan Bos

September 25, 1997

Contents

Preface

These are the course notes for *Representation and Inference for Natural Language: A First Course in Computational Semantics*, which will be given in the second week of ESSLLI97, Aix-en-Provence, by Patrick Blackburn and Johan Bos. The course is an introduction to computational semantics in PROLOG. It introduces some fundamental techniques for computing semantic representations for natural language, and working with the result. Both the underlying theory, and their implementation in PROLOG, are discussed. We believe that the reader who masters these techniques will be in a good position to appreciate (and critically assess) ongoing developments in computational semantics.

Computational semantics is a relatively new subject, and trying to define such a lively area (if indeed it is a single area) seems premature, even counterproductive. However, for the purposes of this course we take 'semantics' to mean 'formal semantics' (that is, the business of giving model-theoretic interpretations to fragments of natural language, usually with the help of some intermediate level of logical representation, with the aim of pinning down the notion of logical consequence) and 'computational semantics' to be the business of using a computer to actually build such representations (*semantic construction*) and reason with the result (*inference*). Thus these notes are devoted to introducing some fundamental techniques for tackling the following two questions:

1. *How can we automate the process of associating semantic representations with expressions of natural language?*
2. *How can we use logical representations of natural language expressions to automate the process of drawing inferences?*

Incidentally, our working definition isn't quite as innocent as it looks. Many semanticists claim that intermediate levels of logical representation are essentially redundant. Whether or not this is true, the move to a computational perspective on formal semantics certainly increases the *practical* importance of the representation level. Logical representations — that is, formulas in some logical language — encapsulate meaning in a clean compact way. Moreover, they offer us the possibility of making use of well understood proof systems to perform inference, and we shall learn how to exploit this possibility. Models may be the heart of semantics, but representations are central to its computational cousin.

The chapter by chapter breakdown is as follows:

Chapter 1. First-Order Logic. Here we introduce the syntax and semantics of first order logic (the representation language used in the first half of the course), show how formulas and (simple) models may be represented in PROLOG, define a simple PROLOG model checker, discuss some of the problems that arise when PROLOG variables are used to represent first-order variables, and conclude with a discussion of propositional calculus, function symbols, and equality.

Chapter 2. Lambda Calculus. Here we start studying the heart of computational semantics: how to automate the process of semantic construction. We start by outlining the methodology underlying our attempts (namely, *compositionality*) and motivate our use of PROLOG's Definite Clause Grammars. We then present two attempts to build semantic representations for a small fragment of English. These experiments lead us to the lambda calculus, the tool we shall use for performing compositional semantic construction. We show that the lambda calculus can be straightforwardly implemented in PROLOG and use it to give a semantics to our little fragment. We conclude with some general remarks on the lambda calculus.

Chapter 3. Quantifier Scoping. Here we tackle one of the central problems in computational semantics: coping with quantifier scope ambiguities. While the lambda calculus provides the fundamental technique needed to drive semantic construction, new ideas are needed to cope with sentences containing multiple quantified noun phrases. We illustrate the problem, and then present three (increasingly more sophisticated) solutions: Montague's quantifier raising approach, Cooper storage, and nested Cooper storage.

Chapter 4. Propositional Inference. Here we start examining the second major theme of the book: how to perform inferencing using the representations we now know how to build. Our aim in these notes is to discuss, and implement, an inference system adequate for full first-order reasoning. In this chapter we take the first step towards this goal. We introduce a *signed tableaux* proof system for propositional calculus and show how to implement it in PROLOG. We conclude by informally discussing a number of theoretical issues.

Chapter 5. First-Order Inference and Unification. Here we extend our propositional signed tableaux system to full-first order logic — and immediately run into a problem. Naive approaches to first-order theorem proving are doomed to hopeless inefficiency. So we rethink the problem. Instead of asking the tableaux system to solve the entire inference problem, we reformulate it slightly so that it generates a collection of constraints on possible solutions. We then feed these constraints to a *unification* component, which (hopefully) will compute a useful solution. As we shall see, this new approach does lend itself to computational implementation.

Chapter 6. Putting it all together. As its name should suggest, in this chapter we combine the techniques we have acquired so far. By 'plugging together' our lambda calculus, quantifier storage, model checking and theorem proving programs we are able (more or less immediately) to define simple question handling and argumentation predicates. Does Vincent give every woman a foot massage? Which robber loves Honey Bunny? Find out in this week's exciting episode ...

Chapter 7. Discourse Representation Theory. This chapter is an introduction to the basic ideas of Discourse Representation Theory (DRT) and how to implement them in PROLOG. The fundamental aim is to show that the key ideas of DRT can be straightforwardly combined with lambda calculus driven approaches to semantic construction. The remainder of the chapter is devoted to anaphora resolution and translating Discourse Representation Structures into first-order logic.

Every chapter concludes with a section called 'Notes', which lists references the reader may find helpful.

We have tried to make these lecture notes self-contained, and believe that readers with even relatively modest backgrounds in linguistics and logic should be able to follow the theoretical discussion. But the fact remains that this is a course in *computational* semantics, and getting the most out of it requires a working knowledge of PROLOG. We have included an appendix which outlines the main ideas of PROLOG, and we hope that this will be useful — but, bluntly, the reader who has never written a PROLOG program will find parts of it tough going. Much of the material (for example, our account of the lambda calculus in Chapter 2) is developed from a computational (and indeed, specifically PROLOG oriented) perspective.

These notes developed out of material for a course on Computational Semantics regularly offered at the Department of Computational Linguistics, University of Saarland. When designing this course, we found no single source which contained all the material we wanted, which is what prompted us to start writing. (That said, it's easy to point to the principal influences on these notes: Pereira and Shieber [PS87] for semantic construction, Fitting [Fit96] for theorem proving, and Kamp and Reyle [KR93] for DRT. We strongly advise the reader to get hold of these books.) The notes are not yet complete — we want to say a lot more about DRT, for example — but we are continually extending them, with the aim of eventually turning them into a relatively comprehensive (and comprehensible) introduction to computational semantics. In the meantime, we'd be very grateful for feedback on the present version.

Enjoy the course!

Patrick Blackburn and Johan Bos,
Computerlinguistik, Universität des Saarlandes.

Chapter 1

First-Order Logic

First-order logic is the logical formalism used in the first half of this book to represent the meaning of natural language sentences. The main purpose of this chapter is to introduce first-order logic, and to write some simple PROLOG programs that work with it.

We proceed as follows. First, we review the syntax and semantics of first order logic (without function symbols or equality). Second, we write a simple first-order model checker (or semantic evaluator) in PROLOG. The model checker takes a first-order formula and a first-order model and checks whether the formula is satisfied in the model. As we shall see, such a model checker is extremely simple to implement if we use PROLOG variables to simulate first-order variables. However this strategy can lead to problems if carried out naively. In the third part we discuss these problems and show how to deal with them. We conclude with a discussion of propositional calculus, function symbols and equality.

1.1 First-Order Logic

In this section we review the syntax and semantics of first-order logic. We first introduce the concepts of *vocabularies*, *first-order models* and *first-order languages*, and then tie these concepts together via the crucial *satisfaction* definition.

Vocabularies

Our ultimate goal is to define how first-order formulas (that is, certain kinds of syntactic entity) are evaluated in first-order models (that is, certain kinds of semantic entity). But, intuitively, we can't evaluate just any formula in any model. For example, if we are given a formula from a first-order language intended for talking about the various relationships and properties (such as loving, being a robber, and being a customer) that hold between Mia, Honey Bunny, Vincent, and Pumpkin, and we are given a model that contains information about something completely different (for example, which household cleaning products are best at getting rid of particularly nasty stains) then it doesn't really make much sense to evaluate this particular formula in that particular model. *Vocabularies* tell us which first order languages, and first-order models 'belong together'.

Here is an example of a vocabulary:

{ (LOVE,2),
(CUSTOMER,1),
(ROBBER,1),
(MIA,0),
(VINCENT,0),
(HONEY-BUNNY,0),
(PUMPKIN,0) }

Intuitively, this vocabulary is telling us two important things: the topic of conversation, and the language the conversation is going to be conducted in. Let's spell out the formal content of this.

First, the vocabulary tells us *what* we're going to be talking about. In the present case, we going to be talking about *loving* (the 2 indicates that loving is regarded as a two place relation) and the properties (or 1-place relations) of being a *customer* and being a *robber*. In addition to these relations we're going to be talking about four special entities named *Mia*, *Honey Bunny*, *Vincent*, and *Pumpkin* (the 0s indicate that these symbols are constants, or names).

Second, the vocabulary also tells us *how* we can talk about these things. In the above case it tells us that we will be using a symbol LOVE of arity 2 (that is, a 2-place symbol) for talking about loving, two symbols of arity 1 (CUSTOMER and ROBBER) for talking about customers and robbers, and four constant symbols (or names), namely MIA, VINCENT, HONEY-BUNNY, and PUMPKIN for naming certain entities of special interest.

In short, a vocabulary gives us all the information needed to define the class of models of interest (that is, the *what*) and the relevant first-order language (that is, the *how*). Let us now look closer at what these models and languages actually are.

First-Order Models

Suppose we've fixed some vocabulary. What should a first-order *model* over this vocabulary be?

Actually, our previous discussion has pretty much given the answer. A model is a *semantic* entity: it contains the kinds of things we want to talk about. Thus a model for a given vocabulary gives us two pieces of information. First, it tells us which set of entities we are talking about. This set of entities is usually called the domain, and is symbolised by D. Second, for each symbol in the vocabulary, it gives us an appropriate semantic entity, built from the items in D. This second task is carried out by a function F which specifies, for each symbol in the vocabulary, what its semantic value is. We call such functions *interpretation functions*. Thus, in set theoretic terms, a model $\mathbf{M}$ is an ordered pair (D, F) containing these two pieces of information.

What are 'appropriate semantic entities'? There's no mystery here. As constants are names, each constant should be interpreted as an element of D. (That is, for each constant symbol c in the vocabulary, $F(c) \in D$.) As as n-place relation symbols are intended to 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.)

Let's consider an example. We shall define a simple model for the vocabulary given above. First, let D be $\{d_1, d_2, d_3, d_4\}$. That is, we have chosen a domain containing four items. To use some traditional terminology, this four element set is the 'universe of discourse' of our little model.

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

$F(\text{MIA}) = d_1$
$F(\text{HONEY-BUNNY}) = d_2$
$F(\text{VINCENT}) = d_3$
$F(\text{PUMPKIN}) = d_4$
$F(\text{CUSTOMER}) = \{d_1, d_3\}$
$F(\text{ROBBER}) = \{d_2, d_4\}$
$F(\text{LOVE}) = \{(d_4, d_2), (d_3, d_1)\}$

Note that every symbol in the vocabulary does indeed correspond to an 'appropriate semantic entity'. The four names pick out individuals, the two arity 1 symbols pick out subsets of D (that is, properties, or 1-place relations on D) and the arity 2 symbol picks out a 2-place relation on D. Intuitively, in this model d_1 is called Mia, d_2 is called Honey Bunny, d_3 is called Vincent and d_4 is called Pumpkin. Both Honey Bunny and Pumpkin are robbers, while both Vincent and Mia are customers. Pumpkin loves Honey Bunny and Vincent loves Mia.

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

$F_2(\text{MIA}) = d_2$
$F_2(\text{HONEY-BUNNY}) = d_1$
$F_2(\text{VINCENT}) = d_4$
$F_2(\text{PUMPKIN}) = d_3$

$F_2(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F_2(\text{ROBBER}) = \{d_3\}$
$F_2(\text{LOVE}) = \{(d_3, d_4)\}$

In this model, three of the individuals are customers, and only one is a robber.

There is one general point worth emphasising. Both previous models have two special properties, namely:

1. Every entity in D is named by some constant.
2. Every entity in D is named by only one constant.

Neither of these properties is necessary For example, suppose we add two more entities to the domain of the first model; that is, we create a new domain $D' = D \cup \{d_5, d_6\}$. We then define a new interpretation function F' on D' as follows:

$F'(\text{MIA}) = d_2$
$F'(\text{HONEY-BUNNY}) = d_1$
$F'(\text{VINCENT}) = d_4$
$F'(\text{PUMPKIN}) = d_3$
$F'(\text{CUSTOMER}) = \{d_1, d_2, d_4, d_5\}$
$F'(\text{ROBBER}) = \{d_3, d_6\}$
$F'(\text{LOVE}) = \{(d_3, d_4), (d_6, d_5)\}$

That is, this new model is like our very first model, except that it has two extra entities. Neither of these new entities has a name, but one of them is a customer, one of them is a robber, and the unnamed robber loves the unnamed customer. This *is* a perfectly good first-order model; there is no requirement that every entity in the model must have a name (intuitively, we only bother to name entities of special interest). Similarly, there is no requirement that each entity in a model must be named by at most one constant. For example, if we wanted to we could give some entity both the name Vincent and Honey Bunny (perhaps one of these names is a nickname).

However, while it is perfectly allowable to have unnamed entities, and entities with multiple names, when we define the simple model checker later in this chapter it will make matters a lot simpler if we don't have to worry about these possibilities. Call a model in which every entity in the domain of discourse is named by exactly one constant a *well-named model*. When we define our PROLOG model checker, we shall confine our attention to well-named models.

First-Order Languages

Given some vocabulary, we build *the first-order language over that vocabulary* out of the following ingredients:

1. All the symbols in the vocabulary. We call these symbols the *non-logical* symbols of the language.

2. A countably infinite collection of variables x, y, z, w, ..., and so on.

3. The Boolean operators $\neg$ (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 (and the comma. (These are essentially 'punctuation' symbols.)

Items 2–5 are common to all first-order languages. The only thing that distinguishes one first-order language from another is the choice of non-logical symbols, that is, the choice of vocabulary.

So, suppose we've chosen some vocabulary. How do we mix these five ingredients together? 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 analogs of proper names, and variables as first-order analogs of pronouns.)

We can then combine our 'noun phrases' with our 'predicates' (that is, the various relation symbols in the vocabulary) to form *atomic formulas*:

> 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 (or basic) formula.

Now that we know how to build atomic formulas, we can define how to build more complex ones. The following definition tells us exactly which *well formed formulas* (or *wffs*, or simply *formulas*) we can build.

1. All atomic formulas are wffs.

2. If ϕ and ψ are wffs then so are $\neg\phi$, $(\phi \rightarrow \psi)$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$.

3. If ϕ is a wff, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are wffs. (We call ϕ the *matrix* of such wffs.)

4. Nothing else is a wff.

The next thing we need to be clear about is the distinction between *free* and *bound* variables. To give an example, consider the following formula (over the vocabulary previously given):

$$\neg\,(\textsc{customer}(x) \vee \forall x(\textsc{robber}(x) \wedge \forall y\ \textsc{person}(y)))$$

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

A formula containing no free variables is called a *sentence*. The previous formula is not a sentence (because the first occurrence of x is free), however the following formula is:

$$\neg\,(\textsc{customer}(\textsc{vincent}) \vee \forall x\ (\textsc{robber}(x) \wedge \forall y\ \textsc{customer}(y)))$$

One final remark. We won't always strictly adhere to the 'official' first-order syntax defined above. In particular, we'll generally try and use as few brackets as possible, as this tends to improve readability. For example, we would rarely choose to write

$$(\textsc{customer}(\textsc{vincent}) \wedge \textsc{robber}(\textsc{pumpkin}))$$

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

$$\textsc{customer}(\textsc{vincent}) \wedge \textsc{robber}(\textsc{pumpkin})$$

To help further reduce the bracket count, we assume the following precedence conventions for the Boolean operators: $\neg$ binds more tightly than $\vee$ and $\wedge$, both of which in turn bind more tightly that $\rightarrow$. What this means, for example, is that the formula

$$\forall x\ (\neg\ \textsc{customer}(x) \wedge \textsc{robber}(x) \rightarrow \textsc{robber}(x))$$

is shorthand for the following:

$$\forall x\ ((\neg\ \textsc{customer}(x) \wedge \textsc{robber}(x)) \rightarrow \textsc{robber}(x))$$

In addition, we sometimes 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.

The Satisfaction Definition

We are now almost ready to present the key definition: what it means for a first-order sentence to be true in a model. There is, however, a difficulty. Although we are primarily interested in the relation called 'truth' which holds (or does not hold) between a sentence and a model, we cannot give a direct inductive definition of the 'truth' relation. This is for a very obvious reason: the matrix of a quantified sentence need not be a sentence! For example, $\forall$x ROBBER(x) is a sentence, but its matrix ROBBER(x) is not.

The solution to this problem is to proceed indirectly. We define 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 device which tell us 'what the free variables stand for'. By making use of assignment functions, we can define a natural notion of satisfaction for *arbitrary* formulas, and this will then make it easy to define the concept of truth for *sentences*.

So, suppose we've fixed our vocabulary. From now on, when we talk of a model $\mathbf{M}$, we mean a model of this vocabulary, and whenever we talk of formulas, we mean the formulas built from the symbols in that vocabulary. We are going to need two further technical definitions to state the satisfaction definition.

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. Then by the *interpretation* of τ with respect to $\mathbf{M}$ and g is meant $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a variable. We denote the interpretation of τ by $I^g(\tau)$.

The second idea 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.

With this technical terminology to hand, we come to the key definition. 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 $\mathbf{M}, g \models \phi$ (ϕ is satisfied in $\mathbf{M}$ with respect to the assignment of values to variables g) is defined inductively as follows:

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

(Here 'iff' is shorthand for 'if and only if'.) Note the crucial — and indeed, quite 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: $\exists \mathrm{x}\phi$ 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.

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

> A sentence 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 $\mathbf{M}, g \models \phi$.

We are now in a position to define two other crucial concepts: what *valid sentences* (or more simply, *validities*) are, and what a *valid argument* (or a *valid inference*) is.

A valid sentence is a sentence that is true in *all* models. (That is, it is impossible to find any model at all in which a validity is false.) For example, it is impossible to falsify

$$\textsc{robber}(\textsc{mia}) \vee \neg\textsc{robber}(\textsc{mia})$$

In any model, one (and indeed, only one) of the two disjuncts must be true, and hence the whole sentence must be true. It is also impossible to falsify

$$\forall \mathrm{x}(\textsc{robber}(\mathrm{x}) \rightarrow \textsc{customer}(\mathrm{x})) \wedge \textsc{robber}(\textsc{mia}) \rightarrow \textsc{customer}(\textsc{mia})$$

as the reader should check. We indicate that a sentence ϕ is valid by writing $\models \phi$.

What is a valid argument? Suppose $\phi_1, \ldots, \phi_n$, and ψ are a finite collection of first-order sentences. Then we say that the argument with *premisses* $\phi_1, \ldots, \phi_n$ and *conclusion* ψ is a *valid argument* if and only if whenever all the premisses are true in some model, the conclusion is true in that model also. (There are a number of other common ways of talking about this concept. For example, we sometimes say that ψ is a *valid inference* from the premisses $\phi_1, \ldots \phi_n$, or that ψ is a *logical consequence* of $\phi_1, \ldots \phi_n$.) The notation

$$\phi_1, \ldots, \phi_n \models \psi$$

means that the argument with premisses $\phi_1, \ldots, \phi_n$ and conclusion ψ is valid.

Here's a simple example. The argument with premisses $\forall \mathrm{x}(\textsc{robber}(\mathrm{x}) \rightarrow \textsc{customer}(\mathrm{x}))$ and $\textsc{robber}(\textsc{mia})$ and the conclusion $\textsc{customer}(\textsc{mia})$ is valid. That is,

$$\forall \mathrm{x}(\textsc{robber}(\mathrm{x}) \rightarrow \textsc{customer}(\mathrm{x})), \textsc{robber}(\textsc{mia}) \models \textsc{customer}(\textsc{mia})$$

That is, it is impossible to find a model in which the two premisses are true and the conclusion is false. As the reader may suspect, there is a connection between the validity of this argument and the validity of the sentence

$$\forall \mathrm{x}(\textsc{robber}(\mathrm{x}) \rightarrow \textsc{customer}(\mathrm{x})) \wedge \textsc{robber}(\textsc{mia}) \rightarrow \textsc{customer}(\textsc{mia})$$

The reader is asked to explore this connection in the exercises below.

The ideas of validity, and valid arguments, are fundamental semantic concepts and underly the idea of logical inference. In Chapters 3 and 4 we shall return to these concepts, and develop techniques for working with them computationally.

This concludes our review of the syntax and semantics of first-order logic. We finish with a global remark. As the reader has doubtless observed, most of the ideas we have discussed are fairly obvious — with one exception. The exception is the way variables are handled. This required a certain amount of care, and the introduction of new concepts such as assignments, and variant assignments. Unsurprisingly, when we turn to Prolog implementations of first order semantics, they key issue that that faces us is how to work with the variables.

Exercise 1.1.1 *We say that two sentences ϕ and ψ are* logically equivalent *if and only if $\phi \models \psi$ and $\psi \models \phi$. Show that $\forall \mathrm{x}\phi$ and $\neg\exists \mathrm{x}\neg\phi$ are logically equivalent.*

Exercise 1.1.2 *The* Deduction Theorem *for first order logic tells us that $\phi_1, \ldots, \phi_n \models \psi$ if and only if $\models (\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow \psi$. (That is, there is an intimate link between valid arguments and valid sentences.) Try and prove the Deduction Theorem.*

1.2 A simple model checker

We will now implement a simple first-order model checker in Prolog. The checker will take the Prolog representation of a model and the Prolog representation of a formula and test whether or not the formula is satisfied in the model. We will learn two principal lessons from this exercise:

1. By making use of PROLOG variables to represent first-order variables, it is very easy to implement such a model checker. This is because PROLOG's built in unification and backtracking mechanisms can take care — more or less automatically — of the process of substituting values for variables.

2. On the other hand, this simple implementation method also has its dangers. As we shall see, our 'obvious' implementation is not completely correct. We demonstrate some of the shortcomings of our implementation and show how to overcome them in the next section.

Let us consider how to implement such a model checker. We have four principal tasks to pursue: deciding how to represent vocabularies, deciding how to represent models, deciding how to represent formulas, and specifying how (representations of) formulas are to be evaluated in (representations of) models.

Representing vocabularies

Vocabularies will be represented as PROLOG databases. Let's consider an example. Suppose we are working with the following vocabulary:

{ (LOVE,2), (HATE,2),
(CUSTOMER,1), (ROBBER,1),
(MIA,0), (VINCENT,0),
(HONEY-BUNNY,0), (PUMPKIN,0) }

This would be represented by the following database:

```
relation(love,2).             constant(mia).
relation(hate,2).             constant(vincent).
relation(customer,1).         constant(honey_bunny).
relation(robber,1).           constant(pumpkin).
```

This should be self explanatory: the database simply lists the relation symbols in the vocabulary (together with their arity), and the constant symbols. Both relation symbols and constant symbols are represented by PROLOG atoms in the obvious way.

Representing models

Suppose we have fixed our vocabulary — for example, suppose we've decided to work with the vocabulary just given. How should we represent models of this vocabulary in PROLOG?

Actually, this is an impossibly difficult question. We can define models for this vocabulary given *any* non-empty set D whatsoever. In particular, we can do this for any *infinite* set D. Now, it is possible to give finite representations of some 'nicely structured' infinite sets — but most infinite sets are too large, or too unruly, to be worked with computationally. Hence we shall confine our attention to *finite* models of this vocabulary. In addition, as has already been mentioned, matters are much simpler if we only have to deal with *well-named* models. (Recall that a well-named model is a model over a vocabulary containing constants such that every element in the model is named by exactly one constant.) So, let's rephrase our question: how should we represent the finite well-named models over the above vocabulary?

Here is an example of how we shall do it:

```
[customer(vincent),customer(mia),
 robber(honey_bunny),robber(pumpkin),
 loves(pumpkin,honey_bunny)].
```

Fairly obviously, this list represents a model in which both Vincent and Mia are customers, both Pumpkin and Honey Bunny are robbers, and Pumpkin loves Honey Bunny. But it is important to be aware that other properties are *implicitly* recorded. For example, no facts about the *hate* relationships are given. This is PROLOG's way of recording the fact that the binary relation *hate* is empty; in this little world, nobody hates anybody. Moreover, although Pumpkin loves Honey Bunny, he doesn't love himself nor anybody else, for no such facts are recorded. In fact no other loving relationships at all hold. (Clearly, this isn't a suitable setting for a soap-opera!) Thus, by explicitly listing positive information, and 'implicitly listing' negative information, we have completely described a unique well-named model over this vocabulary.

Here's a second example, again over the same vocabulary.

```
[customer(vincent),
 robber(honey_bunny),robber(pumpkin),
 loves(pumpkin,honey_bunny)].
```

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 Vincent, Pumpkin and Honey Bunny are explicitly mentioned — you haven't properly grasped the role played by the constants listed in the vocabulary. In *any* model of this vocabulary there is an individual that corresponds to Mia. As it happens, in this particular model no positive facts are listed about Mia — but lots of negative facts are about Mia are implicitly given. For example, Mia is neither a customer nor a robber (perhaps Mia is a waitress), and Mia neither loves nor hates anyone or anything, and is not loved or hated by anyone or anything.

In short, be careful. The list of constants in the vocabulary plays an important role for us: it tells us exactly how many individuals there are in any well-named model of that vocabulary, namely one and only one for each constant. Indeed, we can think of the constants listed in the vocabulary as simply being the domain of quantification of well-named models. All the constants in the vocabulary play a role — even the ones not explicitly mentioned in some particular model.

Exercise 1.2.1 *Give the set theoretic description of the models that the two* PROLOG *terms given above represent.*

Exercise 1.2.2 *Suppose we are working with the following vocabulary:*

{ (WORKS-FOR,2),
(BOXER,1), (PROBLEM-SOLVER,1),
(THE-WOLF,0), (MARSELLUS,0), (BUTCH,0)}

First, represent this vocabulary as a PROLOG *database. Then, represent each of the following two (well-named) models (D, F) over this vocabulary as* PROLOG *terms.*

1. $D = \{d_1, d_2, d_3\}$,
   F(THE-WOLF) $= d_1$,
   F(MARSELLUS) $= d_2$,
   F(BUTCH) $= d_3$,
   F(BOXER) $= \{d_3\}$,
   F(PROBLEM-SOLVER) $= \{d_1\}$.

2. $D = \{$*entity-1*, *entity-2*, *entity-3*$\}$
   F(THE-WOLF) $=$ *entity-3*,
   F(MARSELLUS) $=$ *entity-1*,
   F(BUTCH) $=$ *entity-2*,
   F(BOXER) $= \{$*entity-2*, *entity-2*$\}$,
   F(PROBLEM-SOLVER) $= \{$*entity-2*$\}$,
   F(WORKS-FOR) $= \{($*entity-3*, *entity-1*$), ($*entity-2*, *entity-1*$)\}$.

Exercise 1.2.3 *Write a* PROLOG *program which when given a vocabulary, and a list, determines whether or not the list represents a (well-named) model over that vocabulary. Can your program be used to generate all the well-named models over that vocabulary? If not, write a program that does this.*

Representing formulas

Let us now decide how to represent first-order formulas in PROLOG. The first (and most fundamental) decision is how to represent first-order variables. We make the following choice:

> *First-order variables will be represented by* PROLOG *variables.*

As has already been mentioned, this decision will enable us to implement a model checker very simply — but will also give rise to a number of problems which we will deal with later.

Next, we must decide how to represent the non-logical 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 relation symbol R will be represented by the PROLOG atom `r`.

Given this convention, it is obvious how atomic formulas should be represented. For example, LOVE(VINCENT,MIA) would be represented by the PROLOG term `love(vincent,mia)`, and HATE(BUTCH,X) would be represented by `hate(butch,X)`. Note that first-order atomic formulas (for example BOXER(BUTCH)) are represented by exactly the same PROLOG atom (namely `boxer(butch)`) that is used to represent the fact that Butch is a boxer in our PROLOG representation of models. This will later enable us to implement the satisfaction clause for atomic formulas very simply.

Next the Booleans. The symbols

```
&        v        >        ~
```

will be used to represent the connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ respectively. The following PROLOG code ensure that these operators have their usual precedences:

```
:- op(40,yfx,>).       % implication
:- op(30,yfx,v).       % disjunction
:- op(30,yfx,&).       % conjunction
:- op(20, fy,~).       % negation
```

Finally, we must decide how to represent the quantifiers. Suppose *formula* is a first-order formula, and `Formula` is its representation as a PROLOG term. Then $\forall$x *formula* will be represented as

```
forall(X,Formula)
```

and $\exists$x *formula* will be represented as

```
exists(X,Formula)
```

Semantic evaluation

We now turn to the fourth and final task: evaluating (representations of) formulas in (representations of) models. The predicate which carries out the task is called `sem/2`, and the clauses of `sem/2` mirror the first-order satisfaction definition in a fairly natural way.

The clause which evaluates (representations of) atomic formulas is:

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

That is, an atomic formula is true if and only if that formula is one of the facts recorded in the list `Model`. Naturally, the `member/2` predicate used here is the the familiar one:

```
member(X,[X|_]).
member(X,[_|L]):-
     member(X,L).
```

Now for the clauses for Boolean combinations of formulas.

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

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

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

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

These mirror the first-order satisfaction definition in an obvious way. However note that in the clauses for `>` and `~` we have made use of PROLOG negation (that is, negation as failure). This is a simple and natural thing to do, but as we shall see in the next section, it can interact with our use of PROLOG variables to represent first-order variables in unexpected — and unintended — ways.

Finally come the key quantifier clauses. The one for the existential quantifier couldn't be simpler:

```
sem(exists(X,Formula),Model):-
   constant(X),
   sem(Formula,Model).
```

That is, the program unifies `X` to some entity in the model (recall that because we are working with well-named models, the entities we are talking about are essentially the constants) and then tries evaluating `Formula` in `Model`. If `Formula` is *not* satisfied in `Model` the program will backtrack and unify `X` to another entity (if this is still possible) and then evaluate the formula again. Either it eventually succeeds (which means that there exists an entity in the model does satisfy `Formula`), or it doesn't (which means there isn't). Clearly, this captures the semantics of the existential quantifier. Moreover it does it in a very simple way: PROLOG's unification and backtracking mechanisms are allowing us to simulate the assignment of values to first-order variables very simply.

At first glance, the clause for the universal quantifier may look a little strange — but recall that $\forall x \phi$ is logically equivalent to $\neg\exists x \neg\phi$. The following clause simply represents this equivalence in PROLOG (using negation as failure), and makes use of PROLOG's unification and backtracking mechanisms in much the same way as the clause for `exists` does.

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

So, it's time to start checking models. Here are some examples of models over our original vocabulary:

```
example(1,[customer(mia),customer(vincent),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).

example(2,[customer(mia),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).

example(3,[customer(mia),customer(vincent),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny),love(mia,vincent)]).
```

And here is a driver predicate which evaluates a formula on the desired example model:

```
evaluate(Formula,Example):-
   example(Example,Model),
   sem(Formula,Model).
```

Exercise 1.2.4 *Systematically test the model checker on these models. Are the results always what you expected? If not, why not?*

1.3 Some refinements

As the reader who has tested the model checker will have learnt, the current version of the model checker is far from satisfactory. In fact, it does quite a number of silly things. Let's try pinning down the various problems.

First problem

The first problem has to do with the way negation as failure interacts with our use of PROLOG variables to represent first-order variables.

Suppose we evaluate

```
customer(X)
```

in example 1. That is, we're asking whether it is possible to assign a value to the free variable that satisfies the formula, thus PROLOG's task is to find an entity in this model that is a customer. In example 1, of course, it finds two: Mia and Vincent.

Now suppose we evaluate

```
~ customer(X)
```

in example 1. That is, we're again asking whether it is possible to assign a value to the free variable that satisfies the formula, thus PROLOG's task is to find an entity in this model that is *not* a customer. In example 1, of course, there are two non-customers, namely Honey Bunny and Pumpkin.

However, PROLOG wrongly answers `no`. The reason is not hard to find. Because we defined the interpretation of negated clauses using negation as failure, then because `customer(X)` is satisfiable, PROLOG (incorrectly) decides that `~ customer(X)` is not satisfiable.

This is unacceptable and has to be fixed. But before doing so, let's look at a related problem.

Second problem

We can run into problems if we ask our model checker to verify formulas using completely new non-logical vocabulary. For example, suppose we try evaluating the atomic formula `tasty(royale_with_cheese)` in any of example models. Then our model checker will say `no`. This response is *not*, strictly speaking, correct. Formally the satisfaction relation is not defined between formulas and models of different vocabularies. The correct response of our model checker would be to throw out this formula and say something like 'Hey, I don't know anything about these symbols!'.

This example may not strike the reader as too problematic — but worse is in store. If we evaluate

```
~ tasty(royale_with_cheese)
```

in any of example models they will return the answer `yes`! This happens because we used negation as failure to define the evaluation clause for negated formulas. Clearly this needs fixing.

However the problem is not a particularly deep one. 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 relation symbols we are working with (and, of course, what their arities are), which constant we have, and explicitly state which combinations of these symbols are well formed. (Note that at present we're not making *any* use of the information stored in our vocabulary. It's hardly surprising that we're having difficulties.) Our driver should check that any sequence of symbols it has to evaluate really is a well formed combination of the chosen relation and constant symbols — and it should refuse to evaluate anything that isn't.

Let us now do this. We first define an predicate `wff/1` which checks whether a term represents a well formed formula built out of this primitive vocabulary. The clauses for non-atomic formulas are obvious:

```
wff(forall(X,Formula)):-
   var(X),
   wff(Formula).

wff(exists(X,Formula)):-
   var(X),
   wff(Formula).

wff(Formula1 > Formula2):-
   wff(Formula1),wff(Formula2).

wff(Formula1 & Formula2):-
   wff(Formula1),wff(Formula2).

wff(Formula1 v Formula2):-
```

```
   wff(Formula1),wff(Formula2).

wff(~ Formula):-
   wff(Formula).
```

The real work is done at the atomic level (`functor/3` and `arg/3` are PROLOG built-in predicates, see Appendix A):

```
wff(Formula):-
   functor(Formula,Symbol,Arity),
   relation(Symbol,Arity),
   good_arguments(Arity,Formula).

good_arguments(0,_).
good_arguments(N,Formula):-
   arg(N,Formula,Arg),
   first_order_term(Arg),
   M is N - 1,
   good_arguments(M,Formula).

first_order_term(X):-
   var(X).
first_order_term(X):-
   constant(X).
```

That is, at the base of the recursion we 'take the term apart'. We check that the functor really is one of our relations symbols, that it has the correct number of arguments, and so on.

It only remains to write a new driver which puts these predicates to work:

```
evaluate(Formula,Example):-
   wff(Formula),
   example(Example,Model),
   sem(Formula,Model).

evaluate(Formula,_):-
    \+ wff(Formula),
    nl,write('Not a wff over the given vocabulary.'),
    nl,write('Cannot be evaluated.'),nl.
```

It should be clear that this new version of `evaluate/2` fixes the second problem. However, it also fixes our first problem, and indeed, it also fixes another problem not discussed in the text. That is, making use of the information encoded in vocabularies tidies up a lot of the sillier problems in one fell swoop. It is important that the reader understands why this is; it is a good idea to work through the following two problems right away.

Exercise 1.3.1 *Try out the new version of* `evaluate/2` *on the query* `~ customer(x)`. *Why has problem 1 vanished?*

Exercise 1.3.2 *Suppose we continue to work with the same vocabulary and we add the model* `[]` *to our list of examples. That is, we add the model that contains no positive information over any of our four individuals. Does our first version of* `evaluate/2` *handle queries about this model correctly? Does the second version? Explain why.*

Third problem

Our new version of `evaluate/2` takes care of the sillier shortcomings of our first version, and we won't bother making any further improvements. Nonetheless, the reader should realise that our new version of `evaluate/2` is still not perfect. There is a third problem, due solely to our decision to use PROLOG variables to represent first-order variables.

Although more complex than the original version, `evaluate/2` is a fairly simple predicate. This is because all the real work of variable handling is being taken care of — more or less automatically — by PROLOG's in-built unification and backtracking mechanisms. But in a sense, we've cheated. Our program doesn't really understand how to treat free and bound variables at all.

An example should make this clear. In first-order logic the formulas

$$\exists \mathrm{x}\ (\textsc{customer}(\mathrm{x}) \wedge \exists \mathrm{y}\ \textsc{robber}(\mathrm{y}))$$

and

$$\exists \mathrm{x}\ (\textsc{customer}(\mathrm{x}) \wedge \exists \mathrm{x}\ \textsc{robber}(\mathrm{x}))$$

are equivalent. Now, their PROLOG representations are

```
exists(X,customer(X) & exists(Y,robber(Y)))
```

and

```
exists(X,customer(X) & exists(X,robber(X)))
```

respectively. However

```
?- evaluate(exists(X,customer(X) & exists(Y,robber(Y))),1).
```

succeeds (it returns all four of the successful ways of instantiating X and Y) while

```
?- evaluate(exists(X,customer(X) & exists(X,robber(X))),1).
```

returns `no`.

The difficulty is easy to spot. PROLOG doesn't know that the `X` in `customer(X)` is intended to play a different role than the `X` in `robber(X)`; it will always instantiate them to the same thing.

The moral is clear. We should either write a better version of `evaluate/2` which guards against this shortcoming (the reader is asked to do this as an exercise) or we should take care never to use the same variable in two different quantifiers.

Exercise 1.3.3 *Write a predicate which when given a formula, checks for 'reused' variables. Rewrite* `evaluate/2` *such that it filters out such sentences.*

1.4 Propositional logic, function symbols, and equality

To conclude this chapter we shall discuss an important sub-language of first order logic (namely, the quantifier free fragment, or propositional logic) and two extensions of first-order logic (namely, first-order logic with function symbols, and first order logic with equality). All three languages will play an important role later in the book.

Propositional logic

The quantifier-free fragment of any first-order language (as the terminology suggests) simply consists of all formulas of the language that contain no occurrences of the symbols $\exists$ or $\forall$. For example, ROBBER(PUMPKIN), CUSTOMER(MIA), CUSTOMER(y), and

$$\textsc{customer}(y) \rightarrow \textsc{loves}(\textsc{vincent},y)$$

are all quantifier-free formulas. On the other hand,

$$\forall y[\textsc{customer}(y) \rightarrow \textsc{loves}(\textsc{vincent},y)]$$

isn't, as it contains an occurrence of the quantifier $\forall$.

The key thing to note about quantifier-free formulas is the following. Suppose we are given a model **M** (of appropriate vocabulary) and an assignment g in **M**. Now, normally we need to work with two semantic notions: satisfaction for arbitrary formulas and truth for sentences. However, when working with quantifier-free formulas, there are no bound variables to complicate matters, so this distinction is unnecessary. In fact, when working with a quantifier-free fragment, we may as well view each variable x as a constant interpreted by $g(\mathrm{x})$. If we do this, then every quantifier-free formula is either true or false in **M** with respect to g. Moreover, it is obvious how to calculate the semantic value of complex sentences: conjunctions will be true if and only if both conjuncts are true, disjunctions will be true if and only if at least one disjunct is true, a negated formula will be true if and only if the formula itself is not true, and so on. (In short, we basically need to make the truth table calculations, which the reader is probably familiar with.)

This means we can simplify our notation somewhat. Because we don't have quantifiers, the internal structure of atomic formulas is irrelevant, for we're never going to bind any free variables they may contain. All that is important is whether the atomic symbols are true or false, and how they are joined together using Boolean operators. For example, while it may be mnemonically helpful to choose propositional symbols such as CUSTOMER(x) or LOVES(VINCENT,MIA), we lose nothing if we replace them by simpler symbols such as p and q. Following this line of thought leads us to define *propositional languages*. To specify a propositional language, we first say which symbols we are going to start with. (A fairly standard choice is p, q, r, s, t, and so on, often decorated with superscripts and subscripts: for example, p'', r''', or q_2.) The chosen symbols are called proposition symbols, or atomic sentences. Complex sentences are built up using the standard Boolean connectives (for example, $\neg$, $\wedge$, $\vee$ and $\rightarrow$) in the obvious way, and the truth values of complex sentences are calculated using the familiar truth table rules.

In short, propositional languages are essentially a simple notation for representing the quantifier-free formulas of first-order languages. When we devise inference mechanisms for first order logic in Chapters 4 and 5, it turns out to be sensible to first investigate inference methods for the quantifier free fragment (we do this in Chapter 4), and only then turn to the problem for the full first order language (the task of Chapter 5). Thus in Chapter 4, we will make use of the simpler propositional notation just defined.

Function symbols

Suppose we want to talk about Butch, Butch's father, Butch's grandfather, Butch's great grandfather, and so on. Now, if we know the names of all these people this is easy to do — but what if we don't? A natural way round this problem is to add a 1-place function symbol FATHER to the language. Then if BUTCH is the constant that names Butch, then FATHER(BUTCH) is a term that picks out Butch's father, FATHER(FATHER(BUTCH)) picks out Butch's grandfather, and so on. That is, function symbols are a syntactic device that lets us form recursively structured terms, thus letting us express many concepts (such as fatherhood) in a natural way.

Let's make this precise. First, we shall suppose that it is the task of the vocabulary to tell us which function symbols we have at our disposal, and what the arity of each of these symbols is. Second, given this information, we say (as before) that a model $\mathbf{M}$ is a pair (D, F) where F interprets the constants and relation symbols as described earlier, and, in addition, F assigns to each function symbol f an 'appropriate semantic entity'. What's an appropriate interpretation for an n-place function symbol? Simply a function that takes n-tuples of elements of D as input, and returns an an element of D-as output. Third, we need to say what terms we can form using these new symbols. The definition we require is the following:

1. All constants and variables are terms.
2. If f is a function symbols of arity n, and $\tau_1, \ldots, \tau_n$ are terms, then $f(\tau_1, \ldots, \tau_n)$ is a also a term.
3. Nothing else is a term.

A term is said to be *closed* if and only if it contains no variables.

Only one task remains: interpreting these new terms. In fact we need simply extend our earlier definition of I^g in the obvious way. Given a model $\mathbf{M}$ and an assignment g in $\mathbf{M}$, we define (as before) $I^g(\tau)$ to be $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a variable. On the other hand, if τ is a term of the form $f(\tau_1, \ldots, \tau_n)$, then we define $I^g(\tau)$ to be $F(f)(I^g(\tau_1), \ldots, I^g(\tau_n))$. (That is, we apply the n-place function $F(f)$ — the function interpreting f — to the interpretation of the n argument terms.)

Function symbols are certainly a natural extension to first order languages, as our earlier fatherhood example should suggest. However, they are also useful for more technical purposes. In particular, in this book they will play a key role in Chapter 5, where they will help us formulate a reasonably efficient inference system for first-order logic.

Equality

The first order languages we have so far defined have one very obvious expressive shortcoming: we have no way to assert that two terms denote the same entity. We will need to be able to express such identities in Chapter 7, when we link Discourse Representation Theory with first-order logic. So, what are we to do?

Actually, the solution is straightforward. Given any language of first order logic (with or without function symbols) we can turn it 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 rather ugly $=(\tau_1, \tau_2)$. Beyond this notational convention, there's nothing more to say about the syntax of $=$; it's just a a two place relation symbol. But what about it's semantics?

Here there is something interesting to say. Although, syntactically, $=$ is just a 2-place relation symbol, it is a very special one. In fact (unlike LOVES, or HATES, or any other two place relation symbol) we are *not* free to interpret it how we please. In fact, given any model $\mathbf{M}$, any assignment g in $\mathbf{M}$, and any terms τ_1 and τ_2, we shall insist that

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

That is, the the atomic formula $\tau_1 = \tau_2$ is satisfied if and only if τ_1 and τ_2 have exactly the same interpretation. That is, $=$ really means equality.

To put it another way, although syntactically just a relation symbol, $=$ is really a *logical* symbol on a par with $\neg$ or $\forall$. Just like these other symbols, it has a fixed interpretation, and a semantically fundamental one at that.

Notes

There are many good introductions to first-order logic, and perhaps the best advice we can give the reader is to spend some time browsing in a library to see what's on offer. That said, there's three references we particularly recommend. For a unhurried introduction that motivates the subject linguistically and philosophically, try the first volume of Gamut [Gam91a]. For a wide-ranging discursive overview, we recommend Hodges [Hod83]. This survey article covers a remarkable amount of ground — from truth tables to Lindström's celebrated characterisation of first-order logic. Parts of the article are quite technical, but, by and large, it's the sort of article that many readers will enjoy browsing through. On the other hand, the reader who wants a more focussed mathematically oriented approach, may prefer Enderton [End72]. This is a good, solid, and extremely readable introduction to first-order logic.

Chapter 2

Lambda Calculus

Now that we know something about first-order logic and how to work with it in PROLOG, it is time to turn to the first major theme of this book, namely:

> *How can we automate the process of associating semantic representations with expressions of natural language?*

In this chapter we explore this issue concretely. We proceed by trial and error. We first write a simple PROLOG program that performs this task in a very limited way. We note where it goes wrong, and why, and develop a more slightly more sophisticated alternative. These experiments lead us, swiftly and directly, to formulate a simple version of the *lambda calculus*. The lambda calculus is essentially a tool that steers the process of making substitutions. With its help, we will be able describe, neatly and concisely, exactly how semantic representations should be built. The lambda calculus is one of the main tools we make use of in this book, and by the end of this chapter the reader should have a reasonable grasp of why it is useful to computational semantics and how to work with it in PROLOG.

2.1 Compositionality

Given a sentence of English, is there a systematic way of constructing its semantic representation? This question is far too general, so let's ask a more specific one: is there a systematic way of translating simple sentences such as vincent likes mia and a woman snorts into first-order logic?

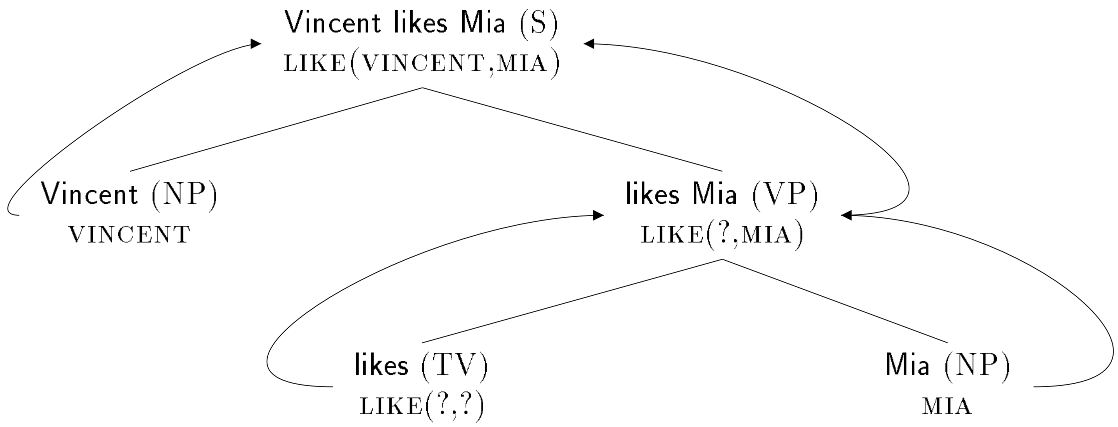
The key to answering this is to be more precise about what we mean by 'systematic'. Consider vincent likes mia. Its semantic content is at least partially captured by the first-order formula LIKE(VINCENT,MIA). Now, the most basic observation we can make about 'systematicity' is the following: the proper name vincent contributes the constant VINCENT to the representation, the transitive verb likes contributes the relation symbol LIKE, and mia contributes MIA.

This first observation is an important one, and is by no means as trivial as it may seem. If we generalise it to the claim that it is the words making up a sentence that contribute all the bits and pieces needed to build the sentence's semantic representation, we have formulated a principle that is a valuable guide to the complexities of natural language semantics. Certainly the principle is a plausible one. Moreover, it has the added advantage of forcing us to face up to a number of non-trivial issues sooner rather than later (for example, what exactly does the determiner every contribute to the representation of every woman loves a boxer?)

Nonetheless, though important, this principle doesn't tell us everything we need to know about 'systematicity'. For example, from the symbols LOVE, MIA and VINCENT we can also form LIKE(MIA,VINCENT). Why don't we get this (incorrect) representation when we translate vincent likes mia? What exactly is it about the sentence vincent likes mia that forces us to translate it as LIKE(VINCENT,MIA)? Note that the answer "But vincent likes mia means LIKE(VINCENT,MIA), stupid!", which in many circumstances would be quite appropriate, simply isn't particularly helpful here. Computers *are* stupid. We can't appeal to their semantic insight, because they don't have any. If we are to have any hope of automating semantic construction, we must find another kind of answer.

What is missing is a notion of *syntactic structure*. vincent likes mia isn't just a string of words: it has a hierarchical structure. In particular, vincent loves mia is an S (sentence) that is composed of the subject NP (noun phrase) vincent and the VP (verb phrase) loves mia. This VP is in turn composed of the TV (transitive verb) loves and the direct object NP mia. Given this hierarchy, it is easy to tell a coherent story — or indeed, to draw a convincing picture — about why we should get the representation LIKE(VINCENT,MIA), and not anything else:

(1)



Why is MIA in the second argument slot of LIKE? Because, 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, MIA) in the *second* argument slot of the VP's semantic representation (in this case, LIKE(?,?)). Why does VINCENT have to go in the *first* argument slot? Because this is the slot reserved for the semantic representations of NPs that we combine with VPs to form an S. More generally, given that we have some reasonable syntactic story about what the pieces of the 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. In short, one reasonable explication of 'systematicity' is that it amounts to using the additional information provided by syntactic structure to spell out exactly how the semantic contributions are to be plugged together.

In fact, our discussion has led us straight to one of the key concepts of modern semantic theory: *compositionality*. Suppose we have some sort of theory of syntactic structure. It doesn't matter too much what sort of theory it is, just so long as it is hierarchical in a way that ultimately leads to the lexical items. (That is, our notion of syntactic structure should allow us to classify the sentence into subparts, sub-subpart, and sub-sub-subparts, ..., and so on — ultimately into the individual words making up the sentence.) Such structures make it possible to tell an elegant story about where semantic representations come from. Ultimately, semantic information flows from the lexicon, thus each lexical item is associated with a representation. How is this information combined? By making use of the hierarchy provided by the syntactic analysis. Suppose, for example, that the syntax tells us that some kind of sentential subpart (a VP, say) is decomposable into two sub-subparts (say, a TV and an NP). Then our task it to describe how the semantic representation of the subpart is to be be built out of the representation of its two sub-subparts. If we succeed in doing this, we will have given what is called a *compositional semantics* for the language under discussion (or at least, for that fragment of the language covered by the syntactic analysis).

Compositionality is a simple and natural concept. It underlies most work in natural language semantics and the semantics of programming languages, and we shall take it for granted in this book. Nonetheless, the reader should be aware that in spite of its simplicity, it raises a number of interesting issues. For example, is every 'systematic' semantics a compositional one? In later versions of these notes we shall discuss the standard top-down algorithm for Discourse Representation Theory. This algorithm is a systematic approach to semantic construction (under any reasonable interpretation of the word 'systematic') but many semanticists have argued that it is *not* compositional. Other interesting issues concerning compositionality abound, and for further discussion the reader should consult the references cited in the Notes.

2.2 Syntax via DCGs

Let us return to our original question: is there a systematic way of translating simple sentences such as vincent likes mia and a woman snorts into first-order logic? We still don't know the answer, but at least we now have a plausible strategy for finding one. We need to:

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

Task 2 Specify semantic representations for the lexical items.

Task 3 Specify the translation compositionally. That is, we should specify the translation of all expressions in terms of the translation of their parts, where 'parts' refers to the substructure given to us by the syntax.

Moreover, all three tasks need to be carried out in a way that leads naturally to computational implementation.

As this is a book on computational semantics, tasks 2 and 3 are where are our real interests lie, and most of our energy will be devoted to them. But since compositional semantics presupposes syntax, we need a way of handling task 1. What should we do?

We have opted for the simplest possible solution. For us, the syntactic analysis of a sentence is simply a tree. This tree consists of nodes which stand for *complex syntactic categories* (such as S, NP and VP) and leaves that represent lexical items. Lexical items are associated with *basic syntactic categories* such as NOUN, TV, DET (determiner), PN (proper noun) and IV (intransitive verb). In short, the tree the reader has just seen is fairly typical of the syntactic structures used in this book. This approach has one obvious drawback (namely, the reader won't learn anything interesting about syntax from this book) but it has one overwhelming advantage: we will be able to make use of Definite Clause Grammars (DCGs), the in-built PROLOG mechanism for grammar specification.

Here is a DCG for the fragment of English we shall work with. (The reader unfamiliar with DCGs is advised to consult Appendix A right away, as otherwise this chapter will shortly stop making much sense.)

```
s --> np, vp.                    noun --> [woman].
np --> pn.                       noun --> [foot,massage].
np --> det, noun.                vp --> iv.
pn --> [vincent].                vp --> tv, np.
pn --> [mia].                    iv --> [walks].
det --> [a].                     tv --> [loves].
det --> [every].                 tv --> [likes].
```

This grammar tells us how to build certain kinds of sentences (`s`) out of proper names (`pn`), determiners (`det`), noun phrases (`np`), nouns (`noun`), intransitive verbs (`iv`), transitive verbs (`tv`), and verb phrases (`vp`). The grammar accepts a simple sentence like vincent walks

because vincent is declared as a proper name, and proper names are noun phrases according to this grammar, walks is an intransitive verb, which is in turn a verb phrase, and finally sentences can be formed by sequence of a noun and a verb phrase. Another simple sentence that this grammar finds okay is:

Mia likes a foot massage

But the real joy of DCGs is that they provide us with a with a lot more than a natural notation for specifying grammars. Because they are part and parcel of PROLOG, we can actually compute with them. For example, by posing the query

```
s([mia,likes,a,foot, message],[])
```

we can test whether this sentence belongs to the fragment the grammar describes. With a little more effort, we can do a lot more. In particular, by making use of extra arguments (again, see Appendix A, if you're uncertain about what is meant by this) we can associate semantic representations with lexical items very straightforwardly. The normal PROLOG unification mechanism then gives us the basic tool needed to combine semantic representations, and to pass them up towards sentence level. In short working with DCGs frees us from the bother of implementing parsers and mechanisms for combining representations. Instead, we'll be able to devote our attention to the really crucial issue: exactly how should semantic information be represented?

2.3 Two experiments

How can we compositionally associate first-order formulas with the sentences produced by our little grammar? Let's just plunge in and try, and see how far our knowledge of DCGs and PROLOG will take us.

Experiment 1

First the lexical items. We need to associate vincent with the constant VINCENT, mia with the constant MIA, 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 arity of `walk` and `love` are explicitly included as part of the PROLOG representation.

```
pn(vincent) --> [vincent].
pn(mia) --> [mia].
iv(snort(_)) --> [snorts].
tv(love(_,_)) --> [loves].
```

How do we build semantic representations for sentences? To keep things simple, let's first consider how to build representations for quantifier-free sentences such as mia loves vincent. The chief problem that faces us is to steer the constants into the correct slot of the relation symbol. (Remember, we want vincent loves mia to be represented by LOVE(VINCENT, MIA), not LOVE(MIA,VINCENT)!) Here's a first (rather naive) attempt. Let's directly encode the idea that when we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP in the second argument slot of the VP semantic representation, and that we have use the first argument slot for the semantic representations of NPs that we combine with VPs to form Ss.

Recall that PROLOG has a built in predicate `arg/3`, and that `arg(N,P,I)` is true if `I` is the `N`th argument of `P`. This is a useful tool for manipulating pieces of syntax, and with its help we can cope with simple quantifier free sentences rather easily. Here's the needed extension of the DCG:

```
s(Sem) --> np(SemNP), vp(Sem), { arg(1,Sem,SemNP) }.
np(Sem) --> pn(Sem).
vp(Sem) --> tv(Sem), np(SemNP), { arg(2,Sem,SemNP) }.
vp(Sem) --> iv(Sem).
```

These clauses work by adding an extra argument to the DCG (here, the position filled by the variables `Sem` and `SemNP`) to percolate up the required semantic information using PROLOG unification. Note that the second and the fourth clauses perform only this 'passing along' function, however the first and third clauses, which deal with branching rules, have a little more work to do: they use `arg/3` to steer the arguments into the correct slots. This is achieved by associating extra pieces of code with the DCG rules, namely `arg(1,Sem,SemNP)` and `arg(2,Sem,SemNP)`. (These are normal PROLOG goals, and are added to the DCG rules in curly brackets to make them distinguishable from the grammar symbols.) In short, this program captures, in a brutally direct way, the idea that the semantic contribution of the object NP goes into the second argument slot of TVs, while the semantic contribution of subject NPs belongs in the first argument slot.

It works. For example, if we pose the query:

```
?- s(Sem,[mia,snorts],[]).
```

we obtain the (correct) response:

```
Sem = snort(mia)
```

But this is far too easy — let's try to extend our fragment with the determiners a and every. First, we need to extend both the lexical entries for these words, and the entries for the common nouns they combine with:

```
det(exists(_,_&_)) --> [a].
det(forall(_,_>_)) --> [every].
noun(woman(_)) --> [woman].
noun(footmassage(_)) --> [foot,massage].
```

We sometimes call NPs formed by combining a DET with an N *quantified noun phrases*.

Next, we need to say how the semantic contributions of determiners and noun phrases should be combined. We can do this by using `arg/3` four times to get the instantiation of the different argument positions correct:

```
np(Sem) --> det(Sem), noun(SemNOUN),
            {arg(1,SemNOUN,X),arg(1,Sem,X),
             arg(2,Sem,Matrix),arg(1,Matrix,SemNOUN)}.
```

The key idea is that the representation associated with the NP will be the representation associated with the DET (note that the `Sem` variable is shared between `np` and `det`) — but with this representation 'fleshed out' with additional information from the NOUN. First, the Prolog variable `X` is the variable the determiner introduces into the semantic representation. The code `arg(1,SemNOUN,X)` and `arg(1,Sem,X)` unifies the argument place of the NOUN with the variable bound by the quantifier. (That is, the NP representation 'takes its variable from the NOUN'.) To understand what is achieved by `arg(2,Sem,Matrix),arg(1,Matrix,SemNOUN)`, it is best to consider an example. If we pose the query:

```
?- np(Sem,[a,woman],[]).
```

we obtain the response:

```
Sem = exists(X,woman(X)&Y)
```

That is, we don't yet fully know what matrix we are going to obtain (the variable `Y` has yet to be instantiated in a concrete formula) but at least we know we are quantifying over the set of women. (This piece of information, of course, is the contribution of the noun.)

If we think of NP semantics in this way, it is pretty clear what must happen when we combine a quantified NP with a VP to form an S: the VP must provide the missing piece of information. (That is, it must provide an instantiation for `Y`.) The following clause does this:

```
s(Sem) --> np(Sem), vp(SemVP),
           {arg(1,SemVP,X),arg(1,Sem,X),
            arg(2,Sem,Matrix),arg(2,Matrix,SemVP)}.
```

But wait a minute! Up till now, we've simply been extending the rules of our original DCG with semantic information — and we've already dealt with `s --> np, vp`. If we add this version of `s --> np, vp` (and it seems we need to) we are duplicating syntactic information. This doesn't seem a particularly economical (or elegant) way to proceed. Moreover, this second sentential rule interacts in an unintended way with the rule

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

As the reader should check, as well as assigning the correct semantic representation to a woman snorts, our DCG also assigns the splendidly useless string of symbols

```
snort(exist(X,woman(X)&Y))
```

But this isn't the end of our troubles. We already have a rule for forming VPs out of TVs and NPs, but we will need a second one to cope with quantified NPs, namely:

```
vp(Sem) --> tv(SemTV), np(Sem),
            {arg(2,SemTV,X),arg(1,Sem,X),
             arg(2,Sem,Matrix),arg(2,Matrix,SemTV)}.
```

If we add this rule, we can assign correct representations to all the sentences in our fragment. However we will also produce a lot of nonsense (for example, `A woman loves a foot massage` is assigned four 'readings', three of which are just jumbles of symbols) and we are being systematically forced into syntactically unmotivated duplication of rules. This doesn't look very promising. Let's try something else.

Experiment 2

Although our first experiment was ultimately unsuccessful, it did teach us something useful: to build representations, we need to work with 'incomplete formulas', and we need some way of indicating where the information they lack must go. This is especially clear if we consider the representations associated with determiners. For example, in experiment 1 we associated a with `exists(_,_&_)`. That is, this determiner contributes the 'skeleton' of a formula whose first slot needs to be instantiated with a variable, whose second slot needs to be filled with the semantic representation of an NOUN, and whose third slot needs to be filled by the semantic representation of a VP. In experiment 1, however, we didn't manipulate this missing information directly. Instead we took a shortcut: we thought in terms of argument position, so that we could make use of `arg/3`. Let's avoid plausible looking short cuts. The idea of missing information is evidently important, so let's take care always to associate it with an explicit PROLOG variable. This will give us a direct handle on it and will make semantic construction a lot easier.

Let's first apply this idea to the determiners. We shall need three extra arguments: one for the bound variable, one for the contribution made by the NOUN, and one for the contribution made by the VP. Incidentally, these last two contribution have a standard name: the contribution made by the NOUN is called the *restriction* and the contribution made by the VP is called the *nuclear scope*. We reflect this terminology in our choice of variable names:

```
det(X,Rest,Scope,exists(X,Rest & Scope)) --> [a].
det(X,Rest,Scope,forall(X,Rest > Scope)) --> [every].
```

But the same idea applies to common nouns, intransitive verbs, and transitive verbs too. For example, instead of associating woman with `woman(_)`, we should state that the translation of woman is WALK(y) for some particular choice of variable y — and we should *explicitly* keep track of which variable we choose. (That is, although the y appears free in WALK(y), we actually want to have some sort of hold on it.) Similarly, we want to associate a transitive verb like loves with LOVE(y,z) for some particular choice of variables y and z — and again, we should keep track of the choices we made. So the following lexical entries are called for:

```
noun(X,woman(X)) --> [woman].
iv(Y,snort(Y)) --> [snorts].
tv(Y,Z,love(Y,Z)) --> [loves].
```

Given these changes, we need to redefine the rules for producing sentences and verb phrases.

```
s(Sem) --> np(X,SemVP,Sem), vp(X,SemVP).
vp(X,Sem) --> tv(X,Y,SemTV), np(Y,SemTV,Sem).
vp(X,Sem) --> iv(X,Sem).
```

The semantic construction rule associated with the sentential rule, for example, tells us that `sem`, the semantic representation of the sentence, is essentially going to be that of the noun phrase (that's where the value of the `sem` variable will trickle up from) but that, in addition, the bound variable `X` used in `sem` must be the same as the variable used in the verb phrase semantic representation `SemVP`. Moreover, it tells us that `SemVP` is — somehow — to be used in forming the noun phrase semantics, since we have to unify this variable with the second 'missing information' variable in the noun phrase representation. What exactly is is involved here?

These, of course, is the key question. Experiment 1 failed because there was no obvious way of making use of the different semantic representations supplied by quantified noun phrases and proper names in a single sentential rule. Here we only have a single sentential rule, so evidently our NP representation works for both quantified noun phrases and proper names. To see how this is achieved, consider the two NP rules:

```
np(X,Scope,Sem) --> det(X,Rest,Scope,Sem), noun(X,Rest).
np(SemPN,Sem,Sem) --> pn(SemPN).
```

Note that we have given all noun phrases — regardless of whether they are quantifying phrases or proper names — the same arity in the grammar rules. (That is, there really is only *one* `np` predicate in this grammar, not two predicates that happen to make use of the same atom as their functor name.)

It should be pretty clear how the first rule works. The 'skeleton' of a quantified noun phrase is provided by the determiner. The restriction of this determiner is filled by the noun. The resultant noun phrase representation is thus a skeleton with two marked slots: `X` marks the bound variable, while `Scope` marks the missing scope information. It is this `Scope` variable that will be instantiated by the verb phrase representation when the sentential rule is applied.

The second rule is trickier. The vital work is performed by doubled `sem` variable. Roughly speaking, when we combine the noun phrase representations formed by this rule with a verb phrase, the effect of this doubling is to 'move the verb phrase representation rightwards' so that it occupies the slot that will ultimately be used in forming the sentential semantics. Intuitively, whereas the representation for sentences that have a quantified noun phrase as subject are essentially the subject's representation 'filled out' by the verb phrase representation, the reverse is the case when we have a proper name as subject. When that happens, the verb phrase is boss. The sentence representation is essentially the verb phrase representation, and the role of the proper name is simply to obediently instantiate the marked verb phrase slot. The 'rightwards shuffle' performed by the doubled variables is the mechanism which captures this role reversal in our grammar. It is important that the reader understands how this works, so don't skip the exercise at the end of the section!

Our second experiment has fared far better than our first. It is clearly a good idea to explicitly mark missing information; this gives us the control required to fill it in and maneuver it into place. Nonetheless, experiment 2 uses this idea rather clumsily. Much of the work is done by the rules. These state how semantic information is to be combined, and (as our NP rule show) this may require some programming tricks. Moreover, it is hard to think about the resulting grammar in a modular way. For example, when we explained why the NP rules took the form they did, we did so by explaining what was eventually going to happen when the S rule was used. Now, perhaps we weren't *forced* to do this, nonetheless, it is not straightforward to give an intuitive explanation of our rules on an individual basis.

This suggests we are missing something. Maybe a more disciplined approach to missing information would reduce — or even eliminate — the need for rule specific combination methods? Indeed, this is exactly happens if we make use of the discipline provided by the *lambda calculus*.

Exercise 2.3.1 *Using either pen and paper or a tracer, compare the sequence of variable instantiations this program performs when building representations for* vincent snorts *and* a woman snorts*, and* vincent loves mia *and* vincent loves a woman*.*

2.4 The Lambda Calculus

The version of the lambda calculus we shall work with is a notational extension of first order logic that allows us to bind variables using a new variable binding operator λ. Occurrences of variables bound by λ should be thought of as 'placeholders' for missing information. They explicitly mark where we should substitute the various bits and pieces obtained in the course of semantic construction. An operation called lambda conversion performs the required substitutions. We suggest that the reader think of the lambda calculus as a programming language dedicated to a single task: gluing together the items needed to build semantic representations.

The lambda operator marks missing information by binding variables. Here is a simple lambda expression:

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

Here the prefix λx binds the occurrence of x in $\textsc{man}(x)$. We sometimes say that the prefix λx *abstracts over* the variable x, and call expressions formed by prefixing λ together with some variable *lambda abstractions* (or, more simply, *abstractions*. In our example, the binding of the free x variable in $\textsc{man}(x)$ explicitly indicates that $\textsc{man}$ has an argument slot where we may perform substitutions. More generally, the purpose of abstracting over variables is to explicitly indicate where we want substitutions to be performed.

Concatenation is used to indicate that we wish to perform substitution. For example, consider the following expression.

$$(\lambda x.\textsc{man}(x))\textsc{vincent}$$

This compound lambda expression consists of the abstraction $\lambda x.\textsc{man}(x)$ written immediately to the left of the expression $\textsc{vincent}$. Such concatenations are called *functional applications*, and the left-hand expression is called the *functor*, and the right-hand expression the *argument*. Such a concatenation is an instruction to throw away the $\lambda x.$ prefix of the functor, and to replace every occurrence of x that it bound by the argument. We call this substitution process *lambda conversion* (other common names include β-reduction and β-conversion). Performing the lambda conversion demanded in the previous example yields:

$$\textsc{man}(\textsc{vincent}).$$

The two concepts just introduced underly much of our subsequent work in this book. The business of specifying semantic representations for lexical items is essentially going to boil down to devising a suitable lambda abstraction that correctly mark the missing information, while lambda conversion will be the engine used to compositionally combine semantic representations.

The above example was rather simple, and in one way rather misleading. As our previous experiments have made clear, to deal adequately with noun phrases and determiners (and indeed, many other things) we need to mark more complex kinds of information than that represented by individual variables. We shall use λ for this task as well. For example, our semantic representation of the noun phrase a woman will be:

$$\lambda Q.\exists x(\textsc{woman}(x) \wedge Q)$$

Here we are using the variable Q to indicate that some information is missing (namely, the nuclear scope, to use the linguistic terminology mentioned earlier) and where this information has to be plugged in when it is found (it will be the conjoined to $\textsc{woman}(x)$). In short, we're also going to make use of a second kind of variable (which we shall typically write using the capital letters P, Q, X, and Y) which we shall use as a place holder for complex information. We'll call these new variables *complex variables*.

We are almost ready to examine some linguistic examples, but first it's worth getting clear about the following point. The lambda expressions $\lambda x.\textsc{man}(x)$, $\lambda y.\textsc{man}(y)$, and $\lambda z.\textsc{man}(z)$ are all equivalent. All of them are functors which when applied to an argument, replace the bound variable by the argument. Which bound variable we use is irrelevant. No matter which argument $\mathcal{A}$ we choose, the result of applying any of these expressions to $\mathcal{A}$ and then lambda converting is $\textsc{man}(\mathcal{A})$.

More generally, if a lambda expression $\mathcal{E}$ can be obtained from a lambda expression $\mathcal{E}'$, and vice-versa, by relabeling bound variables, then we say that $\mathcal{E}$ and $\mathcal{E}'$ are *α-equivalent*. In what follows we often treat α-equivalent formula as if they were one and the same. For example, we will sometimes say that the lexical entry for some word is a lambda 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. Clearly it's alright to do this. What the reader needs to grasp, however, is that sometimes it's important that we work this way.

The reason has to do with lambda conversion. Suppose that the expression in $\mathcal{F}$ in $\lambda V.\mathcal{F}$ is a complex one that contains many λ operators. 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 by a lambda operator when we substitute it into $\mathcal{F}$. *We don't want this to happen.* Such accidental bindings (as they are usually called) defeat the whole purpose of working with the lambda calculus. The whole point of developing the lambda calculus was to gain control over the process of performing substitutions. We don't want to lose control by foolishly allowing unintended interactions.

Such interactions need never be a problem. For example, we don't need to use $\lambda V.\mathcal{F}$ as the functor; any α-equivalent formula will do. Thus, 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. In this book we always take care to choose our variables so that such clashes never arise.

That's pretty much all we need to know about the lambda calculus for now. Let's see how to put the lambda calculus to work in computational semantics.

As a first example, let's build the semantic representation for every boxer walks. As usual, the first step is to assign lambda expressions to the different basic syntactic categories. We assign the determiner every, the common noun boxer, and the intransitive verb walks the following lambda expressions.

every: $\lambda \mathrm{P}.\lambda \mathrm{Q}.\forall \mathrm{x}(\mathrm{P}(\mathrm{x}) \rightarrow \mathrm{Q}(\mathrm{x}))$

boxer: $\lambda \mathrm{y}.\textsc{boxer}(\mathrm{y})$

walks: $\lambda \mathrm{x}.\textsc{walk}(\mathrm{x})$

Before going a step further, pause a moment. These translations should remind the reader of something: the representations used in experiment 2. For example, in experiment 2 we gave the determiner every the representation

```
det(X,Rest,Scope,forall(X,Rest > Scope))
```

If we use the Prolog variable `P` instead of `Rest`, and `Q` instead of scope this becomes

```
det(X,P,Q,forall(X,P > Q))
```

which is clearly analogous $\lambda \mathrm{P}.\lambda \mathrm{Q}.\forall \mathrm{x}(\mathrm{P}(\mathrm{x}) \rightarrow \mathrm{Q}(\mathrm{x}))$. The difference, of course, is that the experiment 2 representation uses Prolog to explicitly mark the missing information. Now we are working with enriched representations — namely, lambda expressions — that can encode unaided everything we need to know about missing information. As we shall see, this difference turns out to be important.

Back to our example. 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 application that has the DET representation as functor and the NOUN representation as argument.

every boxer (NP)
$(\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x)))(\lambda y.\textsc{boxer}(y))$

every (DET)
$\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$

boxer (NOUN)
$\lambda y.\textsc{boxer}(y)$

Now, applications are instructions to carry out lambda conversion, so let's do what is required. (As there is no occurrence of y in the argument expression, there is no risk of accidental capture, so we don't need to relabel the bound variables in the functor.) Performing the demanded substitution yields:

every boxer: $\lambda Q.\forall x((\lambda y.\textsc{boxer}(y))(x) \rightarrow Q(x))$

But this expression contains a subexpression of the form $(\lambda y.\textsc{boxer}(y))(x)$. This is another instruction to perform lambda conversion, and when we do so we obtain:

every boxer: $\lambda Q.\forall x(\textsc{boxer}(x) \rightarrow Q(x))$

We can't perform any more lambda conversions, so lets carry on with the analysis of the sentence. The following tree shows the final representation we obtain:

every boxer walks (S)
$\forall x(\textsc{boxer}(x) \rightarrow \textsc{walk}(x))$

every boxer (NP)
$\lambda Q.\forall x(\textsc{boxer}(x) \rightarrow Q(x))$

walks (VP)
$\lambda x.\textsc{walk}(x)$

every (DET)
$\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$

boxer (NOUN)
$\lambda y.\textsc{boxer}(y)$

Why is the S node associated with $\forall x(\textsc{boxer}(x) \rightarrow \textsc{walk}(x))$? It's certainly what we want, but where did it come from?

In fact we obtain it by a procedure analogous to that just performed at the NP node. First, we associate the S node with the application that has the NP representation just obtained as functor, and the VP representation as argument:

every boxer walks: $(\lambda Q.\forall x(\textsc{boxer}(x) \rightarrow Q(x)))(\lambda x.\textsc{walk}(x))$

Note that $\lambda x.\textsc{walk}(x)$ doesn't contain any free variables, so we can carry out lambda conversion, without relabeling. Doing so yields

every boxer walks: $\forall x(\textsc{boxer}(x) \rightarrow (\lambda x.\textsc{walk}(x))(x))$

We can then perform lambda conversion on the subexpression $(\lambda x.\textsc{walk}(x))(x))$, and when we do so we obtain the desired representation:

every boxer walks: $\forall x(\textsc{boxer}(x) \rightarrow \textsc{walk}(x))$

It is worth reflecting on the previous example. In two important respects, our approach to semantic construction is getting simpler. First, the process of combining two representations we used was uniform. We simply needed to say which of representations was the functor and which the argument, whereupon combination could be carried out simply by applying functor to argument and lambda converting. Moreover, most of the real work of semantic analysis is now being borne by the lexicon: it is here that we use the lambda calculus to stipulate what information is missing.

Are there any clouds on the horizon? For example, while the semantic representation of a quantifying noun phrase such as a woman can be used as a functor, surely the semantic representation of an NP like vincent will have to be used as an argument? We avoided this problem in experiment 2 by the 'variable doubling' trick we used in one of the NP rules — but such an approach is incompatible with our desire to only use functional application and lambda conversion in combining representations. Maybe — horrible thought! — we're going to be forced to duplicate syntactic rules again, just as we did in experiment 1.

In fact, there's no problem at all! The lambda calculus offers us a delightfully simple functorial representation for proper names, as the following examples show:

mia: $\lambda P.P(\textsc{mia})$

vincent: $\lambda P.P(\textsc{vincent})$

These representations are abstractions, thus they can be used as functors. However note what such functors do. They are essentially instructions to substitute their argument in P, which amounts to apply their own arguments to themselves! Because the lambda calculus offers us the means to specify such role-reversing functors, the spectre of rule duplication vanishes.

As an example of these new representations in action, let us build a representation for vincent loves mia. First, we assign loves the following lambda expression:

$$\lambda X.\lambda \mathrm{z}.(X(\lambda \mathrm{x}.\textsc{love}(\mathrm{z},\mathrm{x})))$$

If we assume that TV semantic representations take their object NP's semantic representation as argument, and that subject NP semantic representations take the VP semantic representation as argument, we can build the following tree.

- Vincent loves Mia (S) — $\textsc{love}(\textsc{vincent},\textsc{mia})$
  - Vincent (NP) — $\lambda \mathrm{P}.\mathrm{P}(\textsc{vincent})$
  - loves Mia (VP) — $\lambda \mathrm{z}.(\textsc{love}(\mathrm{z},\textsc{mia}))$
    - loves (TV) — $\lambda X.\lambda \mathrm{z}.(X(\lambda \mathrm{x}.\textsc{love}(\mathrm{z},\mathrm{x})))$
    - Mia (NP) — $\lambda \mathrm{P}.\mathrm{P}(\textsc{mia})$

We suggest that the reader works through the functional applications and lambda conversions required to build the final representation, $\textsc{love}(\textsc{vincent},\textsc{mia})$.

Let's sum up what we have achieved. Our decision to move beyond the approach of experiment 2 to the more disciplined approach of the lambda calculus was sensible. For a start, we don't need to spend any more time thinking about how to combine two semantic representations — functional application and lambda conversion give us a general way of doing this. Moreover, most 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 now has a simple lexical solution. We don't need rule duplication or clever programming tricks: we just need to give a 'role reversing' semantic representation to proper names.

In fact, for the remainder of this book, we will be guided by the following version of the three tasks we listed earlier:

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

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

Task 3 Specify the translation of an item $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ with the help of functional application. That is, specify which part is to be thought of as functor (here it's $\mathcal{F}$), which as argument (here it's $\mathcal{A}$) and then let the resultant translation $\mathcal{R}'$ be $\mathcal{F}'(\mathcal{A}')$.

We must now show that these tasks lend themselves naturally to computational implementation.

2.5 The Lambda Calculus in PROLOG

In this section we show that by making "clever" use of PROLOG variables, it is almost trivial to implement lambda conversion. And, once lambda conversion has been implemented, defining the required semantic representations for our little DCG is very straightforward — certainly more straightforward than either of our earlier efforts. As the scare quotes indicate, however, our implementation of lambda conversion is not quite as clever as it may at first glance seem, but we'll defer till the next section a discussion of the problems it can lead to.

First, we have to decide how to represent lambda expressions in PROLOG. Something as simple as the following, where `L` is a lambda expression or variable, and `F` is a lambda expression or wff, will do:

```
lambda(L,F)
```

Second, we define what happens when we functionally apply an abstracted formula to an argument (that is, lambda conversion). In fact, with clever use of PROLOG unification we can do so with a single clause

```
fa(Functor,Argument,Result):-
    Functor=lambda(Argument,Result).
```

that can be simplified as:

```
fa(lambda(Argument,Result),Argument,Result).
```

The expression `lambda(Argument,Result)` is the functor, the variable `Argument` its argument, and `Result` is the result of the lambda conversion. That's all there is to it. To see why this works, consider the following example. Suppose we wanted to apply λx.WALK(x) to VINCENT. We would do this in PROLOG by means of the following query:

```
?- fa(lambda(X,walk(X)),vincent,Result).

Result = walk(vincent)
```

In short, the required substitutions are automatically carried out by the PROLOG's inbuilt unification mechanism.

In fact, `lambda` and `fa` are all we need to build semantic representations for the sentences produced by our little DCG. Let's deal with the rules first. Actually, there's practically nothing that needs to said here. If we work with rules in the manner suggested by (our new version of) task 3, all we have to do is add on `fa/3` goals to the original DCG rules.

```
s(S) --> np(NP), vp(VP), {fa(NP,VP,S)}.
np(NP) --> pn(NP).
np(NP) --> det(DET), noun(NOUN), {fa(DET,NOUN,NP)}.
vp(VP) --> iv(VP).
vp(VP) --> tv(TV), np(NP), {fa(TV,NP,VP)}.
```

Note that the unary branching rules just percolate up their semantic representation, while the binary branching rules use `fa/3` to build a semantic representations out of their component representations in the manner suggested by task 3.

The real work is done at the lexical level. Some of the lexical entries are very simple:

```
noun(lambda(X,footmassage(X))) --> [foot,massage].
noun(lambda(X,woman(X))) --> [woman].
iv(lambda(X,walk(X))) --> [walks].
```

Abstractions formed using complex variables need a different treatment. In their lexical entries we add `fa/3` goals as well. For example, the PROLOG code stating that λP.P(VINCENT) is the translation of vincent, and λP.P(MIA) the translation of mia, is the following.

```
pn(lambda(P,PN)) --> [vincent], {fa(P,vincent,PN)}.
pn(lambda(P,PN)) --> [mia], {fa(P,mia,PN)}.
```

As a second example, recall that the lambda expressions for the two determiners in our fragment are $\lambda P.\lambda Q.\forall x.(P(x) \rightarrow Q(x))$ and $\lambda P.\lambda Q.\exists x.(P(x) \land Q(x))$ respectively. These are expressed in PROLOG as follows.

```
det(lambda(P,lambda(Q,exists(X,R & S)))) --> [a],
                                              {fa(P,X,R),fa(Q,X,S)}.

det(lambda(P,lambda(Q,forall(X,R > S)))) --> [every],
                                               {fa(P,X,R),fa(Q,X,S)}.
```

Transitive verbs also require an additional `fa/3` goal in their representation. We leave the reader to formulate what is required as an exercise.

We have finally reached a stage where we have a simple uniform way of implementing semantic construction for our little DCG. Indeed, after all our work, it turns out all we need are two simple tools: the `fa/3` goals in the grammar rules and a `lambda/2` notation for the lexical items. Indeed, there is even a way to eliminate the `fa/3` goals and make exclusive use of `lambda/2`! This is because `fa/3` is a basic clause which can also be encoded directly in the rules. For example the grammar rule that makes sentences out of noun phrases and verb phrases can be simplified to:

```
s(S) --> np(lambda(VP,S)), vp(VP).
```

We suggest that you attempt the following exercises before proceeding further.

Exercise 2.5.1 *Give the* PROLOG *code for the lexical entry of the transitive verb* likes*.*

Exercise 2.5.2 *Give the* PROLOG *code for the lexical entry of the determiner* no*.*

Exercise 2.5.3 *Extend the DCG with a simple treatment of adjectives. Your grammar should be able to assign formulas to expressions like* a small man walks*.*

Exercise 2.5.4 *Extend the DCG so that it covers negated sentences such as* vincent does not walk *or* it is not the case that Vincent walks*.*

Exercise 2.5.5 *Eliminate all* `fa/3` *goals from the DCG.*

2.6 Problems

We decided to make "clever" use of PROLOG unification to implement lambda conversion. For some cases, it turns out that this wasn't really a good idea. One of these cases is *coordination* in natural language expressions. We will go through an example to make the problem clear.

Take an expression like vincent and mia dance. We know what the lambda expressions are for vincent, mia, and dance. For and we take the following lambda expression:

(2) $\lambda X.\ \lambda Y.\ \lambda \mathrm{P}.\ (X(\mathrm{P})\ \&\ Y(\mathrm{P}))$

Applying this lambda expression to those of mia and vincent respectively, we get as result the following:

(3) λP. ((λQ. Q(VINCENT))(P) & (λR. R(MIA))(P))

Applying lambda conversion twice yields:

(4) λP. (P(VINCENT) & P(MIA))

That is to say, the same lambda expression P is to be applied to both VINCENT and MIA. This is exactly what we want. In particular, applying this expression to DANCE yields the formula DANCE(VINCENT) & DANCE(MIA), which is the representation we want.

However, if we try to carry out this lambda conversion in PROLOG, we get no result. We will show how this problem can be solved in later versions of these notes. In the meantime, the reader might like to try the following exercises.

Exercise 2.6.1 *Explain why this case is problematic in* PROLOG. *Hint: unification.*

Exercise 2.6.2 *Try to cover coordination cases like* vincent walks and talks. *Does the same problem appear?*

2.7 But what does it all mean?

We have given an informal, computationally oriented, introduction to the lambda calculus and its applications in semantic construction. We adopted a rather procedural perspective, encouraging the reader to think of the lambda calculus as a programming language — indeed, as the sort of language that emerges when one tries to generalise straightforward logic programming approaches to semantic construction (such as that of experiment 2).

However our account hasn't discussed one interesting issue: what do lambda expressions actually mean? Hopefully the reader now has a pretty firm grasp of what one can *do* with lambda expressions — but is one forced to think of lambda expressions purely procedurally? As we are associating lambda expressions with expressions of natural language, it would be nice if we could give them some kind of model theoretic interpretation.

Actually, there's something even more basic we haven't done: we haven't been precise about what counts as a λ-expression! Moreover — as the industrious reader may already have observed — if one takes an 'anything goes' attitude, it is possible to form some pretty wild (and indeed, wildly pretty) expressions. For example, consider the following expression:

$$\lambda x.(xx)x$$

Is this a legitimate lambda expression? Suppose we functionally apply this expression to itself (after all, nothing we have said rules out self-application). That is:

$$(\lambda \mathrm{x}.(\mathrm{xx})\mathrm{x})(\lambda \mathrm{x}.(\mathrm{xx})\mathrm{x})$$

If we now apply lambda conversion we get:

$$((\lambda \mathrm{x}.(\mathrm{xx})\mathrm{x})(\lambda \mathrm{x}.(\mathrm{xx})\mathrm{x}))(\lambda \mathrm{x}.(\mathrm{xx})\mathrm{x})$$

But this is just the functional application followed by an additional occurrence of λx.(xx)x! Obviously we can going on applying lambda conversion as often as we like, producing ever longer expressions as we do so. Now, this is very interesting — but is it the sort of thing we had in mind when we decided to use the lambda calculus?

To conclude this chapter, we shall briefly discuss some of these issues. The main points we wish to make are that there are different version of the lambda calculus, useful for different purposes, and that both major variants of the lambda calculus can be given model theoretic interpretations in terms of functions and arguments. The reader interested in learning more is encouraged to follow up the references given in the Notes.

Lambda calculus comes in two main varieties: the *untyped* lambda calculus, and the *typed* lambda calculus. Both can be be regarded as programming languages, but they are very different.

The untyped lambda calculus adopts an 'anything goes' attitude to functional application. For example, λx.xxx is a perfectly reasonable expression in untyped lambda calculus, and it is fine to apply it to itself as we did above. The untyped lambda calculus is relevant to the study of natural language (explaining various natural language phenomena seems to demand some form of self-application). Moreover, it is relevant to functional programming (the core of the programming language LISP is essentially the untyped lambda calculus). Finally, it does have a model-theoretic semantics, indeed a very beautiful one. As one one might suspect, this semantics interprets abstractions as certain kinds of functions. The primary difficulty is to find suitable collections of functions in which the idea of self application can be captured. (In standard set theory, functions can't apply to themselves, so constructing function spaces with the structure necessary to model self application is a non-trivial exercise.) There are a variety of solutions to this problem, some of which are very elegant indeed.

There are many kinds of typed lambda calculi. The one we shall discuss is called the *simply typed lambda calculus*.

Natural language semanticists generally make use of some version of the simply typed lambda calculus. The key feature of the simply typed lambda calculus is that it adopts a very restrictive approach to functional application. "If it don't fit, don't force it", and typed systems have exacting notions about what fits. Let's discuss the idea of simple typing in a little more detail.

To build the kinds of representations we have been making use of in simply typed lambda calculus, we would proceed as follows. First we would specify the set of types. There would be infinitely many of these, namely (1) the type e of individuals, (2) the type t of truth values, and (3) for any types τ_1 and τ_2, the function type $\langle\tau_1, \tau_2\rangle$. Second, we would specify our logical language. This would contain all the familiar first order symbols, but in addition it would contain an infinite collection of variables of each type (the ordinary first order variables, which range over individuals, would now be thought of as the set of type e variables) together with the λ operator.

In the typed lambda caculus, every expression receives a *unique* type. The key clauses that ensures this are the definitions of abstraction and functional application. First, if $\mathcal{E}$ is a lambda expression of type τ_2, and v a variable of type τ_1 then λv.$\mathcal{E}$ is a lambda expression of type $\langle\tau_1, \tau_2\rangle$. In short, it matters which type of variable we abstract over. Abstracting with respect to different types of variables results in abstractions with different types. Second, if $\mathcal{E}$ is a lambda expression of type $\langle\tau_1, \tau_2\rangle$, and $\mathcal{E}'$ is a lambda expression of type τ_1 then we are permitted to functionally apply $\mathcal{E}$ to $\mathcal{E}'$. If we do this, the application has type τ_2. In short, we are only allowed to perform functional application when the types $\mathcal{E}$ and $\mathcal{E}'$ fit together correctly, *and only then.* The intuition is that the types tell us what the domain and range of each expression is, and if these don't match, application is not permitted. Note, moreover, that the type of the application is determined by the types of $\mathcal{E}$ and $\mathcal{E}'$. In effect, we have imposed a very strict type discipline on our formalism. The typed lambda calculus is a programming language, but when we use it we have to abide by very strict guidelines.

This has a number of consequences. For a start, self-application is impossible. (To use $\mathcal{E}$ as a functor, it must have a function type, say $\langle\tau_1, \tau_2\rangle$. But then its arguments must have type τ_1. So $\mathcal{E}$ can't be one of its own arguments.) Moreover, it is very straightforward to give a semantics to such systems. Given any model M, the denotation D_e of type e expressions are the elements of the model, the permitted denotations D_t of the type t expressions are TRUE and FALSE, and the permitted denotations $D_{\langle\tau_1,\tau_2\rangle}$ of type $\langle\tau_1, \tau_2\rangle$ expressions are functions whose domain is D_{τ_1} and whose range is D_{τ_2}. In short, expressions of the simply typed lambda calculus are interpreted using a straightforward, inductively defined, function hierarchy.

Which particular functions are actually used? Consider λx.MAN(x), where x is an ordinary first order variable. Now, MAN(x) is a formula, something that can be TRUE or FALSE, so this has type t. As was already mentioned, first-order variables are viewed as type e variables, hence it follows that the abstraction λx.MAN(x) has type $\langle e, t\rangle$. That is, it must be interpreted by a function from the set of individuals to the set of truth values. But which one? In fact, it would be interpreted by the function which, when given an individual from the domain of quantification as argument, returns TRUE if that individual is a man, and FALSE otherwise. To put it another way, it is interpreted using the function which exactly characterises the subset of the model consisting of all men. But this subset is precisely the subset that the standard first-order semantics uses to interpret MAN. In short, the 'functional interpretation' of lambda expressions is set up so that, via the mechanism of such *characteristic functions*, it meshes perfectly with the ordinary first order interpretation.

By building over this base in fairly straightforward way, the interpretation can be extended to cover all lambda expressions. For example, the expression λP.$\exists$x(WOMAN(x)$\wedge$P(x)) would be interpreted as a function which when given the type of function that P denotes as argument (and P denotes type $\langle e, t\rangle$ functions, like the characteristic function of MAN) returns a type t value (that is, either TRUE or FALSE). Admittedly, thinking in terms of functions that take other functions as arguments and return functions as values can get pretty tedious, but the basic idea is straightforward, and for applications in natural language semantics, only a very small part of the function hierarchy tends to be used.

In short, throughout this chapter we have been talking about the lambda calculus as a mechanism for marking 'missing information', and we have exploited the mechanisms of functional application and lambda conversion as a way of moving such missing information to where we want it. But in fact there is nothing at all mysterious about this 'missing information' metaphor. It is possible to give precise mathematical models of missing information in terms of functions and arguments. An abstraction is interpreted as a function, and the 'missing information' is simply the argument we will later supply it with. Indeed, a variety of models are possible, depending on whether one wants to work with typed or untyped versions of the lambda calculus.

Notes

Compositionality is a simple and natural idea — and one capable of arousing a surprising amount of passion and controversy. Traditionally attributed to Gottlob Frege (the formulation "the meaning of the whole is a function of the meaning of its parts" is often called Frege's principle) it received a precise mathematical formulation in the late 1960s. For detailed and accessible overview of the compositionality concept, the reader should consult Janssen [Jan97].

The idea of using the simply typed lambda calculus to specify the meanings of lexical entries, and using functional application as the basic mechanism for combining representations, is due to Richard Montague. Indeed, what we have presented in this chapter is simply a computational perspective on a very small part of Montague semantics.

Montague's original papers are still well worth reading (they are collected in Thomason [Tho74]) nonetheless they are densely written and most readers will be better off approaching Montague semantics via either Dowty, Wall and Peters [DDS81] or volume 2 of Gamut [Gam91b]. Both contain good, careful, textbook level expositions of Montague's key ideas. A more demanding (and correspondingly more rewarding) exposition is due to Janssen (see [Jan86a] and [Jan86b]). These are well worth reading, but rather hard to get hold of. If you see them, grab them. For a good overview of Montague's work, and an account of the major directions in which his work has been developed, Partee [Par97] is a must. This account is rich in historic detail, and is an excellent starting point for deeper forays into the semantic literature. For Montague-style and other approaches to semantics that make use of untyped calculus and related formalisms, see Chierchia, Partee and Turner [CGR89].

The mathematical and logical literature on the lambda calculus is huge. Here are some of the more obvious points of entry. The bible of untyped lambda calculus is Barendregt [Bar84]. Barendregt [Bar91] is a good starting point for material on typed systems. However these are very technical accounts, perhaps best suited for occasional reference. For a more approachable account of both typed and untyped systems, try Hindley and Seldin [HS86]. Turner [Tur97] is fairly technical, but usefully broad, and notes applications in linguistics and computer science. Finally, for logical work on the systems Montague himself used, and a number of interesting variants, Gallin [Gal75] is indispensable.

The references given so far approach lambda calculus and its applications in natural language semantics from a logical perspective. In this book we have tried to emphasize that there is a very natural computational perspective too. Not only is the lambda calculus a useful tool for gluing representations together, but the basic ideas emerges, with seeming inevitability, when one sits down and actually tries to do semantic construction in Prolog — or at least, that is what we have tried to suggest by approaching lambda calculus via experiments 1 and 2. We hasten to add that this 'seeming inevitability' is clear only with the benefit of hindsight. The links between the ideas of logic programming and Montague semantics were seem to have first been explicitly drawn in Pereira and Shieber [PS87].

Lambda calculus is probably the mostly widely used tool for semantic construction, and it is the tool of choice in this book — nonetheless, the reader should *not* run away with the idea that it is the only tool available. Indeed, it has a most interesting rival: the use of *feature structures* and *feature structure unification*. The reader who wants to go further in computational semantics really should be acquainted with this approach. Gazdar and Mellish [GM89] contains a good text book level introduction to its use in building syntactic and semantic representations, and indeed, on its uses in pragmatics too. Another interesting and readable paper is Moore [Moo89]. This paper compares the approach with the use of lambda calculus. More recently, linear logic has been proposed as a suitable glue language for semantic construction; see Dalrymple et al [DMV97].

Chapter 3

Quantifier Scoping

In this chapter we discuss quantifier scope ambiguities. These are ambiguities that arise in sentences containing more than one quantifying noun phrase. We implement three different algorithms that generate all possible semantic representations. We don't address the question of which representation is to be preferred.

We start with a very simple method based on Montague's approach to quantification, and develop it into more sophisticated storage techniques. You'll find the programs for this chapter in Appendix B.3 on page 174.

3.1 Quantifier Raising

The methods of the previous chapter allow us to assign semantic representations to sentences with transitive verbs and two quantifying noun phrases. The following example is typical:

(1)

- every boxer loves a woman (S) $\forall x(\textsc{boxer}(x) \rightarrow \exists y(\textsc{woman}(y) \wedge \textsc{love}(x,y)))$
  - every boxer (NP) $\lambda P.\forall x(\textsc{boxer}(x) \rightarrow P(x))$
  - loves a woman (VP) $\lambda z.\exists y(\textsc{woman}(y) \wedge \textsc{love}(z,y))$
    - loves (TV) $\lambda X.\lambda z.X(\lambda x.\textsc{love}(z,x))$
    - a woman (NP) $\lambda Q.\exists y(\textsc{woman}(y) \wedge Q(y))$

Now, the first-order formula constructed for this parse tree states that for each boxer there is a woman that he loves. There might be different women for different boxers. However, every boxer loves a woman has a second meaning (or to use the linguistic terminology, a second "reading") that is captured by the following formula:

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

This says that there is one (obviously very special) woman who is loved by all boxers. In short, every boxer loves a woman is ambiguous, and its ambiguity is due to the two scoping possibilities available to the quantifiers. In the reading assigned to (1), every boxer has scope over a woman, and in reading (2) it is the other way around.

The methods of Chapter 2 (and their Prolog implementation) only yield the first of these readings. The goal of this chapter is to extend our method of semantic construction so that it generates the semantic representations corresponding to all possible readings. We begin by examining Montague's approach.

Montague semantics makes use of the direct method of constructing semantic representations for quantified NPs studied in the previous chapter. However, motivated in part by quantifier scope ambiguities, Montague also introduced a rule of quantification (often called *quantifier raising*) that allowed a more indirect approach. The syntactic idea involved is very simple. Instead of directly combining syntactic entities with the quantifying noun phrase we are interested in, we are permitted to choose an 'indexed pronoun' and to combine the syntactic entity with the indexed pronoun instead. Intuitively, such indexed pronouns are 'placeholders' for the quantifying noun phrase. When this placeholder has moved high enough in the tree to give us the scoping we are interested in, we are permitted to replace it by the quantifying NP of interest.

As an example, let's consider how to analyse every boxer loves a woman. Here's the first part of the tree we need:

(3)

every boxer loves her-3 (S)
$\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_3))$

every boxer (NP)
$\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x))$

loves her-3 (VP)
$\lambda y.\textsc{love}(y,z_3)$

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

her-3 (NP)
$\lambda P.P(z_3)$

Instead of combining loves with the quantifying term a woman we have combined it with the placeholder pronoun her-3. This pronoun bears an 'index', namely the numeral '3'.

The placeholder pronoun is associated with a 'semantic placeholder', namely $\lambda P.P(z_3)$. As we shall see, it is the semantic placeholder that does most of the real work for us. Note that the pronoun's index appears as subscript on the free variable in the semantic placeholder. From a semantic perspective, choosing an indexed pronoun really amounts to opting to work with the semantic placeholder (instead of the semantics of the quantifying NP) and stipulating which free variable the semantic placeholder should contain.

Now, the key point the reader should note about this tree is how *ordinary* it is. True, it contains a weird looking pronoun her-3 — but, that aside, it's just the sort of structure we're used to. For a start, the various elements are syntactically combined in the expected way. Moreover semantic construction is also being performed completely standardly: the placeholder pronoun has the semantics $\lambda P.P(z_3)$ — which is the standard pronoun semantics — and (as the reader should check) the representations for love her-3 and every boxer loves her-3 are constructed using functional application just as we discussed in the previous chapter. In short, we have 'raised' both the syntactic and semantic placeholders high into the tree, and we have done so in a completely orthodox way.

Now for the next step. We want to ensure that a woman outscopes every boxer. By using the placeholder pronoun her-3, we have delayed introducing a woman into the tree. But every boxer is now firmly in place, so if we replaced her-3 by a woman we would have the desired scoping relation. Predictably, there is a rule that lets us do this: given a quantifying NP, and sentence containing a placeholder pronoun, we are allowed to construct a new sentence by substituting the quantifying NP for the placeholder. In short, we are allowed to extend the previous tree as follows:

(4)

every boxer loves a woman (S)

a woman (NP)	every boxer loves her-3 (S,3)
$\lambda P.\exists y(\textsc{woman}(y)\wedge P(y))$	$\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_3))$

But what's happening semantically? We know what formula we want to be assigned to the top S node (namely, the formula 2) but how can we ensure it gets there? Let's think the matter through.

We want a woman to take wide scope over every boxer semantically. Hence we should use the semantic representation associated with a woman as the function. (To see this, simply look at the form of its semantic representation. When we apply it to an argument and lambda convert, we will be left with an existentially quantified expression, which is what we want.) But what should its argument be? There is only one reasonable choice. It must be the representation associated with every boxer loves her-3 lambda abstracted with respect to z_3:

(5) $\lambda z_3.\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_3))$

Why is this? Well, right at the bottom of the tree we made use of the semantic placeholder $\lambda P.P(z_3)$. When we raised this placeholder up the tree using functional application, we were essentially 'recording' what the semantic representation of a woman would have encountered if we had used it directly. (Remember, we did nothing unusual, either sytactically or semantically, during the raising process.) The formula $\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_3))$ is the record of these encounters. When we are ready to 'playback' this recorded information, we lambda abstract with respect to z_3 (thus indicating that this variable is the crucial one, the one originally chosen) and feed the resulting expression as an argument to the semantic representation of a woman. Lambda conversion will glue this record into its rightful place, and, as the following tree shows, everything will work out just right:

(6)

every boxer loves a woman (S)
$\exists y(\textsc{woman}(y)\wedge\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,y)))$

a woman (NP)
$\lambda P.\exists y(\textsc{woman}(y)\wedge P(y))$

every boxer loves her-3 (S, 3)
$\lambda z_3.\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_3))$

That's it. Summing up, Montague's approach makes use of syntactic and semantic placeholders so that we can place quantifying NPs in parse trees at exactly the level required to obtain the desired scope relations. A neat piece of 'lambda programming' (we call it Montague's trick) ensures that the semantic information recorded by the placeholder is re-introduced into the semantic representation correctly.

Enough theory. How can we extend the previous chapter's Prolog implementation to integrate quantifier raising? Let's go through it systematically.

Most of the work takes place in the DCG rules. We add a new grammar rule for noun phrases that introduces the semantic representation of a pronoun (that is, we add a 'placeholder rule'). The pronoun will be associated with an index, and this index, together with the ordinary semantic representation of the noun phrase, will be passed to the mother node of the syntactic tree. (We do this by making use of additional arguments to the DCG rules.)

The following DCG rule implements these ideas. `Sem` is the semantics of the quantifying noun phrase, constructed in the usual way. The term `lambda(P,R)`, together with `fa(P,I,R)` where `I` is the index, mimics the usual pronoun semantics. (It's our semantic placeholder.) The first argument of `np` holds the list of raised quantifiers, together with their associated indexes. We call these elements *indexed binding operators*, and represent them as two-place terms of the form `bo(Sem,Index)`.

```
np([bo(Sem,I)],lambda(P,R)) --> det(SemDet), noun(SemNoun),
                                {fa(SemDet,SemNoun,Sem),fa(P,I,R)}.
```

Note that we have used a PROLOG list to store the indexed binding operators. This is because it makes it easy to introduce more indexes (if we have to) and straightforward to check whether any indexes were introduced at all.

The other clauses for noun phrases are essentially the ones we had in our original implementation, but we have to add an additional argument (namely, an empty list) to make them compatible with our new rule for quantifier raising.

```
np([],Sem) --> det(SemDet), noun(SemNoun),
               {fa(SemDet,SemNoun,Sem)}.

np([],Sem) --> pn(Sem).
```

Needless to say, the other rules in our grammar fragment need to be rewritten to take our extension into account. First we consider verb phrases. Intransitive verbs don't combine with NPs, so we simply give them the empty list as additional argument. Transitive verbs must pass on the list of raised quantifiers (coded as `Q`) that they get from the noun phrase:

```
vp([],Sem) --> iv(Sem).
vp(Q,SemVP) --> tv(SemTV), np(Q,SemNP), {fa(SemTV,SemNP,SemVP)}.
```

We need to be able to replace the placeholder pronouns by the real quantifiers. This happens at the sentential level, thus we need a rule saying that a sentence can be constructed out of sentence with a raised quantifier, by abstracting over the index, and applying the semantics of the raised quantifier to the resulting expression.

```
s([],Sem) --> s([(SemNP,I)],SemS), {fa(SemNP,lambda(I,SemS),Sem)}.
```

The standard rule for forming sentences remains unchanged, save we need (yet again) to add an extra argument and pass through the raised quantifiers.

```
s(Q,Sem) --> np([],SemNp), vp(Q,SemVp), {fa(SemNp,SemVp,Sem)}.
```

It is time to test our program, by posing the following query:

```
?- s([],Sem,[every,boxer,loves,a,woman],[]).
```

```
Sem = exists(X,woman(X)&forall(Y,boxer(Y)>love(Y,X))) ;

Sem = forall(Y,boxer(Y)>exists(X,woman(X)&love(Y,X))) ;

no.
```

It gives us exactly the two readings that we want.

What can we say about Montague's approach? It seems fair to say that while the deeper ideas underlying it are natural (namely, the use of semantic placeholders regulated by Montague's trick) the way quantifier raising makes use of these ideas isn't very appealing. For a start, we are placing a heavy burden on the grammar rules. In principal, grammar rules are there to tell us about syntactic structure — but now we're being forced to write additional rules that say something about scope ambiguity. Moreover, as will be apparent to readers who worked through our PROLOG code, the obvious implementation simply isn't very attractive. All that fiddling about with extra arguments — and just to cope with sentences containing a transitive verb and two quantifying noun phrases in subject and object position! Extending the program to deal with a wide range of scope ambiguity inducing constructions would not be pleasant.

So we abandon quantifier raising, and turn instead to *storage methods*. These neatly decouple scope considerations from syntactic issues, and minimise the effort involved in extending the grammar rules. Nonetheless, we certainly haven't seen the last of either semantic placeholders or Montague's trick; indeed, these ideas are fundamental to the work that follows.

Exercise 3.1.1 *Extend the quantifier raising analysis to ditransitive verbs, and check how many readings it gives for sentences like* a boxer gives every woman a foot massage. *Are these the readings you would expect?*

3.2 Cooper storage

The term *Cooper storage* refers to a technique developed by Robin Cooper for handling quantifier scope ambiguities. In contrast to the work of the previous section, semantic representations are built directly, without adding to the basic grammar rules. The key idea is to associate each node of a parse tree with a *store*, which contains a 'core' semantic representation together with the quantifiers associated with nodes lower in the tree. After the sentence is parsed, the store is used to generate scoped representations. The order in which the stored quantifiers are retrieved from the store and combined with the core representation determines the different scope assignments.

To put it another way, instead of simply associating nodes in parse trees with a single lambda expression (as we have done until now), we are going to associate them with a core semantic representation, together with the information required to turn this core into the kinds of representation we are familiar with. Viewed from this perspective, stores are simply a more abstract form of semantic representation — representations which encode, compactly and without commitment, the various scope possibilities.

Let's make these ideas precise. Formally, a store is an n-place sequence. We represent stores using the angle brackets $\langle$ and $\rangle$. The first item of the sequence is the core semantic representation; it's simply a lambda expression. (Incidentally, if we wanted to, we could insist that we've been using stores all along: we need merely say that when we previously talked of assigning a lambda expression ϕ to a node, we *really* meant that we assigned the 1-place store $\langle\phi\rangle$ to a node.) Subsequent elements (if any) are pairs (β, i), where β is the semantic representation of an NP (that is, another lambda expression) and i is an index. An index is simply a label which picks out a free variable in the core semantic representation. As we shall see, this index has the same purpose as the indexes we used for Montague-style raised quantifiers. We call pairs (β, i) *indexed binding operators*.

How do we use stores for semantic construction? Unsurprisingly, the story starts with the quantified noun phrases (that is, noun phrases formed with the help of a determiner). Instead of simply passing on the store assigned to them, quantified noun phrases are free to 're-package' the information it contains before doing so. (Other sorts of NPs, such as proper names, aren't allowed to do this.) More precisely, quantified noun phrases are free to make use of the following rule:

> **Storage**
> If the store $\langle\phi, (\beta, j), \ldots, (\beta, k)\rangle$ is a semantic representation for a quantified NP, then the store $\langle\lambda P.P(\mathrm{z}_i), (\phi, i), (\beta, j), \ldots, (\beta_k)\rangle)$, where i is some unique index, is also a representation for that NP.

The crucial thing to note is that the index associated with ϕ is identical with the subscript on the free variable in $\lambda P.P(\mathrm{z}_i)$. (If we decide to store ϕ away for later use, we need to keep track of what its argument is!)

In short, from now on when we encounter a quantified NP we will be faced by a choice. We can either pass on ϕ (together with any previously stored information) straight on up the tree, or we can decide to use $\lambda P.P(\mathrm{z}_i)$ as the core representation, store the quantifier ϕ on ice for later use (first taking care to record which variable it is associated with) and pass on this new package. The reader should be experiencing a certain feeling of *deja vu*. We're essentially using the pronoun representation $\lambda P.P(\mathrm{z}_i)$ as a semantic placeholder, just as we did in the previous section. Indeed, as will presently become clear, our shiny new storage technology re-uses the key ideas of quantifier storage in a fairly direct way.

Incidentally, the storage rule is *not* recursive. It offers a simple two way choice: either pass on the ordinary representation (that is, the store $\langle\phi, (\beta, j), \ldots, (\beta, k)\rangle$) or use the storage rule to form $\langle\lambda P.P(\mathrm{z}_i), (\phi, i), (\beta, j), \ldots, (\beta_k)\rangle)$ and pass this new store on up instead. We're *not* offered — and we don't want or need — the option of reapplying the storage rule to this new store to form $\langle \lambda P.P(z_m), (\lambda P.P(z_i), m), (\phi, i), (\beta, j), \ldots, (\beta_k)\rangle)$. Intuitively, we're offered a straight choice between keeping the lambda expression associated with the quantified NP in the active part of the memory (that is, in the first slot of the store) or placing it, suitably indexed, in the freezer for later consumption.

It's time for an example. Let's analyse every boxer loves a woman using Cooper storage. Here's (part of) the relevant tree.

(7)

every boxer loves a woman (S)
$\langle \textsc{love}(z_6,z_7),\ (\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x),6),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

every boxer (NP)
$\langle \lambda Q.Q(z_6),\ (\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x)),6)\rangle$

loves a woman (VP)
$\langle \lambda u.\textsc{love}(u,z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

loves (TV)
$\langle \lambda X.\lambda u.X(\lambda v.\textsc{love}(u,v))\rangle$

a woman (NP)
$\langle \lambda Q.Q(z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

Note that the nodes for a woman and a boxer are both associated with 2-place stores. Why is this? Consider the node for a woman. We know from our previous work that the lambda expression associated with a woman is $\lambda P.\exists y(\textsc{woman}(y)\wedge P(y))$. In our new representations-are-stores world view, this means that the 1-place store $\langle \lambda P.\exists y(\textsc{woman}(y)\wedge P(y))\rangle$ is a legitimate interpretation for the NP a woman. But remember — this is not our only option. We are free to use the storage rule, and this is what we did when building the above tree. We've picked a brand new free variable (namely z_7), used the placeholder $\lambda Q.Q(z_7)$ as the first item in the store, and 'iced' $\lambda P.\exists y(\textsc{woman}(y)\wedge P(y))$, first recording the fact that z_7 is the variable relevant to this expression. Essentially the same story could be told for the NP every boxer, save that there we chose the new free variable z_6.

Once this has been grasped, the rest is easy. In particular, if a functor node $\mathcal{F}$ is associated with a store $\langle \mathcal{F}', (\beta, j), \ldots, (\beta, k)\rangle$ and its argument node $\mathcal{A}$ is associated with the store $\langle \mathcal{A}', (\beta, l), \ldots, (\beta, m)\rangle$, then store associated with the node $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ is

$$\langle \mathcal{F}'(\mathcal{A}'), (\beta, j), \ldots, (\beta, k), (\beta, l), \ldots, (\beta, m)\rangle.$$

That is, the first slot of the store really is the active part: it's where the core representation is built. If you examine the above tree, you'll see that the stores associated with loves a woman and every boxer loves a woman were formed using this 'functional application in the first slot' method. Note that in both cases we've simplified $\mathcal{F}'(\mathcal{A}')$ using lambda conversion.

But we're not yet finished. We now have a sentence, and this sentence is associated with an abstract unscoped representation (that is, a store), but of course, at the end of the day we really want to get our hands on some ordinary scoped first-order representations. How do we do this?

This is is the task of *retrieval*, a rule which is applied to the stores associated with sentences. Retrieval removes one of the indexed binding operators from the freezer, and combines it with the core representation to form a new core representation. (If the freezer is empty, then the store associated with the S node must already be a 1-place sequence, and thus we already have the expression we are looking for.) It continues to do this until it has used up all the indexed binding operators. The last core representation obtained in this way will be the desired scoped semantic representation (we hope!).

What does the combination process involve? Suppose retrieval has removed a binding operator indexed by i. It lambda abstracts the core semantic representation with respect to the variable bearing the subscript i, and then functionally applies the newly retrieved binding operator to the newly lambda-abstracted-over core representation. The result is the new core representation, and it replaces the old core representation as the first item in the store. More precisely:

> **Retrieval**
> Let σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the store $\langle\phi, \sigma_1, (\beta, i), \sigma_2\rangle$ is associated with an expression of category S, then the store $\langle\beta(\lambda z_i.\phi), \sigma_1\sigma_2\rangle$ is also associated with this expression.

Hey — we're simply performing Montague's trick with the aid of stores!

Let's return to our example and apply the retrieval rule to the store associated with the S node. Now, this store contains two indexed binding operators. The retrieval rule allows us to remove either of them and to combine it with the core representation. Suppose we choose to first retrieve the quantifier for every boxer (that is, the indexed binding operator in the second slot of the store). Then the retrieval rule tells us that the following store must be associated with the S node:

(8) $\langle\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x)(\lambda z_6.\textsc{love}(z_6,z_7)),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

Using lambda conversion, this simplifies to:

(9) $\langle\forall x(\textsc{boxer}(x)\rightarrow\textsc{love}(x,z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

No more lambda conversions are possible, but there's still a quantifier left in store. Retrieving it produces:

(10) $\langle\lambda P.\exists y(\text{WOMAN}(y)\wedge P(y))(\lambda z_7.\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,z_7)))\rangle$

The result is the reading where a woman outscopes every boxer, as becomes clear if we perform two more lambda conversions to obtain:

(11) $\langle\exists y(\text{WOMAN}(y)\wedge\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,y)))\rangle$

How do we get the other reading? We simply retrieve the quantifiers in the other order. We suggest that the reader attempts the following exercise immediately.

Exercise 3.2.1 *Show the steps involved in applying retrieval that yield the reading where* every boxer *has scope over* a woman*.*

Let's now implement Cooper storage in PROLOG.

The first steps are pretty obvious: we'll represent stores as lists, and indexed binding operators as terms of the form `bo(Quant,Index)`, just as we did in the previous section. Thus the following PROLOG list represents a store:

```
[walk(X),bo(lambda(lambda(Y,F),forall,Y,boxer(Y) > F)),X)]
```

Now, since the semantic representations we want to work with are stores, not just plain lambda expressions, we have to re-implement functional application. As we've already discussed, when two stores are combined by functional application, we want the actual functional application to be performed on the core representations, and the remaining items in the two stores simply to be collected together. Hence we revise `fa/3` as follows:

```
fa([lambda(Arg,Result)|Store1],[Arg|Store2],[Result|Store]):-
     append(Store1,Store2,Store).
```

In short, the new predicate is basically the same as the old one. The real action takes place in the head of the list in the expected way, and the tails of the two input lists are stuck together using `append/3`.

What about the storage rule? Clearly this needs to be handled in the rules for constructing quantified noun phrases. More precisely, in the DCG rules that combine a determiner with a noun, we are going to need to have at our disposal a predicate that carries out storage rather that functional application. Here's something suitable:

```
np_storage([lambda(Arg,Res)|S1],[Arg|S2],[lambda(P,R),bo(Res,X)|S]):-
    append(S1,S2,S),fa([P],[X],[R]).
```

With the help of this predicate we can state the DCG rule for quantified noun phrases as follows:

```
np(Sem) --> det(SemDet), noun(SemNoun),
            {np_storage(SemDet,SemNoun,Sem)}.
```

One word of warning: this DCG makes storage *compulsory* not optional! Given our previous discussion, the reader will have been expecting the following disjunctive rule:

```
np(Sem) --> det(SemDet), noun(SemNoun),
              {
                np_storage(SemDet,SemNoun,Sem)
                ;
                fa(SemDet,SemNoun,Sem)
              }.
```

However, for our little grammar it is *irrelevant* whether storage is compulsory or optional. This is something we want the reader to think about in the exercises at the end of this section. (It's also a topic we'll be returning to in the following section.) So, for now, let's stick with the compulsory storage DCG rule given above.

Our little program does what we want. An sample query yields the following result:

```
?- s(Sem,[every,boxer,loves,a,woman],[]).

Sem = [love(Y,X),
        (lambda(lambda(A,P),forall(A,boxer(A)>P)),Y),
        (lambda(lambda(B,Q),exists(B,woman(B)&Q)),X)
      ] ;

no
```

In short, we've successfully built the unscoped abstract representation associated with the S node, so all that remains is to implement the retrieval rule.

The Prolog predicate that copes with retrieval is `s_retrieval/2`. Its first argument is the store, its second argument the derived scoped representation. If the store contains just one element, then this is the scoped formula; the first clause deals with this case. Otherwise, it takes an element from the store using the predicate `remove_from_store/3`, lambda abstracts `Sem` (the first item in the list) with respect to the retrieved variable, and applies the retrieved quantifier to it, yielding a new semantic representation. Note that the `s_retrieval` predicate is recursive, thus it will eventually reduce the store to a list containing just one item, namely a scoped representation.

```
s_retrieval([S],S).
s_retrieval([Sem|Store],S):-
   remove_from_store(bo(Q,X),Store,NewStore),
   fa([Q],[lambda(X,Sem)],[Result]),
   s_retrieval([Result|NewStore],S).
```

But why does this generate *all* the possible scoped representations? In fact, we obtain all the possible readings thanks to `remove_from_store/3`. This predicate is defined in such a way that, if there is more than one element in the store, it succeeds when it removes *any* element at all from it. Therefore, `s_retrieval` produces (via the PROLOG backtracking mechanism) all the scoped representation possible.

```
remove_from_store(X,[X|T],T).
remove_from_store(X,[Y|T],[Y|R]):-
   remove_from_store(X,T,R).
```

In our grammar fragment, we make use of the `s_retrieval` rule in the DCG rule for forming sentences. The code involved is transparent:

```
s(Sem) --> np(SemNp), vp(SemVp),
           {fa(SemNp,SemVp,SemS),s_retrieval(SemS,Sem)}.
```

Again, it is time to test our program:

```
?- s([Sem,[every,boxer,loves,a,woman],[]).

Sem = exists(X,woman(X)&forall(Y,boxer(Y)>love(Y,X))) ;

Sem = forall(Y,boxer(Y)>exists(X,woman(X)&love(Y,X))) ;

no
```

Unsurprisingly, these are exactly the results we want.

Summing up, Cooper storage is a generalization of Montague's rule of quantification that makes use of abstract representations called stores. From the perspective of computational semantics, it has a distinct advantage over Montague's approach: we don't need to to make use of additional grammar rules. Indeed, when you get down to it, all we really needed to do to implement Cooper storage was to to define two additional predicates, `np_storage` and `s_retrieval`, that gave us the ability to manipulate stores in ways ways that plain old functional application couldn't. The Cooper storage technique is indeed a useful one.

Exercise 3.2.2 *How many scoped representations are retrieved for* every piercing that is done with a needle is okay*? (Take* is done with *as a two place relation, and view* is okay *as a one place predicate.) Are they all correct?*

Exercise 3.2.3 *What does the store (before retrieval) for* every piercing is done with a needle *look like? And after retrieval?*

Exercise 3.2.4 *In the original discussion of Cooper storage the storage of quantifiers was an* optional *alternative to functional application. In our implementation, all quantified NPs were forced to use storage. Why is there no difference between the two strategies for the given fragment of grammar. Can you think of natural language examples where it does differ?*

3.3 Nested Cooper storage

Cooper storage allows us a great deal of freedom in retrieving information from the store. We are allowed to retrieve quantifiers in any order we like, and the only safety net provided is the use of co-indexed variables and Montague's trick.

Is this really safe? We haven't spotted any problems so far — but then we've only discussed one kind of scope ambiguity, namely those in sentence containing a transitive verb with quantifying NPs in subject and object position. However there are lots of other syntactic constructions that give rise to quantifier scope ambiguities, for instance relative clauses (12) and prepositional phrases in complex noun phrases (13):

(12) Every piercing that is done with a gun goes behind the entire idea of piercing

(13) Mia told a joke in every episode

Both examples give rise to scope ambiguities. For example, in (13) there is a reading where Mia told (possibly different) jokes in all episodes, and a reading where Mia told one and the same joke in all episodes. Moreover, both examples contain nested NPs. In the first example a gun is a sub-NP of every piercing that is done with a gun, while in the second, every episode is a sub-NP of a joke in every episode.

We've never had to deal with nested NPs before. Is Cooper storage delicate enough to cope with them, and does it allow us to generate all possible readings? Let's examine example (13) more closely and find out.

(14) $\langle \textsc{told}(\textsc{mia},z_2), (\lambda P.\exists y(\textsc{joke}(y)\wedge \textsc{in}(y,z_1)\wedge P(y)),2), (\lambda Q.\forall x(\textsc{episode}(x)\rightarrow Q(x)),1)\rangle$

Exercise 3.3.1 *Verify this.*

There are two ways to perform retrieval: by pulling the existential quantifier off the store before the universal, or vice versa. Let's explore the first possibility. Pulling the existential quantifier off the store yields (after lambda conversion):

(15) $\langle\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge\textsc{told}(\textsc{mia},y)), (\lambda Q.\forall x(\textsc{episode}(x)\rightarrow Q(x)),1)\rangle$

Retrieving the universal quantifier then yields (again, after lambda conversion):

(16) $\langle\forall x(\textsc{episode}(x)\rightarrow \exists y(\textsc{joke}(y)\wedge\textsc{in}(y,x)\wedge\textsc{told}(\textsc{mia},y)))\rangle$

This states that in every episode, some joke was told by Mia, possibly a different joke in different episodes. This is one of the readings we would like to have.

So let's explore the other option. If we pull the universal quantifier from the S store first we obtain:

(17) $\langle\forall x(\textsc{episode}(x)\rightarrow\textsc{told}(\textsc{mia},z_2)), (\lambda P.\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge P(y)),2)\rangle$

Pulling the remaining quantifier off the store then yields:

(18) $\langle\exists y(\textsc{joke}(y)\wedge\textsc{of}(y,z_1) \wedge \forall x(\textsc{episode}(x)\rightarrow\textsc{told}(\textsc{mia},y)))\rangle$

But this is not at all we wanted! Cooper storage has given us not a sentence, but a formula containing the free variable z_1. What is going wrong?

Essentially, the Cooper storage mechanism is ignoring the hierarchical structure of the NPs. The sub-NP every episode contributes the free variables z_1. However, this free variable does not stay in the core representation: when the NP a joke in every episode is processed, the variable z_1 is moved out of the core representation and put on ice. Hence lambda abstracting the core representation with respect to z_1 *isn't* guaranteed to take into account the contribution that z_1 makes — for z_1 makes its contribution indirectly, via the stored existential quantifier. Everything is alright if we retrieve this quantifier first (since this has the effect of 'restoring' z_1 to the core representation) but if we use the other retrieval option it all goes horribly askew. Cooper storage doesn't impose enough discipline on storage and retrieval, thus when it has to deal with nested NPs, it overgenerates.

What are we to do? An easy solution would be to build a 'free variable check' into the retrieval process. That is, we might insist that we can only retrieve an indexed binding operator if the variable matching the index occurs in the core representation.

But this isn't a very principled. It deals with the symptoms, not the cause. The heart of the problem is that Cooper storage rides roughshod over the hierarchical structure of NPs; we should try to deal with this head on. (Incidentally, arguably there's also an empirical problem: the free variable solution isn't adequate if one extends the grammar somewhat. We won't discuss this here but refer the reader to the Notes.)

Here's an elegant solution: allow nested stores. That is, allow stores to contain other stores. Intuitively, the nesting structure of the stores should automatically track the nesting structure of NPs in the appropriate way. As an added bonus, nesting is easier to implement than a free variable check.

Here's the new storage rule:

> **Storage**
> If the (nested) store $\langle \phi, \sigma \rangle$ is an interpretation for an NP, then the (nested) store $\langle \lambda P.P(z_i),(\langle \phi,\sigma \rangle,i)\rangle$, for some unique index i, is also an interpretation for this NP.

To see how this new storage rule works, consider how we assemble the representation associated with the complex noun phrase a joke in every episode.

a joke in every episode (NP)
$\langle \lambda P.P(z_2),\ (\langle \lambda P.\exists y(\textsc{joke}(y) \wedge \textsc{in}(y,z_1) \wedge P(y)),\ (\langle \lambda Q.\forall x(\textsc{episode}(x) \rightarrow Q(x))\rangle,1)\rangle,2)\rangle$

a (DET)
$\langle \lambda Q.\lambda P.\exists y(Q(y) \wedge P(y))\rangle$

joke in every episode (CN)
$\langle \lambda u.\textsc{joke}(u) \wedge \textsc{in}(u,z_1),\ (\langle \lambda Q.\forall x(\textsc{episode}(x) \rightarrow Q(x))\rangle,1)\rangle$

joke (CN)
$\langle \lambda x.\textsc{joke}(x)\rangle$

in every episode (PP)
$\langle \lambda P.\lambda u.P(u) \wedge \textsc{in}(u,z_1),\ (\langle \lambda Q.\forall x(\textsc{episode}(x) \rightarrow Q(x))\rangle,1)\rangle$

As for the retrieval rule, it will now looks like this.

> **Retrieval**
> Let σ, σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the (nested) store $\langle \phi, \sigma_1, (\langle \beta, \sigma \rangle, i), \sigma_2 \rangle$ is an interpretation for an expression of category S, then $\langle \beta(\lambda z_i.\phi), \sigma_1 \sigma \sigma_2 \rangle$ is too.

The new retrieval rule ensures that any operators stored while processing β become accessible for retrieval only after β itself has been retrieved. Nesting automatically overcomes the problem of generating readings with free variables. To see how it works in practice, let's return to our original example. The nested store associated with mia told a joke in every episode is

$$\langle \textsc{told}(\textsc{mia},z_2),\ (\langle \lambda P.\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge P(y)),\ (\langle \lambda Q.\forall x(\textsc{episode}(x)\rightarrow Q(x))\rangle,1)\rangle,2)\rangle$$

There is only one way to perform retrieval: to first pull of the universal quantifier, followed by the existential quantifier, resulting in (16). Since this is the only possibility the unwanted reading (18) is not generated.

A few rather trivial changes are all that's needed to turn our Cooper storage program into a nested Cooper storage program. Here's the new version of np_storage/3 we need. It's identical to the previous version, save for the third argument in the head, which now treats stores as lists of lists:

```
np_storage([lambda(Argument,Result)|Store1],
           [Argument|Store2],
           [lambda(P,R),bo([Result|Store],X)]):-
    append(Store1,Store2,Store),
    fa([P],[X],[R]).
```

The only other change is in the first clause of remove_from_store/3, to enable it to deal with lists of lists:

```
remove_from_store(bo(O,I),[bo([O|T1],I)|T2],T):-
    append(T1,T2,T).
```

And that's it. The overgeneration problem is solved. We'll never see unwanted free variables again.

But wait a minute: how do we get the reading where Mia told one and the same joke in all episodes? In fact, this couldn't be easier. All we have to do is avoid storing the sub-NP every episode. If we do this, we can produce the following tree:

(19)

a joke in every episode (NP)
$\langle \lambda P.P(z_2),\ (\langle \lambda P.\exists y(\textsc{joke}(y)\wedge\forall x(\textsc{episode}(x)\rightarrow\textsc{in}(z,x))\wedge P(y))\rangle,2)\rangle$

a (DET)
$\langle \lambda Q.\lambda P.\exists x(Q(x)\wedge P(x))\rangle$

joke in every episode (CN)
$\langle \lambda z.\textsc{joke}(z)\wedge\forall x(\textsc{episode}(x)\rightarrow\textsc{in}(z,x))\rangle$

joke (CN)
$\langle \lambda x.\textsc{joke}(x)\rangle$

in every episode (PP)
$\langle \lambda P.\lambda z.P(z)\wedge\forall x(\textsc{episode}(x)\rightarrow\textsc{in}(z,x))\rangle$

This leads to the following analysis for mia told a joke in every episode:

(20) $\langle$TOLD(MIA,z_2),($\langle\lambda$P.$\exists$y(JOKE(y)$\wedge\forall$x(EPISODE(x)$\rightarrow$IN(y,x))$\wedge$P(y))$\rangle$,2)$\rangle$

There is only one operator the store. Retrieving it yields the reading we want.

(21) $\langle\exists$y(JOKE(y)$\wedge\forall$x(EPISODE(x)$\rightarrow$IN(y,x))$\wedge$TOLD(MIA,y))$\rangle$

So our earlier PROLOG implementation of Cooper storage — which made storage compulsory — really was linguistically naive! While making storage compulsory didn't have any malign effects in the previous section, this was only because we were working with a very restricted grammar fragment. As the previous example shows, as soon as we move to a richer fragments in which nested NPs are possible, storage really does need to be optional to ensure that all possible readings are generated.

We've already know how to incorporate this optionality into the PROLOG program, namely by making use of the following disjunctive rule.

```
np(Sem) --> det(SemDet), noun(SemNoun),
              {
                np_storage(SemDet,SemNoun,Sem)
                ;
                fa(SemDet,SemNoun,Sem)
              }.
```

Note, however, that this extension brings a small piece of computational unpleasantness in its wake. Let's test our program and see what happens:

```
?- s(Sem,[mia,told,a,joke,in,every,episode],[]).

Sem = forall(Y,episode(Y)>exists(X,(joke(X)&in(X,Y))&told(mia,X))) ;

Sem = forall(Y,episode(Y)>exists(X,(joke(X)&in(X,Y))&told(mia,X))) ;

Sem = exists(X,(joke(X)&forall(Y,episode(Y)>in(X,Y)))&told(mia,X)) ;

Sem = exists(X,(joke(X)&forall(Y,episode(Y)>in(X,Y)))&told(mia,X)) ;

no
```

The two correct results are there — but both of them twice. This is side-effect of backtracking through the disjunctive predicate. But there is a little trick we can apply here to overcome this problem. This is to gather all solutions with the PROLOG built-in predicate `bagof/3`, and then remove all duplicate items in our list of solutions. (Since we can have different variables for the same solutions, the built-in predicate `setof/3` doesn't help us much.) We do this as follows:

```
mysetof(X,Goal,Solutions):-
   bagof(X,Goal,Temp),
   remove_duplicates(Temp,Solutions).

remove_duplicates([],[]).
remove_duplicates([X|L1],L2):-
   member(X,L1),!,
   remove_duplicates(L1,L2).
remove_duplicates([X|L1],[X|L2]):-
   remove_duplicates(L1,L2).
```

Having added this code, we can pose a query using `mysetof/3` and obtain the list of results that we wish to have:

```
?- mysetof(Sem,s(Sem,[mia,told,a,joke,in,every,episode],[]),L).

L = [forall(Y,episode(Y)>exists(X,joke(X)&of(X,Y)&told(mia,X))),
     exists(U,joke(U)&forall(V,episode(V)>of(U,V))&told(mia,U))]
```

Let's sum up what we have learned. First, the original version of Cooper storage doesn't handle storage and retrieval in a sufficiently disciplined way, and this causes it to generate spurious readings (in fact, nonsensical readings) when faced with nested NPs. The problem can be elegantly cured by making use of nested stores, and implementing this idea requires only trivial changes to our earlier PROLOG code. Second, storage really must be optional. If we are to generate all the required readings in examples involving nested NPs, we must be free not to store the NPs encountered low down in the hierarchy. A little care is required in implementing optionality if we want to avoid generating one reading several times.

We conclude this chapter with some general remarks. Consider the example that we started the discussion with, every boxer loves a woman. In a sense, representation (1) is actually sufficient to cover both readings of our example, since it is entailed by the other reading (2). Arguably, that makes the stronger representation superfluous: perhaps the stronger reading was inferred from the weaker one with the help of contextual knowledge. So do we really need techniques for coping with quantifier ambiguity?

We do. There are examples of quantifier scope ambiguities giving rise to logically independent reading. Another, independent, reason is that direct construction won't always lead to the weakest reading (that is, the reading entailed by the other possible readings).

There are many important issues to do with quantifier scoping which we have ignored. One such issue has to do with logical redundancies: our algorithm will generate two logically equivalent readings for every boxer loves every woman. A second issue concerns the incorporation of world knowledge to exclude possible reading. For example, given a sentence like a fist of every boxer, our 'generate all possibility' methods will happily generate readings in which a fist is shared by several boxers, which is biologically implausible.

Incidentally, scope ambiguities are not only introduced by quantifiers, but also by operators like negation. In the sentence vincent doesn't clean every car there is a reading where for each car it is not the case that Vincent doesn't clean it. A second, possibly preferred, reading is one where there are some cars (not all) that Vincent doesn't clean.

Exercise 3.3.2 *Give the PL translations for the five readings of* a man likes every woman with a five-dollar shake*. You might have noticed that the number of correct readings for this example is less than the combinatorial possibilities of the quantifiers that are involved. Naively, one would expect to have 3! = 6 readings for this example. Why is one reading excluded?*

Notes

More about quantifier scoping in general, and Montague's approach to quantification in particular, can be found in Dowty, Wall and Peters [DDS81] or volume 2 of Gamut [Gam91b]. Cooper storage was first used in Cooper [Coo75]; a more refined version is presented in Cooper [Coo83]. Nested Cooper storage is due to Keller ([Kel88]). This very readable paper introduces Cooper storage, explains where the problems lie, gives an example (involving an interaction between scope and anaphoric pronouns) which suggests that a simple check for free variables during retrieval isn't an adequate solution, and then introduces nested Cooper storage.

There are other approaches to quantifier scope ambiguity the reader should be aware of. Hobbs and Shieber [HS87] provide an alternative to Cooper storage that overcomes some of the problems mentioned in this chapter. An improved version of their algorithm has been developed by Lewin [Lew90].

As we mentioned in the text, stores (and nested stores) are essentially a more abstract form of semantic representation — representations which encode multiple possibilities, without committing us to any particular choice. Viewed in this light, stores are ancestors of the current crop of *underspecification formalisms*, representation languages specifically designed to cope with ambiguity by avoiding overcommitment. Probably the best way for the reader to find out more about this area is to consult the recent collection edited by van Deemter and Peters [vDP96].

Chapter 4

Propositional Inference

In this chapter we turn to the second major theme of the book, namely:

> *How can we use the logical representations of natural language expressions to automate the process of drawing inferences?*

Recall from Chapter 1 that a valid formula is a formula that cannot be falsified in any model, and that a valid argument (or valid inference) is an argument such that whenever all the premisses are true in some model, the conclusion is true in that model also. These concepts are the fundamental ones underlying the idea of logical inference, and so we need a way of getting to grips with them computationally. But there is a problem. Both concepts are semantic: that is, they are defined in terms of models. Moreover, they use models in a very strong way: their definitions refer to the class of *all* models. Now, the class of all models is a very large and abstract entity. Certainly we cannot put all models in the computer (there are infinitely many of them, most which are infinite) and check whether an argument is valid by checking its premisses and conclusions in them all. So, if we want programs that perform logical inference, we will have to look for another approach.

Fortunately, alternatives are available. The branch of logic called *proof theory* studies *syntactically* defined inference methods. That is, proof theorists are interested in inference methods which only make use of the syntactic structure of sentences; models play no role. Of course, proof theoretic methods must always be *justifiable* in semantic terms. That is, if someone asks why some proposed syntactic proof method really is a way of performing genuine logical inference — not just some bizarre way of manipulating logical symbols — it must be possible to show that the method is faithful to the fundamental semantic concepts of validity. Nonetheless, *using* proof-theoretic methods requires no appeal to semantic concepts; only syntactic manipulation of formulas is required. Since we are interested in automating inference, it is clearly sensible to try and make use of such techniques, for syntax is pleasantly concrete — it's something that a computer can easily get to grips with.

Proof theorists have studied many different proof methods. Among the better known ones are axiomatic systems, natural deduction systems, resolution-based systems, sequent calculi, and the method we shall discuss here: *tableaux systems*. The various proof methods were developed for different purposes, and have different strengths and weaknesses. For example, axiomatic systems are excellent if one wants to study provability at an abstract level, but awful to actually use. Natural deduction systems, on the other hand, are nice and easy for people to use (as their name suggests, they try to capture something of what is involved when a human being goes about proving something), but, precisely because they demand human insight, are not the best choice for automated theorem proving. Resolution, on the other hand, has proved to be the key method for automated theorem proving (PROLOG is based on it), but it is a 'machine oriented' rather than a 'human oriented' approach. We have chosen to work with tableaux systems because they offer the following blend of advantages:

1. Tableaux are a very intuitive proof method. The tableaux method is certainly syntactic, nonetheless it based on utterly transparent semantic intuitions — indeed, tableaux are often called *semantic tableaux*. They are certainly a 'human oriented' approach to inference.

2. Although intuitively natural, the process of finding a tableaux proof does not *depend* on human insight. Certainly human insight can streamline the process of performing tableaux inferences; nonetheless, the basic mechanism is suitable for machine implementation. Indeed, tableaux proof systems are relatively simple to implement, and, resolution aside, are one of the more popular automated theorem proving methods.

3. The tableaux method can be adapted to many different logics, such as higher-order and non-classical logics. The basic ideas are relatively robust, and the reader who masters them in the case of first-order logic has actually mastered ideas of far wider applicability.

4. Tableaux systems are more than just theorem provers: they can also be regarded as *model building tools*. (This is another reason why they are sometimes called semantic tableaux.) For some applications this additional ability is useful.

In this chapter we are going to develop the tableaux method for propositional languages. Recall from Chapter 1 that propositional languages are essentially a user-friendly notation for the quantifier-free fragment of first order languages. It will turn out that this is a very important fragment to examine, for in the next chapter we shall see how to efficiently reduce the tableaux method for a full first-order language to the tableaux method for its propositional fragment.

4.1 The basic ideas

The key intuition underlying the tableaux proof technique revolves around the following *semantic* question:

> *Suppose we are given a formula, and one of the truth values* TRUE *or* FALSE. *Is it possible to find a model in which the given formula has the given truth value?*

The tableaux method is essentially a *syntactic* way of answering this question. More precisely, a tableaux proof is a *systematic* check, making use of only syntactic concepts, which lets us know whether or not it is possible to build a model in which some given formula is true or false.

Suppose that we had such a systematic check at our disposal. (We soon will have.) Why would it be useful? First, note that it would give us a way to test formulas for validity: 'valid' means 'true in all models', so a formula is valid if and only if it is not possible to falsify it in any model. Hence, a formula ϕ would be valid iff the systematic method told us that there was no way to build a model that falsified ϕ.

Second, note that such a method would also give us a way of performing logical inference. For suppose we are given $\phi_1, \ldots, \phi_n$ are premisses, and we wish to judge whether the argument to the conclusion ψ is valid or not. That is, we wish to know whether or not

$$\phi_1, \ldots, \phi_n \models \psi.$$

Now, by the Deduction Theorem (discussed in Chapter 1), this argument is valid precisely when

$$\models \phi_1 \wedge \cdots \wedge \phi_n \rightarrow \psi,$$

that is, when $\phi_1 \wedge \cdots \wedge \phi_n \rightarrow \psi$ is a valid formula. In short, the problem of deciding whether arguments are valid is reducible to the problem of deciding whether formulas are valid. So, given a method of determining whether a formula is valid, we can use it for judging the validity of arguments as well.

Thus such a systematic check for the existence of satisfying or falsifying models would certainly be worth having, and this is exactly what tableaux systems give us. But what is a tableaux system, and how do they systematically check for the existence of suitable models? In the remainder of this section we informally introduce the key ideas involved. We do this by presenting three examples of tableaux proofs, taking care to point out the underlying semantic intuitions. In the following section we will make our discussion more precise.

Consider the formula $p \vee \neg p$. This is a validity — we certainly can't falsify it — but what would a systematic search for a falsification looks like? Now, we've already seen one answer to this question — simply fill out its truth table! But this answer isn't very appealing. For a start, it's not an answer we can generalise to first-order languages. Moreover, while a truth table is easy to fill out for $p \vee \neg p$, the truth table for a formula containing 8 different propositional variables contains 256 lines, while the table for a formula containing 20 variables contains 2^{20} lines, which is too large for comfort. So we won't bother with truth tables — instead, we'll develop a number of *tableaux expansion rules*. These rules tell us how to make complex formulas true (or false) by breaking them down into their component formulas and giving the components the appropriate truth value. Let's see what sort of rules are need, and how to use them, by constructing a tableaux proof of $p \vee \neg p$.

We want to try and falsify $p \vee \neg p$, so let's introduce a piece of notation to express this goal. Writing $F\phi$, where ϕ is any formula, will mean that we want to make ϕ false. Similarly, writing $T\phi$ will mean that we want to make ϕ true. (T and F are called *signs*, and a formula preceded by a sign is called a *signed formula*.) Thus, as we are going to try to falsify $p \vee \neg p$, our initial goal is:

$$F(p \vee \neg p).$$

This rather trivial looking object is our first example of a tableau. Actually, when writing out tableaux by hand, it is handy to include a little extra book-keeping information, such as line numbers. So, in practice we'd tend to write the above tableau as:

$$1 \quad F(p \vee \neg p)$$

How do we proceed? Essentially we use the tableaux expansion rules to smash the signed formula into smaller and smaller pieces until we reach the atomic level. So, what expansion rule should we apply here? Obviously we need a rule that tells us how to *falsify a disjunction*. The required rule is clear: to make a disjunction false, falsify both disjuncts. So, applying this rule — let's call it $F_{\vee}$ — we expand our one line tableau to a three line tableau as follows:

$$\begin{array}{lcl} 1 & F(p \vee \neg p) & \surd \\ 2 & Fp & 1, F_{\vee} \\ 3 & F\neg p & 1, F_{\vee} \end{array}$$

Here, lines 2 and 3 are the extra information we have deduced by applying the $F_{\vee}$ rule. The third column contains more book-keeping information: the $\surd$ symbol in line 1 records the fact that we've applied the appropriate rule to line 1, while the annotations '$1, F_{\vee}$' in lines 2 and 3 record the fact that these lines were obtained from line 1 using rule $F_{\vee}$.

What next? In fact, there's only one more thing we can do. We've already applied a rule to line 1, so we've finished with that. Furthermore, line 2 tells us something about *atomic* information (namely that we need to make p false). This is simply a blunt fact, not something that can be further analysed. Thus only line 3 offers us the possibility of further progress; it tells us that we need to falsify the negation of p. So, we need an expansion rule that tells us how to do this. Again, the required rule is clear: to make the negation of a formula is false, make the formula itself true. So, applying this rule — call it $F_\neg$ — we can expand our three line tableau to a four line tableau as follows:

$$\begin{array}{lcl} 1 & F(p \vee \neg p) & \surd \\ 2 & Fp & 1, F_\vee \\ 3 & F\neg p & 1, F_\vee, \surd \\ 4 & Tp & 3, F_\neg. \end{array}$$

Note that we have marked line 3 with a $\surd$ (thus recording that the applicable rule has been applied) and have indicated that line 4 was obtained from line 3 using $F_\neg$.

There are two important observations that must be made about this tableau. First, it is *rule saturated*. That is, we cannot expand it any more. Either we've already applied the applicable rule (lines 1 and 3), or the line contains instructions about what to do with atomic information (lines 2 and 4). Second, the tableau is *closed*. That is, it contains contradictory instructions. The tableau tells us that if we want to falsify $p \vee \neg p$, we have to make p false (line 2) and p true (line 4). As this is impossible, and as it should be pretty clear that the above tableau really does indicate all possible ways falsifying $p \vee \neg p$, we conclude that this formula is valid. This closed tableau is called a *tableaux proof* of $p \vee \neg p$.

The previous example is a perfectly good tableaux proof, but it's also just about the simplest one the reader is ever likely to see. So let's consider a slightly more demanding task: testing $\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$ for validity.

Just as in the previous example, our initial goal is to to try and falsify the given formula. Thus our initial tableau is:

$$1 \quad F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$$

We need to falsify an implication, so we need an expansion rule that tells us how to do this. Again, the required rule is clear: to falsify an implication, make the antecedent true and the consequent false. Applying this rule — let's call it $F_\rightarrow$ — yields the following tableau:

$$\begin{array}{lcl} 1 & F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r) & \surd \\ 2 & T\neg(q \wedge r) & 1, F_\rightarrow \\ 3 & F(\neg q \vee \neg r) & 1, F_\rightarrow \end{array}$$

Now, line 3 demands that we falsify a disjunction. We've already met the required expansion rule, namely $F_\vee$. Applying it to line 3 yields:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}$
5	$F\neg r$	$3, F_{\vee}$

Now at this point the alert reader should be saying "Hold on a minute! Why are we free to apply this rule to line 3? Sure, the rule fits — but look at line 2. There's a formula there that we need to make true. Don't we need to take care of that first?"

A sensible question, but the answer is: *no, we don't*. One of the pleasant things about tableaux is that we are free to apply applicable rules in any order we like. Yes, we could have applied the relevant rule to line 2 at this point, had we wanted to — but we're equally free to apply a rule to line 3, just as we did above. Tableaux rules just tell us what we're *permitted* to do to expand a tableau; we're not forced to apply expansion rules in any particular order.

So let's move on. Note that both lines 4 and 5 ask us to falsify a negated formula. Again, we have already met the relevant rule, namely $F_{\neg}$. Applying it first to line 4, and then to line 5 (a completely arbitrary ordering choice) yields:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}, \surd$
5	$F\neg r$	$3, F_{\vee}, \surd$
6	Tq	$4, F_{\neg}$
7	Tr	$5, F_{\neg}$

Let's now deal with line 2. (In fact, there is nothing else we can do.) We need to make a negation true. The required rule is clear: to make the negation of a formula true, make the formula itself false. So, applying this rule — call it $T_{\neg}$ — we can expand our tableau to obtain:

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}, \surd$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}, \surd$
5	$F\neg r$	$3, F_{\vee}, \surd$
6	Tq	$4, F_{\neg}$
7	Tr	$5, F_{\neg}$
8	$F(q \wedge r)$	$2, T_{\neg}$

Now we have to deal with line 8, which asks us to falsify a conjunction. Doing so leads us to the first real complication in the story we have been telling. The point is this: there's not just one way of making a conjunction false, there's two. Making either conjunct false will falsify the whole formula. As tableaux are meant to be *systematic* searchs for falsifications, we're going to have to examine both possibilities. Thus the relevant expansion rule — call it $F_\wedge$ — is going to yield two alternative ways of expanding the tableau, and we're going to have to keep track of both.

$F_\wedge$ is our first example of a *disjunctive* (or *branching*) expansion rule; we'll encounter more such rules in the following section. Because of such rules, tableaux *won't* generally consist of a single straight line down the page (that is, they're no longer going to to consist of a single *branch*, to use the official terminology). Rather, they will be tree-like structures, possibly containing many branches. In the present case, what we get after applying $F_\wedge$ is:

1	$F\neg(q \wedge r) \rightarrow (\neg q \wedge \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_\rightarrow, \surd$
3	$F(\neg q \wedge \neg r)$	$1, F_\rightarrow, \surd$
4	$F\neg q$	$3, F_\vee, \surd$
5	$F\neg r$	$3, F_\vee, \surd$
6	Tq	$4, F_\neg$
7	Tr	$5, F_\neg$
8	$F(q \wedge r)$	$2, T_\neg, \surd$

9 Fq $8, F_\wedge$ 10 Fr $8, F_\wedge$

That is, we have recorded two distinct possibilities: we must either falsify q, or falsify r.

But let's return to our tableaux proof. What do we do next? Actually, we've finished: our two-branch tableau is rule-saturated, as the reader can easily check. So, after all that, is $\neg(q \wedge r) \rightarrow (\neg q \wedge \neg r)$ a validity or not? Yes, it is. This is because the rule-saturated tableau we produced is *closed*, which means that *all the branches it contains are closed.* To see this, note that the left-hand branch is closed because it contains the contradictory instructions Fq (at line 9) and Tq (at line 6); whereas the right-hand branch is closed because it contains Fr (at line 9) and Tr (at line 7). This closed tableau is a tableaux proof of $\neg(q \wedge r) \rightarrow (\neg q \wedge \neg r)$.

Let's consider a final example to illustrate what happens if the formula we are working with is *not* a validity. Now, the formula $(p \wedge q) \rightarrow (r \vee s)$ is certainly not valid; what happens when we try and falsify it using the tableaux method?

As usual, the first step simply states our goal, namely :

1 $F(p \wedge q) \rightarrow (r \vee s)$

As we need to falsify an implication, the relevant expansion rule is $F_{\rightarrow}$:

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}$
3	$F(r \vee q)$	$1, F_{\rightarrow}$

Now line 2 requires us to make a conjunction true. The expansion rule $T_{\wedge}$ required is clear: we must make both conjuncts true. Applying $T_{\wedge}$ yields:

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}, \surd$
3	$F(r \vee q)$	$1, F_{\rightarrow}$
4	Tp	$2, T_{\wedge}$
5	Tq	$2, T_{\wedge}$

Now let's deal with line 3. The relevant rule is $F_{\vee}$. Applying it yields:

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}, \surd$
3	$F(r \vee q)$	$1, F_{\rightarrow}, \surd$
4	Tp	$2, T_{\wedge}$
5	Tq	$2, T_{\wedge}$
6	Fr	$3, F_{\vee}$
7	Fs	$3, F_{\vee}$

But now the expansion process halts. This tableau is rule-saturated: we've applied the relevant rules to line 1–3, while lines 4–7 simply stipulate what has to be done with atomic information. But note that there is crucial difference between this tableau and the previous rule-saturated tableaux we have seen: the single branch in this tableau is not closed, it's *open*. That is, it does *not* contain contradictory instructions. In fact, it gives us very sensible instructions indeed: lines 4–7 tell us to make p true, q true, r false, and s false. As the reader can easily check, doing this falsifies $(p \wedge q) \rightarrow (r \vee s)$, thus this formula is *not* a validity.

More generally, a rule-saturated tableau is called *open* iff it contains at least one open branch. If we obtain an open tableau when we try to falsify some formula, this means that the formula is *not* a validity. Moreover, just as in the previous example, every open branch on the open tableau actually contains an explicit 'falsification recipe' for the formula: we falsify it by assigning truth values to the atomic symbols in the the way the open branch stipulates. To put it another way, open branches of rule-saturated tableaux tell us build a model that falsifies the formula we started with.

4.2 Propositional Tableaux

The reader should now have a fairly clear grasp of the main ideas and intuitions underlying tableaux proofs, so let's now discuss the method more systematically. We begin by listing the 8 main expansion rules, and classifying them into three types: *unary rules*, *conjunctive rules*, and *disjunctive rules*. This classification isn't vital, but it will help us keep the implementation neat.

For each Boolean connective B, we have two tableaux rules, T_B and F_B. T_B tells us how to make a formula with B as its main connective true, while F_B tells us how to make it false. The rules for negation, $T_\neg$ and $F_\neg$, are the simplest:

$$\frac{T\neg\phi}{F\phi} \qquad\qquad \frac{F\neg\phi}{T\phi}$$

These rules should be read from top to bottom. Given the examples discussed in the previous section, their meaning should be reasonably clear. In each rule, the signed formula above the horizontal line is the *input* to the rule, and the signed formula below it is the *output*. For example, $T_\neg$ takes as input signed formulas of the form $T\neg\phi$, and returns as output signed formulas of the form $F\phi$. Similarly, $F_\neg$ takes as input signed formulas of the form $F\neg\phi$ and returns as output signed formulas of the form $T\phi$. In what follows, we shall call these two rules our *unary* rules. This is simply shorthand for the fact that both rules return a single formula as output.

The rules for the binary connectives $\wedge$, $\vee$ and $\rightarrow$ are rather more interesting. Here are $T_\wedge$ and $F_\wedge$, $F_\vee$ and $T_\vee$, and $F_\rightarrow$ and $T_\rightarrow$, respectively:

$$\frac{T(\phi\wedge\psi)}{\begin{array}{c}T\phi\\T\psi\end{array}} \qquad\qquad \frac{F(\phi\wedge\psi)}{F\phi \mid F\psi}$$

$$\frac{F(\phi\vee\psi)}{\begin{array}{c}F\phi\\F\psi\end{array}} \qquad\qquad \frac{T(\phi\vee\psi)}{T\phi \mid T\psi}$$

$$\frac{F(\phi\rightarrow\psi)}{\begin{array}{c}T\phi\\F\psi\end{array}} \qquad\qquad \frac{T(\phi\rightarrow\psi)}{F\phi \mid T\psi}$$

Again, all six rule should be read from top to bottom, the top being the input to the rule, the bottom the output. We shall call the three rules in the left-hand column (that is, $T_\wedge$, $F_\vee$, and $F_\rightarrow$) *conjunctive rules*. The three rules on the right-hand side (that is, $F_\wedge$, $T_\vee$, and $T_{\rightarrow}$) are called *disjunctive rules*. Note that each of $\wedge$, $\vee$ and $\rightarrow$ gives rise to a pair of rules, one of which is conjunctive, the other disjunctive. Both conjunctive and disjunctive rules return two formulas as output — however, as we have already seen, they differ in the effect they have on tableaux. In particular, use of a disjunctive rule splits the branches of the tableaux containing the input formula into distinct branches, each of which records one of the two alternative output formula.

In the discussion that follows, we'll assume that we only have these 8 rules at our disposal. However, the reader should be aware that it is very straightforward to give expansion rules for other connectives, though such rules won't always fit into our three way classification. For example, it can can be useful to have the expansion rules for $\leftrightarrow$ at our disposal. Here are the required rules:

$$\frac{T(\phi \leftrightarrow \psi)}{\begin{array}{c|c} T\phi & F\phi \\ T\psi & F\psi \end{array}} \qquad\qquad \frac{F(\phi \leftrightarrow \psi)}{\begin{array}{c|c} T\phi & F\phi \\ F\psi & T\psi \end{array}}$$

Note that for this connective we *don't* obtain a neat pair of rules, one of which is conjunctive, the other disjunctive, as we did for $\wedge$, $\vee$ and $\rightarrow$. Moreover, though both rules are 'disjunctive' in the sense that both force us to split tableaux branches, these rules yield 4 output formulas, 2 for each possible branch. The exercises below ask the reader to develop expansion rules for some other connectives.

Now that we know what our expansion rules are, let's make our previous discussion of tableaux and tableaux proofs a little more rigorous.

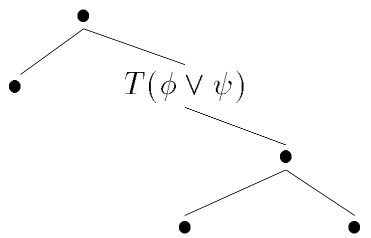
A (propositional) *tableau* is simply a tree, each of whose nodes is a signed (propositional) formula. A *branch* of a tableau is simply a branch of such a tree — that is, a collection of nodes (that is, signed formula) that contains exactly one leaf node together with all the nodes which dominate it.

An *initial tableau* — that is, the kind of tableau with which we start the tableaux expansion process — is a tableau that has only has a single branch. The reader may be slightly surprised by this definition. Isn't it too general? Why don't we simply define an initial tableau to be a tableau consisting of a single signed formula of the form $F\phi$? Actually we *could* use this more specific definition — but, as we shall later see, the slightly more general definition given is rather useful.

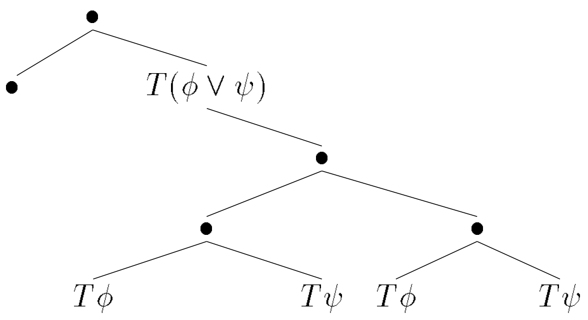
We carry out the tableau expansion process as follows. Given a tableau, try to find a node in it that (a) isn't a signed atomic formula, and (b) to which we haven't already applied an expansion rule. Let's call such nodes *unexpanded nodes*. If there are no unexpanded nodes, we can't do anything: we have a *rule-saturated* tableau and are finished. So suppose there is at least one unexpanded node. This node has the form $S\phi$, where S is either T or F, and ϕ is a propositional formula. Moreover, as ϕ is not atomic, it has a main connective B, where B is one of the connectives $\neg$, $\wedge$, $\vee$, or $\rightarrow$. We can then 'read off' the required rule: we have to apply rule S_B.

What happens when we apply a rule to a node? That depends on whether the rule is unary, disjunctive, or conjunctive. If the rule is unary, we extend every branch containing the input node by adding on the output formula of the rule as the new leaf node. If the rule is conjunctive, we extend every branch containing the input node by adding on, again at the leaf node, both output formula of the rule. Finally, if the rule is disjunctive, we extend every branch containing the input node to two distinct branches, one containing each of two choices of output formulas. Again, both the required additions are carried out at the leaf node of the original branches.

The basic idea of tableau expansion should be clear form the examples we discussed earlier, but there is one point worth emphasizing. A signed formula may belong to several branches. (For example, the root node belongs to every branch of a tableau.) When we perform an expansion, we have to extend *every* branch on which the input formula lies in the appropriate way. For example, consider the following tree:



If we expand the indicated node $T(\phi \vee \psi)$, we obtain:



The tableau expansion process starts when we are given an initial tableau. We apply rules to the initial tableau, and then to the tableaux obtained by earlier rule applications, and so on, until it is not possible to apply any more rules. As we discussed earlier, we can apply the rules in any order we like. The tableau expansion process stops when it is not possible to apply any more rules, that is, when we obtain a rule-saturated tableau. Here an important question looms: is the expansion process always guaranteed to stop? Yes, it is. We will point out why later in the chapter — but it would be a good idea for the reader to try figuring this out now unaided.

We are now almost ready to say what a *tableau proof* is. First, a branch of a tableau is *closed* if it contains both $T\phi$ and $F\phi$, where ϕ is some formula. A branch that is not closed is called *open*. A tableaux is closed if *every* branch it contains is closed, and open if it contains *at least one* open branch. Now for the key definition:

> *A formula ϕ is* tableaux-provable *(or, more simply:* provable*) if and only if it is possible to expand the initial tableau consisting of a single node $F\phi$ to a closed tableau. We use the notation $\vdash \phi$ to indicate that ϕ is provable.*

And that's the (propositional) tableaux method. To conclude our discussion, two remarks.

First, note that the tableaux method really is a purely *syntactic* method. This should be clear. If we want to test whether ϕ is provable we have to start with the initial tableau $F\phi$, and then try and expand it to a closed tableau. The entire expansion process is governed by syntactic ideas. For example, when we expand a node, by simply looking at its sign and main connective we know which rule to apply. Moreover, closure is a purely syntactic concept: it simply amounts to looking for items of the form $T\phi$ and $F\phi$ on some branch. Of course, as our presentation has tried to emphasize, the tableaux method is driven by clear *semantic* ideas. Nonetheless, *using* the method doesn't require any semantic insight at all. Even if we didn't know what the signs T and F were meant to stand for — or indeed, what $\neg$, $\wedge$, $\vee$, and $\rightarrow$ were meant to represent — we would still have a well defined way of manipulating logical formulas. It would probably be an exaggeration to claim that we could train a monkey to carry out tableaux proofs, but as we shall see below, we can certainly implement them in PROLOG rather easily.

The second point concerns our definition of initial tableaux. Why did we opt for the general definition, rather simply stating that initial tableau always have the form $F\phi$? Simply because, as we shall now see, by using different initial tableau we can use tableaux for different purposes.

First, we can use tableaux to *directly* test whether arguments are valid. For example, suppose we wanted to test

$$\phi_1, \ldots, \phi_n \models \psi,$$

for validity. Now, as we pointed out earlier, by the Deduction Theorem this is equivalent to testing whether $\phi_1 \wedge \cdots \wedge \phi_n \rightarrow \psi$ is a valid formula, thus we could simply try using the tableaux method to falsify this formula. But there is a more direct way. An argument is valid if and only if whenever all the premisses are true, the conclusion is true also. Hence, to test whether the above argument is valid, we could form the initial tableau:

$$\begin{array}{c} T\phi_1 \\ \cdot \\ \cdot \\ \cdot \\ T\phi_n \\ F\psi \end{array}$$

That is, we simply use the tableau method to see whether it is possible to make all the premisses true and the conclusion false. If we *can't* do this — that is, if we obtain a closed tableau — the argument is valid.

But we can do more with tableaux than simply testing whether formulas and arguments are valid: we can use them as a tool for telling us how to build models. To give a very trivial example, suppose we wanted to know exactly how to make the formula $\neg p \wedge (r \vee q)$ true. The tableau method will tell us how to do this. As our initial tableau we take:

$$T\neg p \wedge (r \vee q).$$

That is, we are simply asking: "How do we make this formula true?" To get an answer, all we have to do is carry out the expansion process in the usual way:

$$\begin{array}{llll}
1 & & T\neg p \wedge (r \vee q) & \surd \\
2 & & T\neg p & 1, T_\wedge, \surd \\
3 & & T(r \vee q) & 1, T_\wedge, \surd \\
4 & & Fp & 2, T_\neg, \surd \\
\end{array}$$

$$\begin{array}{lll|lll}
5 & Tr & 3, T_\vee & 6 & Tq & 3, T_\vee
\end{array}$$

This tableaux is rule-saturated and contains two open branches. The left hand branch tells us that one way to make the input formula true is to make p false and r true, while the right hand branch tells us that there is also another way to achieve this, namely by making p false and q true.

Exercise 4.2.1 *Show that the following formula's are all provable:*

1. $\neg\neg p \rightarrow p$
2. $((p \rightarrow q) \rightarrow p) \rightarrow p$
3. $(\neg p \rightarrow \neg q) \rightarrow (p \rightarrow q)$
4. $p \leftrightarrow (p \wedge (q \vee p))$
5. $(p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r))$

Exercise 4.2.2 *The connectives* nand *and* nor *are defined by the following truth tables:*

p	q	*p nand q*	*p nor q*
True	True	False	False
True	False	True	False
False	True	True	False
false	False	True	True

Give the tableaux expansion rules for both connectives.

Exercise 4.2.3 *There are two ways of using tableaux to show that an argument is valid: the indirect method that appeals to the Deduction Theorem, and the direct method just discussed. Explain why both methods give rise to the same result.*

4.3 A Prolog implementation

We shall represent tableaux in Prolog as lists containing lists of signed formulas. Signed formulas are represented as normal Prolog terms with functors `f` and `t` of arity 1. For example, the following is the Prolog representation of a tableau:

```
[[f(p > q)]].
```

This represents the tableau with just one branch, namely a branch containing the single signed formula $F(p \rightarrow q)$.

Here's a second example:

```
[[t(p & q), f(p & r)]].
```

Again, this represents a tableau with just one branch. This time, however, the branch contains *two* signed formulas, namely $T(p \wedge q)$ and $F(p \wedge q)$.

The Prolog implementation of propositional tableaux consists of a small collection of predicates which manipulate such lists in accordance with the tableaux expansion rules. For example, the list just given contains `t(p & q)`. Now, to make a conjunction true, we use expansion rule $T_\wedge$, which in this case tells us to add Tp and Tq to the given branch. It is easy to mimic the required expansion in Prolog: all we have to do is define a predicate which when given the previous list as input, returns the following one as output:

```
[[t(p), t(q), f(p & r)]].
```

Note two things. First, as expected, the output list contains `t(p)` and `t(q)`. Second, it does *not* contain `t(p & q)`. Why is this? Because once the relevant expansion rule has been applied to some formula, that formula is no longer relevant and can be removed. That is, removal of signed formulas from lists is the Prolog analog of the $\surd$ marking we used in our handwritten proofs.

Note that our new list contains `f(p & r)`, thus we can process it even further. Now, the rule for falsifying a conjunction is $F_\wedge$, a *disjunctive* rule. How are disjunctive expansions to be handled in PROLOG? Again, the basic idea is very simple. To handle $F_\wedge$, for example, we need simply define a predicate which when given the previous list as input, return the following one as output:

```
[[t(p), t(q), f(p)], [t(p), t(q), f(r)]]
```

Note that this list contains *two* lists. It is thus the PROLOG representation of a tableau containing *two* branches, namely the tableau with the signed formulas Tp, Tq and Fp on one (closed) branch, and the formulas Tp, Tq and Fr on the other.

Note that this approach to disjunctive rules differs from our handwritten approach. Our PROLOG code make two copies of the branch, one for each of the two disjunctive possibilities. This is somewhat heavy handed — but it makes the code easy to understand. On the other hand, when writing out proofs by hand we want to cut down the writing needed as much as possible — hence our use of tree-like structures in which the information common to two branches is shared. Of course, by taking trees as the fundamental data structure, it is possible to implement this kind of structure sharing in PROLOG, and doing so is an extremely valuable exercise.

Summing up, our PROLOG implementation is based round the representation of tableaux as lists containing lists of signed formulas. Each list of signed formula corresponds to a branch. We carry out expansions by manipulating these lists of signed formulas in the appropriate way. In particular, conjunctive (and unary) expansions will be carried out by taking the list representation of a branch, removing the relevant conjunctive (or unary) signed formula, and adding its component(s) to the list. Disjunctive expansions are carried out in much the same way, save that the process returns *two* lists, one containing each of the two possible components. Incidentally, the program given below does not handle the $\leftrightarrow$ connective; a good way of testing your understanding of the code is modify it so that it does.

Let's go systematically through the program. The outermost predicate is called `saturate`. This takes a single argument, a tableau, which it recursively attempts to rule-saturate with the help of the `expand` predicate. Note that base clause of `saturate` calls the `closed` predicate, which tests whether the input tableau is closed. By making this test here, we ensure that once a closed tableau is found, no further attempt is made to apply expansion rules.

```
saturate(Tableau):-
   closed(Tableau).

saturate(OldTableau):-
   expand(OldTableau,NewTableau),
   saturate(NewTableau).
```

The `closed` predicate is recursively defined in the expected way. To ground the recursion, the empty list is defined to be closed. A tableau containing branches can then be tested for closure by recursively peeling off its branches and checking for occurrences of both `t(X)` and `f(X)` for some formula `X`.

```
closed([]).

closed([Branch|Rest]):-
    member(t(X),Branch),
    member(f(X),Branch),
    closed(Rest).
```

The `expand/2` predicate is a high level 'organisational' predicate: it simply works its way recursively through all the branches in the input tableau, looking for branches to which one of the predicates `unary_expansion`, `conjunctive_expansion`, or `disjunctive_expansion` apply. The fourth, recursive clause of `expand/2` takes the next branch of the tableau if no more expansions can be carried out to the current branch.

```
expand([Branch|Tableau],[NewBranch|Tableau]):-
   unary_expansion(Branch,NewBranch).

expand([Branch|Tableau],[NewBranch|Tableau]):-
   conjunctive_expansion(Branch,NewBranch).

expand([Branch|Tableau],[NewBranch1,NewBranch2|Tableau]):-
   disjunctive_expansion(Branch,NewBranch1,NewBranch2).

expand([Branch|Rest],[Branch|Newrest]):-
   expand(Rest,Newrest)
```

The expansion predicates do the real work. Both the predicates `unary_expansion` and `conjunctive_expansion` have two arguments, namely an 'input branch', and an 'output branch', while `disjunctive_expansion` takes three arguments, an input branch and *two* output branches. All three predicates look for the occurrence of the appropriate type of signed formula (namely `unary`, `conjunctive`, and `disjunctive` respectively) as the first item in the input branch, use the library predicate `remove_first/3` to remove that occurrence, and then build the required new branch (or branches, in the case of disjunctive formulas) out of the component (or components) of the signed formula.

```
unary_expansion(Branch,[Component|Temp]):-
   unary(SignedFormula,Component),
```

```
   remove_first(SignedFormula,Branch,Temp).

conjunctive_expansion(Branch,[Comp1,Comp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Temp).

disjunctive_expansion(Branch,[Comp1|Temp],[Comp2|Temp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Temp).
```

Thus, it only remains to define what `conjunctive`, `disjunctive`, and `unary` signed formula are:

```
conjunctive(t(X & Y),t(X),t(Y)).
conjunctive(f(X v Y),f(X),f(Y)).
conjunctive(f(X > Y),t(X),f(Y)).

disjunctive(f(X & Y),f(X),f(Y)).
disjunctive(t(X v Y),t(X),t(Y)).
disjunctive(t(X > Y),f(X),t(Y)).

unary(t(~X),f(X)).
unary(f(~X),t(X)).
```

And that's that. Of course, using `saturate` directly is a little clumsy, and it is preferable to call it in a slightly more user-friendly way. For example, we can define a predicate `valid/1`:

```
valid(F):-
   saturate([[f(F)]]).
```

The reader should experiment with this program, and attempt at least the first two of the following exercises.

Exercise 4.3.1 *Modify the code so that it handles the nand and nor connectives defined above.*

Exercise 4.3.2 *Modify the code so that it handles the* $\leftrightarrow$ *connective.*

Exercise 4.3.3 *Reimplement propositional tableaux in a way that is more faithful to our hand-written tree based approach. That is, find a nice way of representing trees so that we don't need to duplicate branches when using disjunctive rules. [Hard]*

4.4 A little theory

To conclude our discussion, we shall discuss four concepts that every reader should have at least a nodding acquaintance with: *soundness*, *completeness*, *termination* and *efficiency*. We won't pursue these topics in depth, nor will we prove any technical results. Rather, our aim is simply to give a clear account of what the concepts are, and how they relate to tableaux systems; for further details, the reader is advised to follow up the references cited in the Notes. Along the way, we shall clarify some points that may be bothering readers.

First of all, the tableaux system presented in this chapter is a *sound and complete* proof system for propositional inference. What does this mean?

As we remarked at the start of the chapter, although a syntactic proof method only *uses* syntactic information, it has has to be *justifiable* in semantic terms. That is, when someone proposes some new proof method, we need guarantees that the proposal is semantically sensible.

The most fundamental property we demand of a proof system is that it be *sound*. Soundness is a essentially a 'no garbage' demand: if the proof system proves some formula, then that formula must be valid. Sound proof systems are thus semantically justifiable in a very strong sense: they will *never* prove formulas which are falsifiable.

Our tableaux system is sound. That is, for any propositional formula ϕ,

$$\text{if } \vdash \phi \text{ then } \models \phi.$$

So tableaux proofs will never lead us astray — but this shouldn't really come as a surprise. After all, we developed our tableaux expansion rules by thinking in overtly semantic terms. For example, we asked "How do we go about making a conjunction true?" and gave the (obviously sensible) answer "By making both conjuncts true". Indeed, *all* our expansion rules reflect the semantic definitions of the connectives in an obvious way, so it is hardly surprising that they are 'semantically safe'. And in fact, one proves the soundness of the tableaux method simply by thinking a little more systematically about the consequences of the 'semantic safety' of the expansion rules. We won't spell the details out here, but they are not particularly difficult and we suggest that mathematically inclined readers try to pin down the required argument precisely.

What about completeness? This is a much stronger (and more interesting) demand: a complete proof system is one which is capable of proving *all* valid formula. That is, if ϕ is any valid formula whatsoever, then it must be possible to give a proof of ϕ.

Our tableaux system is complete. That is, for any propositional formula ϕ,

$$\text{If } \models \phi \text{ then } \vdash \phi.$$

Intuitively, no validity lies beyond the reach of the tableaux proof method: if a formula

ϕ is valid, then it is possible to expand the initial tableau $F\phi$ to a closed tableau.[1] Note that because the tableaux method is both sound and complete we have that:

$$\vdash \phi \quad \text{iff} \quad \models \phi.$$

That is, we have a perfect match between the syntactic concept of tableaux-provability and the semantic concept of validity.

Proving that the tableaux method is complete is a non-trivial exercise which lies beyond the scope of this book; readers interested in the details should consult one of the references cited in the Notes. However, it *is* worth knowing that the key step of the completeness proof is to prove the following:

> *If it is possible to expand an initial tableau $F\phi$ to an* open *rule-saturated tableau, then one can falsify ϕ by giving the atomic symbols the truth values stipulated on any of the open branches of such a tableau.*

This is worth knowing, because it allows us to dispel an issue that may be worrying some readers: how do we know that a formula *isn't* provable?

The problem is this. We defined $\vdash \phi$ to mean that it is possible to extend the initial tableaux $F\phi$ to a closed tableaux. Now, by changing the order we apply expansion rules, it may be possible to expand the initial tableau $F\phi$ to many different rule-saturated tableau. This is worrying: perhaps some of these tableaux are open and others are closed! So, if we expand $F\phi$ and obtain an open rule-saturated tableau, this doesn't seem to tell us much. Perhaps some other expansion would have led to a closed tableaux. Perhaps — what a horrible thought! — we have to check all possible tableaux expansions before we can safely conclude that ϕ isn't provable.

In fact, we don't need to worry. For suppose some expansion of $F\phi$ leads to an open rule-saturated tableau. Then, by the property just noted, it is possible to falsify ϕ. It follows that no other expansion of $F\phi$ can possibly lead to a closed rule-saturated tableaux. To see this, note that if there was such a closed rule-saturated tableau, this would mean that ϕ was provable. By soundness, this would mean that ϕ was valid — but this would contradict the fact that we can falsify ϕ. In short, as soon as we succeed in expanding $F\phi$ to an open rule-saturated tableau, it is safe to conclude that ϕ is *not* provable.

Soundness and completeness are not the only desirable properties of proof systems. If one is interested in implementation, one wants to know something about a system's computational properties. For example, is the method guaranteed to *terminate* (or *halt* or *stop*) on all possible input formula? And how *efficient* is the method anyway?

[1] Incidentally, it is easy to give examples of tableaux systems that are sound but *not* complete — simply throw away expansion rules! For example, if we discard the rule $F_\neg$, we still have a *sound* tableaux system, but we don't have enough power left to prove all validities, as we can no longer have all the rules we need for coping with negated formulas.

The expansion process used by our propositional tableaux is guaranteed to halt on all possible input. To see why, note that every tableaux expansion rule takes a signed formula and returns a *finite* number of signed formulas (in fact, no rule returns more than 4 signed formula). Moreover, crucially, each of the formula returned contains fewer connectives than the input formula. Thus, as we never have to apply a rule to the same formula twice, after every expansion the collection of unexpanded formulas remains finite, and moreover, any new unexpanded formulas we obtain as a result of rule applications are *simpler*. Thus — no matter which sequence of expansions we choose to make — we will achieve rule-saturation after a finite number of steps. Termination is thus guaranteed. Making this argument really tight requires a little care, but the basic idea should be clear.

Are tableaux an efficient method of doing propositional inference? The honest answer is: *nobody knows*. In fact nobody knows if there is *any* proof method which can prove all valid formulas efficiently. Let's discuss this a little further.

For many formulas, tableaux fare better than other methods. To give an obvious example, suppose we tested the validity of the following formula using truth tables:

$$(q \rightarrow (r \vee (p \wedge \neg t \rightarrow \neg\neg s))) \vee \neg(q \rightarrow (r \vee (p \wedge \neg t \rightarrow \neg\neg s))).$$

The required truth table has 32 rows, which takes quite some time to fill out. The tableaux method, on the other hand, gets this example right in a single step. More generally, the tableaux method is obviously a fairly sensible, and by using a little common-sense when forming tableau (for example, applying as many conjunctive expansion rules as possible before applying disjunctive expansion rules) it is possible to give fairly concise proofs of typical validities.

Unfortunately, not all validities are 'typical'. It is possible to show that there are an infinite number of formulas which create terrible problems for the tableau method: any closed tableau formed by starting with one of these formulas contains a huge number of nodes. Finding such validities isn't easy. Devising them requires careful thought about which kind of formulas require heavy use of disjunctive rules. However, such validities are out there, thus although the tableau method looks better than the truth table method, the bottom line is that it is subject to the same kind of combinatorial explosions.

Now — as far as anybody knows — this isn't because tableaux is a poor proof method. In fact, nearly all the well known proof methods (such as resolution, natural deduction, and so on) have been shown to be subject to similar combinatorial explosions. There simply is no known proof method for propositional languages that avoids combinatorial explosion. In fact, it is widely conjectured that no such proof method exists. For further discussion of this point, consult the references cited in the Notes.

Let us sum up what we have learned in this section. As a proof method, the tableaux system introduced in this chapter is as well-behaved as one could reasonably hope for. It is semantically sensible, and moreover, strong enough to prove all valid formulas. Moreover, it is guaranteed to terminate on all possible input. On 'typical' formulas it seems reasonably efficient — though the reader should be aware that there are (infinitely many) formulas which, if given as input, will give rise to tableaux that are too big to be practicably computed.

Notes

Proof theory is a rich and fascinating subject. The reader can gain a good overview of various proof systems, and how they relate to each other, by consulting Sundholm [Sun83]. Both signed and unsigned tableaux systems are discussed.

For tableaux systems, the classic source is Smullyan [Smu95]; our discussion of signed tableaux is based on Smullyan's treatment. Good treatments of signed tableaux may be found in Fitting [Fit96] and Bell and Machover [BM77]. The reader interested in finding out how to prove the tableaux method complete will find all that is needed in these three books. Our implementation of propositional tableaux was influenced by the implementation of unsigned tableaux given in Fitting [Fit96]. The reader will find it instructive to compare the two implementations. (Apart from anything else, it's a good idea to find out about what unsigned tableaux are and how they work.)

The analysis of the complexity of proof methods is an active field of research. Perhaps the best starting point is Urquhart [Urq95]. This contains a good discussion of why the tableau method sometimes leads to combinatorial explosion. Another interesting (and very readable) paper is D'Agostino [D'A92], which shows that in certain cases the tableaux method performs *worse* than the truth table method.

Finally, although it takes us far from the concerns of the present book, it is worth noting that proof theory arguably has far deeper connections with natural language semantics than our discussion might suggest. There is an important tradition which claims that that meaning shouldn't be explained in terms of truth conditions, but in terms of assertability conditions. (Roughly speaking, on this view the meaning of an utterance is the reason we have for holding it, not the situations in which it is true.) Semanticists in this tradition tend to regard proofs as the primary semantic objects. At first sight this may appear to be a rather strange view, but it has a lot to recommend it. A good introduction to this line of thought is Sundholm [Sun86]. Moreover, as Ranta [Ran94] demonstrates, the approach is of relevance to computational semantics.

Chapter 5

First-Order Inference and Unification

In this chapter we extend our tableaux system to first-order languages. As promised, we do so by reducing first-order formulas to propositional ones. The resulting system is natural and easy for human beings to use. Unfortunately, as a candidate for automation it's awful: it makes use of an infinite space of possible substitutions that requires human insight to navigate.

So we rethink our strategy. Roughly speaking, we decide to delay the problem of finding good substitutions. Our plan is to use free variables to build up a set of constraints on substitutions, and then to hand over the problem of solving the constraints to something new: a *unification* component. Our hope is that a composite system of this sort will lend itself to computational implementation.

The remainder of the chapter is spent developing the various ideas needed to carry this plan out. We discuss unification and its implementation in PROLOG in some detail. We then consider how to combine unification with the tableaux proof method, a discussion which leads us to the use of Skolem terms. Finally, we draw all the threads together and implement a full first-order theorem prover in PROLOG.

5.1 First-Order Tableaux: a first attempt

We now extend the signed tableaux proof method to first-order languages. Conceptually, the extension is very simple. Our new tableaux rules will allow us get rid of quantifiers by substituting suitable terms for bound variables. In effect, they let us reduce first-order formulas to propositional ones.

What kinds of rules enable us to do this? Let's consider two examples. Suppose a tableau contains the signed formula $T\forall$xKILLER(x), and suppose we are working with a first-order language containing the constants JULES and BUTCH and a 1-place function symbol FATHER. Then it should be legitimate to extend the tableau by adding on any of the following signed formulas: TKILLER(JULES), TKILLER(BUTCH), and

$$T\text{KILLER(FATHER(FATHER(FATHER(JULES))))}.$$

(After all, if everyone is a killer, every term picks out a killer.) More generally, given any universal formula prefixed by the sign T, we should be free to throw away the universal quantifier, substitute any term for the newly freed variable in the matrix, prefix the result by T, and extend the tableau with the result.

Analogous tableaux extensions should be legitimate when we are told that an existential sentence is false. For example, suppose a tableau contains $F\exists$xSHOOTS(JULES,x). Then it should be legitimate to deduce that FSHOOTS(JULES,FATHER(JULES)), and that FSHOOTS(JULES,BUTCH), and so on. (After all, if it's false that Jules shoots someone, then its false that Jules shoots any person we care to name.)

Such examples lead us to formulate the following two tableaux rules, $T_\forall$ and $F_\exists$:

$$\frac{T\forall \mathrm{x}\phi}{T\phi(\tau)} \qquad\qquad \frac{F\exists \mathrm{x}\phi}{F\phi(\tau)}$$

Here $\phi(t)$ denotes the result of replacing the variable bound by the quantifier by some closed term t. (Recall that a closed term is a term that does not contain any variables.) We call these two rules *universal rules*.

While both these rules are sound, and clearly trade on the semantic intuitions underlying the tableaux method, they differ from the propositional tableaux rules in important respects. First, each rule licenses as many tableaux extensions as there are closed terms in the language. In particular, if there are infinitely many closed terms in the language, each rule licenses infinitely many ways of extending a tableau. Second, in general we will have to apply a universal rule more than once to a particular occurrence of a signed formula. Here's a simple example. We shall show that

$$\forall \mathrm{x}\text{DIE}(\mathrm{x}) \rightarrow \text{DIE(MIA)} \wedge \text{DIE(ZED)}$$

is a theorem.

1	$F(\forall x\textsc{die}(x) \rightarrow \textsc{die}(\textsc{mia}) \wedge \textsc{die}(\textsc{zed}))$	
2	$T\forall x\textsc{die}(x)$	$1, F_{\rightarrow}$
3	$F(\textsc{die}(\textsc{mia}) \wedge \textsc{die}(\textsc{zed}))$	$1, F_{\rightarrow}$
4	$T\textsc{die}(\textsc{mia})$	$2, T_{\forall}$
5	$T\textsc{die}(\textsc{zed})$	$2, 2_{\forall}$
6	$F\textsc{die}(\textsc{mia})$ $3, F_{\wedge}$	7 $F\textsc{die}(\textsc{zed})$ $3, F_{\wedge}$

Note the way we had to apply $T_{\forall}$ twice to line 2: once to get $T\textsc{die}(\textsc{mia})$, and once to get $T\textsc{die}(\textsc{zed})$. Thus we have lost one of the pleasant properties of propositional tableaux: it is no longer true that we are through with an occurrence of a signed formula once we've applied a rule to it.

Let's turn to to the next issue. What sort of tableaux rules are needed to deal with signed formulas of the form $T\exists x\phi$ or $F\forall x\phi$? This is a more subtle matter. Suppose a tableau contains the signed formula $T\exists x\textsc{killer}(x)$. It is *not* legitimate on the basis of this information to deduce that $T\textsc{killer}(\textsc{jules})$, or that $T\textsc{killer}(\textsc{butch})$, or indeed to deduce that $T\textsc{killer}(\textsc{closed-term})$ for *any* closed term of the language we are working with. *Someone* is a killer — but we don't know who. So how are we to eliminate the quantifier?

Actually, the solution is straightforward: we invent a brand new name for the entity whose existence is asserted, and eliminate the quantifier by substituting this new name. Such brand new names are called *parameters*, and by using parameters we can deal with true existential statements, and false universal ones too. For example, if a tableau contains $T\exists x\textsc{killer}(x)$, we give this killer (whoever he or she is) a new name (that is, we choose a new parameter, say c) and extend the tableau by asserting $T\textsc{killer}(c)$. Similarly, given the signed formula $F\forall x\textsc{religious}(x)$, we christen the unbeliever (whoever he or she is) with a new name (say c_8) and extend the tableau by asserting $F\textsc{religious}(c_8)$.

Let's make these ideas precise. Suppose we are working in a first order language over some vocabulary V. Let PAR be a (countably infinite) set of new constant symbols, that is, constant symbols that *don't* belong to V. We'll call these new constant symbols parameters, and reserve the symbols c, c_1, c_2, ..., and so on, for them. From now on, when we want to do tableaux proofs, we *won't* work in our original language (that is, the language built over the vocabulary V). Rather we'll work in the first-order language whose vocabulary consists of all the original vocabulary V, and in addition, all the new constant symbols in PAR.

Given these ideas, it's easy to define the the rules $F_{\forall}$ and $T_{\exists}$:

$$\frac{F\forall x\phi}{F\phi(c)} \qquad\qquad \frac{T\exists x\phi}{T\phi(c)}$$

Here $\phi(c)$ denotes the result of substituting a parameter c *that we haven't used so far in the tableau proof*, for the newly freed variable in the matrix. We call these two rules *existential rules*.

Two points about these rules must be grasped. First, when we use the existential rules, it is absolutely vital that we substitute parameters that haven't been used so far in the tableau construction. To see why, suppose a tableau contains both $T\exists$xKILLER(x) and $T\exists$xRELIGIOUS(x). Suppose we first apply $T_\exists$ to $T\exists$xKILLER(x) using the parameter c_5 (which, let us suppose, hasn't been previously used) to name the killer. That is, we extend the tableau with TKILLER(c_5). Now, if at some later stage we apply $T_\exists$ to $T\exists$xRELIGIOUS(x), it would be outrageous to re-use the parameter c_5. If we did this (that is, if we 'deduced' that TRELIGIOUS(c_5)) we would in effect be claiming that there is a single individual (namely the one named by c_5) who is both a killer and religious. This simply doesn't follow from the given information. All we know is that there is at least one killer, and at least one religious person. They may well be different people, hence we need to assign each of them a fresh new name. In short, once we've used an existential rule to replace a quantifier by a parameter, that parameter becomes 'old', and cannot later be re-used.

There is a second important point that the reader should note: the definition of the existential rules has consequences for the universal rules. When carrying out first-order tableaux proofs we no longer work in the original first-order language, but in the original language enriched with an infinite collection PAR of new constants. Now, constant symbols — including parameters — are closed terms, so we should be free to substitute parameters when we use a universal rule. (In fact, its *crucial* that we be able to do this, as the following example will make clear.) But this means that we will *always* have infinitely many choices when it comes to using a universal rule, even if the original language had no constants or function symbols at all.

Let's have a look at another example. We will show that

$$\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y)$$

is a theorem. (This is a rather pretty example. It's simple, but puts all four quantifier rules to work.)

1	$F(\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y))$	
2	$T\exists x \forall y \textsc{shoots}(x,y)$	1, $F_\rightarrow$
3	$F\forall y \exists x \textsc{shoots}(x,y)$	1, $F_\rightarrow$
4	$T\forall y \textsc{shoots}(c_1, y)$	2, $T_\exists$
5	$F\exists x \textsc{shoots}(x, c_2)$	3, $F_\forall$
6	$T\textsc{shoots}(c_1, c_2)$	4, $T_\forall$
7	$F\textsc{shoots}(c_1, c_2)$	5, $F_\exists$

The key point to notice about this proof is the way the existential and universal rules interact. In particular, note the way we used the existential rules to introduce the new parameters (c_1 in line 4 and c_2 in line 5) and then used the universal rules to make further use of these symbols. It should be clear from this example that it is vital that the universal rules have access to the parameters.

5.2 An efficiency problem

How efficient is this tableaux system? This is a question we must answer in two ways.

The first (and most fundamental) thing the reader should know is the following: the first-order tableaux system does *not* give rise to an algorithm for determining which first-order formulas are valid. An algorithm is a recipe which, when given an instance of a problem to solve, halts after a finite number of steps with the correct answer. *There is no algorithm at all for determining the validity of arbitrary first-order formulas.* That is, first-order validity is *undecidable*. It is certainly possible to implement syntactic proof systems (for example, tableaux systems) and we shall do in this chapter, but no implementation of any system is guaranteed to terminate on all possible input. Incidentally, the tableaux system just described *is* (sound and) complete. That is, if a formula ϕ is valid, then it possible to construct a (finite) closed tableaux that has $F\phi$ as its root node. Thus the proof system just described is a genuine syntactic analysis of the semantic concept of first-order validity, and indeed a rather natural one. However this analysis does not yield an algorithm for determining which first-order formulas are valid, for no such algorithm exists.

As there is no general computational solution to the problem, it is meaningless to speak of efficiency in an absolute sense, so let's restate the question in a more realistic way. Many first-order theorem provers are useful practical tools. Proof search won't always terminate, but they work well on wide ranges of input, and can be used as components of larger systems quite satisfactorily. So the question we should next pose is: how good is the tableaux system just described as a practical basis for automated first-order theorem proving? This question has a clear answer: *it's terrible*. Although our tableaux system is conceptually simple, it's a computational nightmare.

The heart of the problem lies with the universal rules. They offer us an infinite menu of substitutable terms. If $T\forall x\phi$ belong to a tableau, then we are free to extend extend it by first adding $\phi(c_1)$ then $\phi(c_2)$ then $\phi(c_3)$, $\ldots$, and so. (We could have done this in the previous tableau, for example, starting at line 4.) Most such extensions will be completely pointless. In the examples given above, it was intuitively clear which substitutions were sensible; we used our sense of what was relevant to guide our choice of substitutions. Unfortunately, computers lack our intuitions. If we want a reasonably practical implementation, we need to find a method choosing substitutions that doesn't depend on human insight.

Here's what we shall do. First, we'll change the universal rules slightly: we'll never substitute closed terms, we'll *always* substitute free variables instead. In a sense, we are not going to make a real choice of substitutions at all — we are going to use free variables as 'dummies' that will enable us to delay making this decision. In this way, we will gradually build up a whole system of 'substitution equations', a set of constraints on variable values that contain a great deal of information about the terms in the tableau we are building. Crucially, there is an algorithm for solving such constraint sets, the famous *unification algorithm*. We will use unification to look for solutions to the constraints that lead to branch closure.

That's our strategy in outline. An awful lot of detail remains to be filled in. For a start, blending unification with tableaux is going to force us to rethink the existential rules, and there are many other details that will require careful attention. Nonetheless, unification is the key to further progress, so let's examine this concept in some detail.

5.3 Unification

Unification is the process of carrying out substitutions on two terms so that they become identical. The reader who has made it this far will certainly have a informal idea of what substitutions are, and indeed, since we are working with Prolog, a reasonable impression of what unification involves too. However, if we are to develop and implement a unification algorithm suitable for automated theorem proving, we need to make the concept of substitution more precise, and spell out exactly exactly what the unification process involves. We'll also need to figure out a natural way of implementing unification in Prolog. Now, the most straightforward way of doing so is to adapt Prolog's in-built unification mechanism, and we will do this. However this means we will need to know exactly how Prolog unification differs from 'real unification'. All in all, there's rather a lot to do, so let's get straight down to business.

Suppose we have chosen the first-order language we are going to work with. Then a *substitution* is a function that maps the set of variables to the terms of this language. We use the notation $\mathrm{x}\sigma$ (rather than $\sigma(\mathrm{x})$) to denote the value of x under the substitution σ.

We are most interested in *finite substitutions*. These are substitutions which only assign new terms to a finite number of variables; the rest they leave alone. That is, if σ is a finite substitution, then for all but a finite number of variables, $x\sigma = x$.

The simplest finite substitution is the one which does not assign new terms to *any* of the variables. This substitution is called the *identity substitution*, and we denote it by $\{\}$. We also have a special notation for other finite substitutions, namely:

$$\{\mathrm{x}_1/\tau_1, \ldots, \mathrm{x}_n/\tau_n\}.$$

Here x_1, ..., x_n are distinct variables, τ_1, ..., τ_n are terms, and $\tau_i \neq \mathrm{x}_i$ for any i from 1 to n. The notation x_i/τ_i means that the variable x_i is mapped to τ_i.

As we have defined them, substitutions only act on variables. In this section we will be exploring the effect of of substitutions on arbitrary terms. Here's a recursive definition of this concept.

Substitutions on Terms. Let σ be a substitution and τ a term. Then:

1. If τ is a variable x, then $\tau\sigma = \mathrm{x}\sigma$;
2. If τ is a constant, then $\tau\sigma = \tau$;
3. If τ has the form $f(\tau_1, \ldots, \tau_n)$, then $[f(\tau_1, \ldots, \tau_n)]\sigma = f(\tau_1\sigma, \ldots, \tau_n\sigma)$. (Here f is an n-place function symbol.)

Actually, we need to extend the substitution concept even further, for in the following section we will want to apply substitutions to formulas, and indeed, to entire tableaux. So let's define these concepts right away. First a piece of notation. If σ is a substitution, then by σ_x we mean the substitution that is exactly like σ except that $x\sigma_x = x$. We can now define recursively define $\phi\sigma$, the result of applying the substitution σ to the formula ϕ.

Substitutions on formulas.

1. If $\mathrm{R}(\tau_1, \ldots, \tau_n)$ is an atomic formula, then $[\mathrm{R}(\tau_1, \ldots, \tau_n)]\sigma$ is $\mathrm{R}(\tau_1\sigma, \ldots, \tau_n\sigma)$. (Here R is an n-place relation symbol.);
2. $[\neg\phi]\sigma$ is $\neg[\phi\sigma]$;
3. $[\phi \wedge \psi]\sigma$, $[\phi \vee \psi]\sigma$, and $[\phi \rightarrow \psi]\sigma$, are $[\phi\sigma] \wedge [\psi\sigma]$, $[\phi\sigma] \vee [\psi\sigma]$, and $[\phi\sigma] \rightarrow [\psi\sigma]$ respectively;
4. $[\forall \mathrm{x}\phi]\sigma$ is $\forall \mathrm{x}[\phi\sigma_x]$, and $[\exists \mathrm{x}\phi]\sigma$ is $\exists \mathrm{x}[\phi\sigma_x]$.

Substitutions on signed tableaux. If σ is a substitution and $\mathcal{T}$ is a signed tableau, then $\mathcal{T}\sigma$ is the signed tableau obtained by replacing every signed formula of the form $T\phi$ in $\mathcal{T}$ by $T[\phi\sigma]$, and every signed formula of the form $F\phi$ in T by $F[\phi\sigma]$.

With these definitions out of the way, let us return to our main task: understanding unification.

Because substitutions are functions we can compose them in the usual way. That is, if σ_1 and σ_2 are substitutions, then we can define a new substitution $\sigma_1\sigma_2$, which we call the *composition* of σ_1 and σ_2. For every variable x, we define $\mathrm{x}(\sigma_1\sigma_2)$ to be $(\mathrm{x}\sigma_1)\sigma_2$. That is, $\sigma_1\sigma_2$ first carries out the substitution σ_1 and then carries out the substitution σ_2.

There is one more concept we need to discuss before we can define unification. Let's approach it via an example. Suppose we want to make the terms f(c,y,w) and f(x,y,g(z)) identical (where c is a constant, w, x, y, and z are variables, f is a 3-place function symbol, and g is a 1-place function symbol). Let σ_1 be the substitution $\{x/c, w/g(z)\}$. Applying σ_1 to these terms has the desired result, for f(c,y,w))σ_1 = f(x,y,g(z))σ_1 = f(c,y,g(z)).

But there are other ways of making these terms identical. For example, let σ_2 be the finite substitution $\{x/c, w/g(z), y/h(u, x)\}$. Applying σ_2 to either term yields f(c,h(u,x),g(z)). Nonetheless, clearly σ_1 is a more general solution to the problem: σ_2 does too much work. For suppose we apply σ_1 to some term. Then, if we want to (or need to) we are always free to later give y the value h(u,x). To do so we need simply apply the substitution $\{y/h(u, x)\}$), and this two step process gives us the same effect we would have achieved by directly applying σ_2. But of course, we might not want to take this further step. (Perhaps mapping y to h(u,x) is incompatible with other substitutions we need to make.) If we use σ_2 to solve the problem, we are overcommitting ourselves.

Such considerations motivate the following definition. A substitution σ_1 is said to be *more general* than a substitution σ_2 if and only if there is some substitution θ such that $\sigma_2 = \sigma_1\theta$. That is, σ_1 is more general than σ_2 if we can get the effect of σ_2 by first carrying out σ_1 and then making a further substitution θ. Thus, reverting to our motivating example, $\{x/c, w/g(z)\}$ is more general than $\{x/c, w/g(z), y/h(u, x)\}$ because there is a substitution θ (namely $\{y/h(u, x)\}$) such that $\{x/c, w/g(z), y/h(u, x)\} = \{x/c, w/g(z)\}\theta$.

Incidentally, note that under this (standard) definition of 'more general than', each substitution σ is more general than itself. This is because we can get the effect of σ by first carrying out σ and then carrying out the identity substitution $\{\}$. Thus the 'more general than' relation is reflexive. It is also transitive. That is, if σ_1 is more general than σ_2, and σ_2 is more general than σ_3, then σ_1 is more general that σ_3. The reader may like to try proving this.

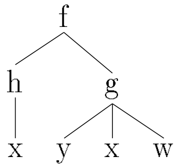
We are ready for the key definition.

Unification. Let τ_1 and τ_2 be terms. A substitution is a *unifier* for τ_1 and τ_2 if and only if $\tau_1\sigma = \tau_2\sigma$. Terms τ_1 and τ_2 are said to be *unifiable* if and only they have a unifier. A substitution σ is a *most general unifier* (or *mgu*) for two terms if and only if it is unifier for these terms, and is more general than any other unifier.

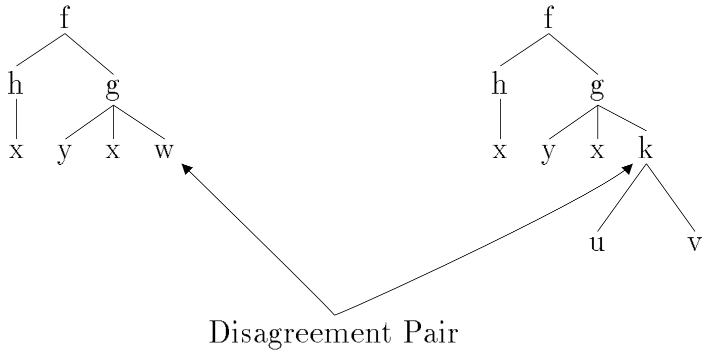
Now we know what unification is — but what is involved in computing unifiers? Ideally, what we want is an algorithm which will take as input two terms and determine whether or not they are unifiable. If the terms are unifiable, it should return their unifier as output. If they are not unifiable, it should halt and tell us so.

Such algorithms exist. Let's consider one of the more straightforward ones in some detail.

To appreciate what this algorithm does, we really need to think of terms as trees. For example, consider the term f(h(x),g(y,x,w)). Its parse tree — that is, the tree showing how it is built up out of sub-terms — looks like this:



When are two terms different? For our purposes, the following answer is the most useful: two terms are different if and only if their parse trees contain at least one *disagreement pair*. What's a disagreement pair? Here's an example.



Intuitively, the pair of terms $(\mathrm{w}, \mathrm{k}(\mathrm{u}, \mathrm{v}))$ is a disagreement pair for these terms because they are distinct terms that occupy 'corresponding places' in the two parse-trees. What is meant by 'corresponding places'? Simply the nodes that one reaches by following the same sequence of transitions from the root in the two trees. For example, if we follow the transition sequence $\langle$second daughter, second daughter$\rangle$ from the root of the first tree we arrive at the node labeled w, and if we follow the same transition sequence in the second tree we arrive at the node labeled k.

Disagreement pairs make terms different, thus unification algorithms should try to eliminate disagreement pairs. When are disagreement pairs eliminable, and how can they be eliminated?

First, suppose that two terms τ_1 and τ_2 are different because there a disagreement pair (d_1, d_2) such that *neither* d_1 nor d_2 is a variable. Then there is nothing we can do. No substitution can help us out. The terms τ_1 and τ_2 are not unifiable.

Now for the tricky question. Suppose that one of these terms (d_1 say) is a variable. Are the two terms unifiable? The answer is 'yes', *provided that the variable d_1 does not occur in d_2*. Think about it. Suppose d_1 is a variable, say x, and that x *doesn't* occur in d_2. Then we can eliminate this disagreement pair very easily: we simply need to replace x by d_2 (That is, we need simply perform the substitution $\{\mathrm{x}/d_2\}$.) To give a concrete example, the disagreement pair $(\mathrm{w}, \mathrm{k}(\mathrm{u}, \mathrm{v}))$ shown above is of this form. We eliminate it by replacing w by k(u,v).

On the other hand, if d_1 is a variable (say x) and x *does* occur in d_2, then unification is impossible. For example, suppose that d_2 is f(x). Then, no matter what value we choose for x, we will *never* render these two terms identical — we'll always have that extra function symbol f to reckon with.

Let's make these observations a little more systematic. Suppose we have found a disagreement pair (d_1, d_2) We will call this pair a *simple* disagreement pair if and only if at least one of the terms d_1 or d_2 is a variable that does not occur in the other. (It could happen, of course, that *both* d_1 and d_2 are variables that don't occur in the other — for example if $d_1 = \mathrm{x}$ and $d_2 = \mathrm{y}$. That's fine. As long as there's at least one we're happy.) Simple disagreement pairs are the ones we can repair. We do so by carrying out what we shall call the *relevant repair*. If d_1 is a variable that does not occur in d_2, then the relevant repair is the substitution $\{d_1/d_2\}$. If d_2 is a variable that does not occur in d_1, then the relevant repair is the substitution $\{d_2/d_1\}$. If both d_1 and d_2 are variables that don't occur in the other, then there are two substitutions that will repair the problem, namely $\{d_1/d_2\}$ and $\{d_2/d_1\}$. We'll arbitrarily stipulate that in such cases the relevant repair is $\{d_1/d_2\}$.

On the other hand, if the disagreement pair we have found is *not* simple, there's nothing we can do. Either we have that neither d_1 nor d_2 is a variable, or one of them is a variable that occurs in the other term.

We now know which disagreement pairs are eliminable, and how to carry out the elimination. And this means, we are only one small step away from an algorithm for solving the unification problem. To unify two terms, simply try and eliminate *all* the disagreement pairs! This idea immediately suggests the following non-deterministic algorithm:

> **input** terms τ_1 and τ_2
>
> **let** $\sigma := \{\}$
>
> **while** $\tau_1\sigma \neq \tau_2\sigma$
>
> > **choose** a disagreement pair (d_1, d_2) for $\tau_1\sigma$, $\tau_2\sigma$
> >
> > **if** (d_1, d_2) is not simple **then**
> >
> > > **write** τ_1 and τ_2 are not unifiable **and** HALT
> >
> > **else**
> >
> > > **let** $\sigma := \sigma \cup \rho$, where ρ is the relevant repair
> >
> > **endif**
>
> **endwhile**

This is a genuine computational solution of the unification problem. No matter which two terms it is given as input, it will halt after finitely many steps. When it halts, it will either have told us that the terms are not unifiable (and if it says this, it's right!) or it will have found out how to build the mgu of the two terms. These claims are not obvious; they require proof. The reader interested in finding out more should consult the references cited in the Notes.

In fact, the algorithm has one additional property: the mgus it produces are *idempotent*. That is, if σ is an mgu produced by this algorithm, then $\sigma\sigma = \sigma$. Why on earth is such an abstract looking property interesting? The answer is: idempotent mgus provide us with an easy way of solving the *simultaneous unification problem*, and it actually this problem will be important when working with tableaux.

We will be using unification to try and close tableaux branches. That is, we will look for branches containing pairs of atomic formula $T(R(\tau_1, \ldots, \tau_n))$ and $F(R(\tau_1, \ldots, \tau_n))$. If we can find a substitution σ that makes τ_1 identical to τ_1', and ..., τ_n identical to τ_n', then by applying σ we obtain a branch containing contradictory formula, that is, a closed branch. Thus we need to solve the problem of finding a single substitution that makes n pairs of terms identical; this is called the simultaneous unification problem.

Now, this problem may look harder than the ordinary unification problem, but actually it's not. It can be solved as follows. To find a substitution that identifies n pairs of terms, first find an *idempotent* mgu σ_1 for τ_1 and τ_1'. We can do this using the above algorithm. Next, find an *idempotent* mgu σ_2 for $\tau_1\sigma_1$ and $\tau_1'\sigma_1$. Again we can do this using the above algorithm. Next, find an *idempotent* mgu σ_3 for $\tau_1\sigma_1\sigma_2$ and $\tau_1'\sigma_1\sigma_2 \ldots$. In fact, all we need to do is keep 'chaining together' the solutions to each individual pair. The substitution $\sigma = \sigma_1\sigma_2 \ldots \sigma_{n-1}\sigma_n$ that is obtained in this way is a simultaneous mgu for the n pairs of terms. For this 'chaining together' method to work, the substitutions constructed at each step must be idempotent, and as the above algorithm yields idempotent mgus, it really does deliver everything we shall need for tableaux theorem proving.

The basic concepts should now be clear, so let's turn to a more practical issue: how can we use Prolog to unify terms? Now, we could approach this problem from scratch. However (as the reader is undoubtedly aware) one of the main mechanisms underlying Prolog is a form of unification. Perhaps we can use Prolog's in-built unification mechanism directly, or adapt it straightforwardly, to perform the term unifications we require? This certainly seems a sensible choice of strategy. Let's see what it involves.

The way Prolog performs unification is reasonably close to the above algorithm. There are two main differences. First, the algorithm given above is non-deterministic, whereas Prolog's in-built mechanism for unifying terms is deterministic. Clearly this is not a difference that need concern us. The second difference, however, is important. In the interests of efficiency, Prolog does not bother making the 'occurs check'. That is, given a disagreement pair (d_1, d_2), one of which is a variable, Prolog does *not* check whether this variables occurs in the other term, but will go straight ahead and attempt to carry out what it thinks the required repair is. This leads it to attempt to unify terms that aren't unifiable, and thus to stack overflows and other undesirable behaviour.

Now, we are interested in using unification as part of a first-order theorem prover. We really need to know whether or not two terms are unifiable, and we certainly *don't* want

PROLOG to mess things up with its 'Hey, just go for it!' behaviour. Can PROLOG be tamed? That is, can we make use of its in-built unification mechanism, but modify it so that makes an occurs check? The answer is 'yes'. The following code, adapted from Chapter 10 of Sterling and Shapiro [SS86], does precisely this.

The main predicate `unify` uses the in-built unification predicate `=` to carry out the required unifications, but carefully inserts an occurs check (via the `not_occurs_in/2` predicate) where required.

```
unify(X,Y):-
   var(X),var(Y),X=Y.
unify(X,Y):-
   var(X),nonvar(Y),not_occurs_in(X,Y),X=Y.
unify(X,Y):-
   var(Y),nonvar(X),not_occurs_in(Y,X),X=Y.
unify(X,Y):-
   nonvar(X),nonvar(Y),atomic(X),atomic(Y),X=Y.
unify(X,Y):-
   nonvar(X),nonvar(Y),compound(X),compound(Y),term_unify(X,Y).
```

So how is `not_occurs_in/2` to be defined? Well, if the term is a variable, then the key idea is to make use of the in-built PROLOG metalogical predicate `\==` (Recall that query `X \== Y` succeeds if and only if `X` and `Y` are not identical.) This is coded in the first clause. The second clause handles the case where the term is atomic (we don't have to do anything else in such a case), and the third clause takes case of the case where the term is complex.

```
not_occurs_in(X,Term):-
   var(Term), X \== Term.
not_occurs_in(_,Term):-
   nonvar(Term),atomic(Term).
not_occurs_in(X,Term):-
   nonvar(Term),compound(Term),
   functor(Term,_,Arity),
   not_occurs_in_complex_term(Arity,X,Term).
```

The `not_occurs_in_complex_term/3` predicate recursively checks whether the variable `X` does not occur in either of the arguments of the complex term `Term`, making use of the `not_occurs_in/2` predicate as you would have expected.

```
not_occurs_in_complex_term(N,X,Y):-
   N > 0, arg(N,Y,Arg),
   not_occurs_in(X,Arg),
```

```
      M is N - 1,
      not_occurs_in_complex_term(M,X,Y).
   not_occurs_in_complex_term(0,_,_).
```

All that remains is to define the predicates which were used in the definition of `unify/2`. The predicate `term_unify/2` first checks if the terms have the same functor symbols and arity, and then uses `unify_args/3` to unify their arguments.

```
term_unify(X,Y):-
   functor(X,Functor,Arity),
   functor(Y,Functor,Arity),
   unify_args(Arity,X,Y).

unify_args(N,X,Y):-
   N > 0,
   M is N - 1,
   arg(N,X,ArgX),
   arg(N,Y,ArgY),
   unify(ArgX,ArgY),
   unify_args(M,X,Y).
unify_args(0,_,_).
```

Some versions of Prolog already support `compound/1`. If yours doesn't, then it's easy to define it.

```
compound(Term):-
   functor(Term,_,Arity), Arity > 0.
```

5.4 Free variable tableaux

Now that we know what unification is, let's see how we can use it to define a more computationally realistic signed tableaux proof system.

As we discussed earlier, the problem with our first attempt at a first-order tableaux system lay with the universal rules. Because they offered us too many ways of making substitutions (in fact, infinitely many) human insight was required to find proofs. By substituting free variables, and gradually building up a system of constraints which we will solve using unification, we hope to bypass the need for human insight. Let's work through this idea in detail, and see where it leads. As a first step, here are our new universal rules:

$$\frac{T\forall \mathrm{x}\phi}{T\phi(\mathrm{v})} \qquad\qquad \frac{F\exists \mathrm{x}\phi}{F\phi(\mathrm{v})}$$

Here $\phi(\mathrm{v})$ denotes the result of replacing all instances of the variable that the quantifier bound by a new variable v that does not occur bound anywhere in the tableau. (This restriction is simply to prevent any 'accidental bindings' taking place. In fact, every time we apply these rules, we're going to substitute a previously unused variable.)

So far so good — but a moments thought will convince the reader that our new strategy could lead to serious problems with the existential rules. Recall that the basic idea behind the existential rules was to invent a brand new name for the entity asserted to exist (we called these new names 'parameters') and to eliminate the quantifier by substituting these parameters. Unification threatens to undercut this strategy completely: the substitutions it makes may undo all our careful choices of new names! In short, we seem to be in a dilemma. The use of free variables in the universal rules is essentially a 'delaying' device; we don't want to make a real choices of what to substitute, we want unification to sort it all out for us. But if we delay in this way, how can we guarantee that unification will respect the 'new names' concept that is crucial to the existential rules?

A very clever — and very simple — idea allows us a way round this problem. We are going to substitute *structured* terms when we use the existential rules; the term structure itself will ensure that unification cannot spoil anything. To be more precise, we are to use what are known as *Skolem terms*. Without further ado, here are the new existential rules:

$$\frac{F\forall \mathrm{x}\phi}{F\phi(s(\mathrm{x}_1,\ldots,\mathrm{x}_n))} \qquad\qquad \frac{T\exists \mathrm{x}\phi}{T\phi(s(\mathrm{x}_1,\ldots,\mathrm{x}_n))}$$

Here s is a new *Skolem function symbol*, and x_1, …, x_n are all the free variables in ϕ. (If there are no free variables, s is a new *Skolem constant*.) What does this mean, and why does it work?

The basic idea is a straightforward generalisation of what we did earlier with parameters. Once again, instead of working with the first-order language built over the original vocabulary V, we are going to choose a set of new symbols SKO consisting of a countably infinite set of Skolem constants (these are essentially the same as our earlier parameters) and for every natural number n a countably infinite set of Skolem function symbols of arity n. When carrying out tableaux proofs, we won't work in the in the original first-order language, but first-order language whose vocabulary consists of all the original vocabulary V plus all the new symbols in SKO.

The symbols in SKO enable us to manufacture new names. Crucially, however, because we now have Skolem *function* symbols at our disposal, we can do something that we couldn't do with parameters: we can 'build in newness' in a way that will survive the unification operation! Look at the term the existential rules demands we substitute: $s(\mathrm{x}_1,\ldots,\mathrm{x}_n)$, where $x_1, \ldots, x_n$ are all the free variables in ϕ. Now recall our discussion of the occurs check. Quite simply, $s(x_1, \ldots, x_n)$ *cannot* unify with any of $x_1, \ldots, x_n$. Our new 'Skolem structured' term really will be new.

"Oh really?" the skeptical reader will demand. "The basic idea is clear, but the restriction simply isn't strong enough! When we used parameters, we had to pick a parameter that was brand new *to the entire tableau*. But look: the above rule only ensures that the new structured name won't unify with any of the free variables in ϕ itself! We should really insist that the new instantiated name be $s(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are all the free variables *in the entire tableau*!"

This is an intelligent observation. Our skeptical reader is certainly thinking about the rule in the right way. The suggestion is sensible: instantiations certainly *could* be carried out that way. Nonetheless, we don't *have* to (and for this we should be grateful, since it means we can substitute smaller Skolem terms). The simpler rule stated above is sound. This is not obvious, and for a proof (and further discussion) of this matter the reader should consult the sources cited in the Notes.[1]

We now have the new quantifier rules we need. Only one task remains: bringing unification into play.

We want unification to be the 'intelligence' guiding the proof search. We're going to us our quantifier rules essentially 'blindly': we're simply going to build up a set of constraints and hope that unification can do something useful with them. In particular, we hope that unification will be able to close branches for us. How could it do this for us? Let's consider an example. We shall redo our proof of

$$\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y)$$

using our new quantifier rules and unification. Here are the first 7 steps of the construction.

1	$F(\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y))$	
2	$T\exists x \forall y \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
3	$F\forall y \exists x \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
4	$T\forall y \textsc{shoots}(s_1, y)$	2, $T_{\exists}$
5	$F\exists x \textsc{shoots}(x, s_2)$	3, $F_{\forall}$
6	$T\textsc{shoots}(s_1, v_1)$	4, $T_{\forall}$
7	$F\textsc{shoots}(v_2, s_2)$	5, $F_{\exists}$

The first 5 lines are essentially identical with the previous version, save that we have used the Skolem constants s_1 and s_2 (rather than the parameters c_1 and c_2) in lines 4 and 5. (Note that we don't need to use Skolem functions as neither formula contains free

[1]Readers with some background in logic should try proving it themselves. Once you've been alerted to the fact that the simpler rule is sound, it's not that difficult to work out why. (Like so many things in logic and computer science, it's spotting the simplification in the first place that's tricky.)

variables.) The real difference occurs in lines 6 and 7. In both lines we have 'blindly' instantiated in new variables, namely v_1 and v_2. So we don't (yet) have closure.

But closure is easy to get. Consider the substitution $\{v_1/s_2, v_2/s_1\}$. If we apply this to the tableau (recall that we defined the concept of applying a substitution to a tableau in the previous section) we get the following tableau:

1	$F(\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y))$	
2	$T\exists x \forall y \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
3	$F\forall y \exists x \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
4	$T\forall y \textsc{shoots}(s_1, y)$	2, $T_{\exists}$
5	$F\exists x \textsc{shoots}(x, s_2)$	3, $F_{\forall}$
6	$T\textsc{shoots}(s_1, s_2)$	4, $T_{\forall}$
7	$F\textsc{shoots}(s_1, s_2)$	5, $F_{\exists}$

This tableau *is* closed.

This example motivates the addition of the following *Atomic MGU Closure Rule*:

> Suppose that $\mathcal{T}$ is a tableau formed from some initial tableau $\mathcal{I}$, and that some branch of $\mathcal{T}$ contains a pair of atomic formulas of the form $T(R(\tau_1, \ldots, \tau_n))$ and $F(R(\tau_1, \ldots, \tau_n))$. Then $\mathcal{T}\sigma$ is also a tableau formed from the initial tableau $\mathcal{I}$, where σ is a simultaneous mgu of τ_1 and τ_1', $\ldots$, and τ_n and τ_n'.

This rule is rather different from the other rules we've seen: it's not an extension rule, rather it's a transformation rule. It tells us that if we have tableau formed from an initial set $\mathcal{I}$, and we transform it by applying a substitution having certain properties, we obtain another tableau for the same initial set.

The basic idea guiding the choice of transformation should be clear: we want to apply substitutions that could lead to branch closure. One point, however, may be puzzling. Obviously we are interested in pairs of formulas of the form $T\phi$ and $F\phi$, but why have we restricted our attention to *atomic* formulas? The answer is: *simplicity*. In fact we could have formulated a more general rule, but if we did so we would have to state it carefully (we would need to avoid 'accidental capture' of variables) and checking for accidental capture would be computationally expensive. By restricting our attention to atomic formulas we avoid these difficulties.

And that's our free variable tableaux proof system. Frankly, it's not nearly as nice as our previous system if one wants to prove things by hand — playing with Skolem functions swiftly get unwieldy, and thinking in terms of unification is cumbersome. But it wasn't designed with the needs of humans in mind, it was designed for automation. And, as we shall now see, for this purpose it is really rather good.

5.5 A Prolog implementation

We shall now present an implementation of the free variable signed tableaux proof system. The implementation is an extension of the propositional tableaux implementation — but it can't really be described as a straightforward extension. While the basic ideas underlying signed free variable tableaux make it possible to devise practical implementations, there is more detail to take care of than in the propositional case. Moreover, the fact remains that first-order logic is undecidable, so the implementation has to defuse the (very real) threat of non-terminating tableaux constructions, and this requires a little care.

Our implementation draws on the Fitting [Fit96] implementation of free variable unsigned propositional tableaux. Roughly speaking, we take our signed propositional tableaux implementation as the starting point, and extend it to a first-order system by adopting many of Fitting's design choices. For further information, see the Notes.

Before examining the main body of the code, let's take a quick look at the main supporting routines we shall use.

First, we need predicates to handle substitution. The workhorses here are the `substitute` and `substitute_list` predicates (which may be found in `ComSemLib.pl`).

The `substitute/4` predicate 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.

```
substitute(Term,Var,A,Term):-
   Var == A,!.
substitute(_Term,_Var,A,A):-
   atomic(A),!.
substitute(_Term,_Var,A,A):-
   var(A),!.
substitute(_Term,Var,forall(Var,Y),forall(Var,Y)):-!.
substitute(Term,Var,forall(X,Y),forall(X,Z)):-
   !,substitute(Term,Var,Y,Z).
substitute(_Term,Var,exists(Var,Y),exists(Var,Y)):-!.
substitute(Term,Var,exists(X,Y),exists(X,Z)):-
   !,substitute(Term,Var,Y,Z).
substitute(Term,Var,Formula,NewFormula):-
   Formula =.. [Functor|Arglist],
   substitute_list(Term,Var,Arglist,Newarglist),
   NewFormula =.. [Functor|Newarglist].
```

The `substitute_list/4` predicate recursively applies `substitute/4` down the length of the input list to produce the output list.

```
substitute_list(_Term,_Var,[],[]).
substitute_list(Term, Var, [Head | Tail], [Newhead | Newtail]) :-
   substitute(Term, Var, Head, Newhead),
   substitute_list(Term, Var, Tail, Newtail).
```

With the substitute/4 predicate at our disposal, it is straightforward to define what we mean by instances of quantified formulas:

```
instance(t(forall(X,F)),Term,t(NewF)):-
    substitute(Term,X,F,NewF).

instance(f(exists(X,F)),Term,f(NewF)):-
    substitute(Term,X,F,NewF).

instance(t(exists(X,F)),Term,t(NewF)):-
    substitute(Term,X,F,NewF).

instance(f(forall(X,F)),Term,f(NewF)):-
    substitute(Term,X,F,NewF).
```

To handle the existential rule correctly, we need to be able generate new Skolem function symbols on demand. The following predicate does this. (The predicates it calls may be found in ComSemLib.pl.)

```
sko_fun(Varlist,Skoterm):-
   new_fun_count(N),
   Skoterm =.. [fun,N|Varlist].
```

We shall also need to know what free variables a formula contains. We associate this information explicitly with each formula. The following predicate does this.

```
notated_formula(n(Free,Formula),Free,Formula).
```

We are now ready to discuss the main code. As in the propositional case, the outermost predicate is called saturate. This recursively attempts to rule-saturate the input tableau with the aid of the expand predicate. Moreover, as in the propositional case, the base clause of saturate tests for closure via a predicate called closed.

But there is an important difference. The first-order version saturate has a second argument, a number called Qdepth (which can be read as 'quantification depth'). This number is the maximum number of times that we are allowed to apply universal rules in the course of constructing a tableau. This is the mechanism which wards off the threat of non-terminating tableaux constructions.

```
saturate(Tableau,_):-
   closed(Tableau).

saturate(OldTableau,Qdepth):-
   expand(OldTableau,Qdepth,NewTableau,NewQdepth),!,
   saturate(NewTableau,NewQdepth).
```

The `closed/1` predicate tests for branch closure using unification. It attempts to find a pair of signed formulas of the form `t(X)` and `f(X)`, and then uses the `unify` predicate discussed in the previous section to test for branch closure.

```
closed([]).

closed([Branch|Rest]):-
    member(NotatedOne,Branch),
    notated_formula(NotatedOne,_,t(X)),
    member(NotatedTwo,Branch),
    notated_formula(NotatedTwo,_,f(Y)),
    unify(X,Y),
    closed(Rest).
```

Like its propositional cousin, the `expand/4` predicate is a high level organisational predicate which works works its way recursively through the branches of the input tableau, and tries to apply the various kinds of expansion. Its two extra arguments keep track of the input and output quantification depths. Quantification depth is unaffected by all expansions save universal expansions. Each universal expansion 'uses up' one of our predetermined quota of expansions and thus decrease the quantification depth by 1.

In addition, `expand/4` performs a more interesting task. Note the use of `append` (in the clause handling universal expansions) to glue the new branch back onto the *end* of the tableau. Why do this? Essentially, it's an attempt to ensure that our predetermined quota of universal expansions are 'spread around fairly'. It would be rather silly to use up our entire quota on a single branch. By using append to 'rotate' the branches on the tableau, we ensure that every branch receives its fair share of universal expansions.

```
expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
   unary_expansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
   conjunctive_expansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
```

```
   existential_expansion(Branch,NewBranch).

expand([Branch|Tableau],OldQD,NewTableau,NewQD):-
   universal_expansion(Branch,OldQD,NewBranch,NewQD),
   append(Tableau,[NewBranch],NewTableau).

expand([Branch|Tableau],QD,[NewBranch1,NewBranch2|Tableau],QD):-
   disjunctive_expansion(Branch,NewBranch1,NewBranch2).

expand([Branch|Rest],OldQD,[Branch|Newrest],NewQD):-
   expand(Rest,OldQD,Newrest,NewQD).
```

Now for the predicates that actually carry out the expansions. The expansion predicates for unary, conjunctive, and disjunctive formulas are essentially the same as in the propositional case. The only difference is that we have to translate a notated formula into a signed formula to calculate its components, and then translate the components back again into notated components.

```
unary_expansion(Branch,[NotatedComponent|Temp]) :-
   unary(SignedFormula,Component),
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotatedComponent,Free,Component).

conjunctive_expansion(Branch,[NotatedComp1,NotatedComp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotatedComp1,Free,Comp1),
   notated_formula(NotatedComp2,Free,Comp2).

disjunctive_expansion(Branch,[NotComp1|Temp],[NotComp2|Temp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotComp1,Free,Comp1),
   notated_formula(NotComp2,Free,Comp2).
```

The most interesting are `existential_expansion/2` and `universal_expansion/4`. Note the use of `sko_fun/2` to instantiate a new Skolem term in `existential_expansion`. In `universal_expansion/4`, note that quantification depth is decremented, a new variable `V` is 'blindly' instantiated, and then — on the very last line — note the way `append` is used to replace the universal formula we have been working with back on *end* the branch. Why do this? Well, we need to replace the formula because (as we have already discussed) we will often have to re-use universal formulas. But then it makes very good sense to replace the formula at the *end* of the branch. If we leave it where it is, we run the risk of using up our entire quota of universal rule applications on this one formula. It is far more sensible to ensure fairness by 'rotating' the universal formulas on the branch, much as we rotated the branches of the tableau earlier.

```
existential_expansion(Branch,[NotatedInstance|Temp]):-
   notated_formula(NotatedFormula,Free,SignedFormula),
   existential(SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   sko_fun(Free,Term),
   instance(SignedFormula,Term,Instance),
   notated_formula(NotatedInstance,Free,Instance).

universal_expansion(Branch,OldQD,NewBranch,NewQD):-
   member(NotatedFormula,Branch),
   notated_formula(NotatedFormula,Free,SignedFormula),
   universal(SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   OldQD > 0, NewQD is OldQD - 1,
   instance(SignedFormula,V,Instance),
   notated_formula(NotatedInstance,[V|Free],Instance),
   append([NotatedInstance|Temp],[NotatedFormula],NewBranch).
```

There is one tricky point in the definition of the universal expansion predicate: the use of `member/2` in the very first lines of it. At first sight this line of code seems superfluous (note that we don't have it in the other expansion predicates). However, if we leave out the call to `member/2`, `universal/1` will pick the first of the two universal formulas format, adds it component to the beginning of the branch, and puts the universal formula back at the end of the branch (via `remove_first` and `append`). In a case where there are two different universal formulas on a branch, one of them will never be touched, and the Q-depth will be waisted on the other. Adding `member` at the beginning of the clause overcomes this problem.

Now it only remains to spell out what kinds of signed formula we are working with.

```
conjunctive(t(X & Y),t(X),t(Y)).
conjunctive(f(X v Y),f(X),f(Y)).
conjunctive(f(X > Y),t(X),f(Y)).
```

```
disjunctive(f(X & Y),f(X),f(Y)).
disjunctive(t(X v Y),t(X),t(Y)).
disjunctive(t(X > Y),f(X),t(Y)).

unary(t(~X),f(X)).
unary(f(~X),t(X)).

universal(t(forall(_,_))).
universal(f(exists(_,_))).

existential(t(exists(_,_))).
existential(f(forall(_,_))).
```

As with the propositional implementation, it's nice to have a driver that translates the formula into a notated, signed formula, and calls the `saturate` predicate. We call this predicate `valid/2`, as in our implementation for propositional tableaux. Its second additional argument is the value of Q-depth.

```
valid(X,Qdepth):-
   notated_formula(NotatedFormula,[],f(X)),
   saturate([[NotatedFormula]],Qdepth).
```

We conclude with a warning. Remember that this program can only construct those tableaux which require fewer applications of the universal rules than the user-imposed `Qdepth` limit. Thus if checking a formula with `valid/2` fails, this most emphatically does *not* mean "Not a first-order tableau theorem"! The formula in question may well be a tableaux theorem — perhaps we just set `Qdepth` to too small a value for a proof to be found. (Incidentally, while trying this program out on the kinds of formula found in introductory logic books, we usually had `Qdepth` set to 25.) On the other hand, if the program tells us that ϕ *is* a first-order tableaux theorem, this is not open to question. If a closed tableau has been formed from the initial tableau $F\phi$, then ϕ is provable, and that's that.

Notes

The human-oriented tableaux system presented at the start of the chapter is a typical first-order signed tableaux system; essentially identical systems may be found in Smullyan [Smu95] and Sundholm [Sun83], and Bell and Machover [BM77], Fitting [Fit96], Smullyan [Smu95], and Sundholm [Sun83] discuss it's unsigned counterpart. Incidentally, we really meant what we said in the text. If you want to do tableaux proofs by hand, use *this* system; its free-variable cousin wasn't designed for people but for computers.

Unification and the unification algorithm were introduced in Robinson [Rob65] for the purposes of automated theorem proving. The algorithm discussed in the text is essentially Robinson's original. The literature unification is now huge, and a wide variety of unification algorithms have been investigated. We suggest that the reader who wants to know more try Fitting [Fit96] or Apt [Apt95]. Fitting's discussion is very clear, and should be accessible to most readers of this book. Apt's discussion is more advanced, but still very approachable. He gives a nice account of idempotent substitutions, and analyses three unification algorithms, including the non-deterministic Robinson algorithm presented here. This is also a good source for further pointers to the literature.

The free variable proof system presented in the text is a signed version of the unsigned free variable tableaux system presented in Fitting [Fit96], and the reader interested in learning more about the soundness and completeness of the system is advised to start there (perhaps using Smullyan as a backup). Any skeptical readers still worried by the soundness of the existential rule should consult Hähnle and Schmitt [HS94]. A useful general source is Gallier [Gal86]. This is a detailed introduction to the theory of first-order theorem proving that covers tableaux, unification and much else besides.

To extend our propositional implementation to first-order logic, we have by and large followed the Fitting [Fit96] strategy for unsigned tableaux. In particular, our use of `Qdepth`, and the subsequent decision to use `append` to 'rotate' branches of tableaux, and universal signed formulas on branches, follows Fitting. (Incidentally, this technique has a name. We are essentially treating both tableaux and branches as *priority queues*. The idea of doing this in tableaux proofs dates back to Smullyan [Smu95]).) The implementation of `sko_fun` is taken from Fitting, and the `notated_formula` predicate is Fitting's `notation` and `fmla` predicates rolled into one.

Chapter 6

Putting It All Together

We now have some answers to the two questions with which these notes began.

1. *We can build first-order representations in a compositional way for simple natural language expressions. (Moreover, we are able to do in a way that takes scope ambiguities into account.)*
2. *We have automated the process of performing reasoning with (equality free) first-order representations.*

Along the way, we've developed a number of useful tools. Now it's time to have a little fun and bring the various bits and pieces together. As we shall see, by 'plugging together' our lambda calculus, quantifier storage, model checking and theorem proving programs we are able to define simple question handling and argumentation predicates. All that's involved is some minor extensions to our DCGs, and a few simple driver definitions.

6.1 Questions

From Chapter 2 we know how to compositionally build semantic representations for simple expressions of English, and from Chapter 1 we know how to evaluate first-order formulas in well-named models. Now, well-named models are just (rather primitive) databases. Let us extend our grammar coverage so that we can pose simple natural language database queries to them.

First we introduce some new syntactic categories: `q` for questions, and `wh` for wh-phrases. Second, we define a DCG rule that makes a question out of a wh-phrase and a verb phrase:

```
q(Sem) --> wh(SemWH), vp(SemVP), {fa(SemWH,SemVP,Sem)}.
```

(Note that this rule is the essentially the same as the rule that makes sentences out of noun phrases and verb phrases.) The third step is to add the rules for the wh-phrases:

```
wh(lambda(P,lambda(X,R))) --> [who], {fa(P,X,R)}.

wh(lambda(P,lambda(X,S & R))) --> [which], noun(SemNoun),
                                  {fa(SemNoun,X,S),fa(P,X,R)}.
```

With these entries we can handle questions like Who is a customer? or Which robber loves Honey Bunny?. The semantic representations that are derived for questions are lambda expressions, for example:

```
?- q(Sem,[who,is,a,customer],[]).

Sem = lambda(X,customer(X))
```

The final step is to write a driver that translates a question into a lambda expression, feeds it to an example model, and returns the answer. (The example models we use here are those of Chapter 1.) The driver can be coded as follows:

```
query(Question,Example,Answer):-
    q(lambda(Answer,Sem),Question,[]),
    example(Example,Model),
    sem(Sem,Model).
```

Here is an example query, evaluated to example model 1:

```
?- query([which,robber,loves,honey,bunny],1,Answer).

Answer = pumpkin
```

This driver has other uses. Since the code involved is declarative, we can generate the questions that produce a particular answer:

```
?- Answer = mia, query(Question,3,Answer).

Answer = mia,
Question = [who,loves,a,customer] ;

Answer = mia,
```

```
Question = [who,loves,vincent] ;

Answer = mia,
Question = [who,is,a,customer] ;

Answer = mia,
Question = [which,customer,loves,a,customer]
```

Exercise 6.1.1 *Extend the grammar fragment and the driver predicate in such a way that yes-no questions are covered as well.*

6.2 Basic Argumentation

We might want to let the computer decide whether a natural language argument is valid. (A natural language argument is a sequence of sentences, the last of which is called the *conclusion*, the rest, *premisses*.) Here, for example, is a simple two premiss natural language argument:

(1)

Vincent gives every woman a foot massage.
Mia is a woman.
―――――――――――――――――――
Vincent gives Mia a foot massage.

This argument is *valid*. If the premisses are true, then the conclusion is true too.

Now, we already have the basic tool needed to test arguments for validity, namely our first-order tableaux theorem prover. What we need to add is a driver which hands the theorem prover a list of formulas instead of just a single formula. The theorem prover will then attempt to make the premisses true and the conclusion false. If it can't do this — that is, if it terminates with a closed tableau — then the argument is valid.

Let's implement a predicate which does this. We'll call it `valid_argument/2`, and it will feed the theorem prover with a list of formulas, where the first formula of the list is the conclusion, and the rest of the list are premisses. We will also need a predicate `premisses/3` that turns a list of formulas in a list of signed, notated formulas.

```
valid_argument([Conclusion|Premisses],Qdepth):-
   notated_formula(NotatedConclusion,[],f(Conclusion)),
   premisses(Premisses,[],NotatedPremisses),
   saturate([[NotatedConclusion|NotatedPremisses]],Qdepth).

premisses([],N,N).
```

```
premisses([P|Premisses],SoFar,NotatedPremisses):-
   notated_formula(NotatedP,[],t(P)),
   premisses(Premisses,[NotatedP|SoFar],NotatedPremisses).
```

The only thing missing is a driver that takes a natural language argument, gives the sentences in the argument to a DCG that builds first order representations, and then feeds the representations to `valid_argument`. But this is easily done. (We allow 25 applications of the universal rules):

```
argument(P1,P2,C):-
   s(SemP1,P1,[]),
   s(SemP2,P2,[]),
   s(SemC,C,[]),
   valid_argument([SemC,SemP1,SemP2],25).
```

Time for a test session to see if this all works. Consult `lambdaPL.pl` and `freeVarTabl.pl` and pose the query

```
?- argument([vincent,gives,every,woman,a,foot,massage],
            [mia,is,a,woman],
            [vincent,gives,mia,a,foot,massage]).

yes
```

and PROLOG returns "yes". Equally well, we can try and see if the whole thing works for arguments that are *not* valid. The following, for instance, is not a valid argument.

(2)
Mia loves a man.
Every boxer is a man.
———
Mia loves a boxer.

So let's try:

```
?- argument([mia,loves,a,man],
            [every,boxer,is,a,man],
            [mia,loves,a,boxer]).

no
```

Now PROLOG gives the answer "no", as expected.

6.3 Argumentation with Scope Ambiguities

Let's have a look at the following argument:

(3)
Every man is loved by a woman.
Every boxer is a man.

A woman loves every boxer.

Is this a valid argument? The answer is: *sometime yes, sometimes no.* The premisses are ambiguous, so it depends on which reading we give them. It *is* a valid argument if the first premise is understood as: there is a woman such that she loves every man. (That is, the reading where a woman outscopes every man.)

It would be nice to be able to test whether a natural language argument is valid under some reading of the premisses. Obviously our program `lambdaPL.pl` won't help us here, since it can't cope with scope ambiguities. However, if we consult `cooperStorage.pl` instead of `lambdaPL.pl`, we get both readings for the first premise:

```
?- s(Sem,[every,man,is,loved,by,a,woman],[]).

Sem = exists(Y,woman(Y)&forall(X,man(X)>love(Y,X))) ;

Sem = forall(X,man(X)>exists(Y,woman(Y)&love(Y,X))) ;

no
```

The first one is the reading we are interested in, the second isn't. So now we are able to do test arguments with scoping ambiguities:

```
?- argument([every,man,is,loved,by,a,woman],
            [every,boxer,is,a,man],
            [a,woman,loves,every,boxer]).
```

and Prolog is able to show that this argument has a valid reading.

Exercise 6.3.1 *The* `argument/3` *driver only handles arguments with two premisses. Natural language arguments may have arbitrary numbers of premisses (though we shall assume that they only have a single conclusion). Write a new version of* `argument` *that can cope with arguments with arbitrarily many premisses.*

Exercise 6.3.2 *As we mentioned in Chapter 3, simply generating all possible permutations of quantifier scope is naive — ideally, we want world knowledge to filter out some of the possibilities. Try and think of some (very simple!) examples of world knowledge that are relevant to this problem, and then use the theorem prover to perform the required filtration.*

Notes

We won't give the reader any references here — indeed we're going swap roles and ask for some help. We would like to obtain references for programs that work more or less like the ones sketched in this chapter. That is, we're interested in hearing about examples of programs that 'plug together' parsers, theorem provers, and so on. All help greatly appreciated!

Chapter 7

Discourse Representations

In this chapter we change our underlying logical representation formalism: instead of using first-order representations we will use *Discourse Representation Structures* (DRSs). DRSs are the representations used in *Discourse Representation Theory* (DRT), one of the most influential (and certainly one of the most interesting) current approaches to the semantics of natural language. This chapter is an introduction to the basic ideas of DRT and how to implement them in PROLOG.

We proceed as follows. We first introduce DRSs, give some examples of their use, and explain how we will represent them in PROLOG. We then consider how to build DRS representations of natural language sentences. We show that this can be done quite straightforwardly by making use of the lambda calculus technology developed in Chapter 2; we call the resulting blend of ideas λ-DRT. Following this, we turn to a key topic for DRT: anaphora resolution. We give a PROLOG program which determines which antecedent NPs are accessible, and performs anaphora resolution on the basis of this information. We conclude by showing how to translate DRSs into first-order logic with equality, and implement the translation.

Three important topics are not covered in this version of the notes. First, we don't define a direct semantics for DRSs; instead we rely on the translation into first-order logic with equality to tell us what DRSs say about models. Second, we only mention in passing the standard DRT top-down DRS construction algorithm; we don't discuss it or implement it. (This is a significant omission, since the top-down construction algorithm is one of the most controversial aspects of DRT.) Third, we don't define an inference system for DRT. We plan to cover all three topics in later versions of the notes.

7.1 Discourse Representation Theory

DRSs are pairs consisting of a list of *discourse referents* and a list of *conditions*. However, some of the conditions we shall need are themselves made out of DRSs, so we must define DRSs and conditions by mutual induction. Here's how this is done. Suppose we are given a vocabulary V that contains no function symbols. Then we define:

1. If $\mathrm{x}_1, \ldots, \mathrm{x}_n$ are discourse referents ($n \geq 0$) and $\gamma_1, \ldots, \gamma_m$ ($m > 0$) are conditions then

$$\begin{array}{|l|}\hline \mathrm{x}_1, \ldots, \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}$$

   is a DRS.

2. If R is a relation symbol of arity n, and $\mathrm{x}_1,\ldots,\mathrm{x}_n$ are some discourse referents, then $\mathrm{R}(\mathrm{x}_1,\ldots,\mathrm{x}_n)$ is a condition.
3. If x is a discourse referent, and c is a constant, then x=c and c=x are conditions.
4. If x and y are discourse referents, then x=y is a condition.
5. If K_1 and K_2 are DRSs, then $\mathrm{K}_1 \rightarrow \mathrm{K}_2$ is a condition.
6. If K_1 and K_2 are DRSs, then $\mathrm{K}_1 \vee \mathrm{K}_2$ is a condition.
7. If K is a DRS, then $\neg$ K is a condition.
8. Nothing is a condition or a DRS unless it can be shown to be so using clauses 1–7.

Conditions of the form licensed by clauses 2–4 are called *basic* conditions, while those licensed by clauses 5–7 are called *complex* conditions.

Clause 1 is in many ways the key to the DRT enterprise. It specifies that a DRS is a structure (often called a 'box', for obvious reasons) consisting of two components. The top component (which may be empty) consists of a list of discourse referents; we call this component the *universe* of the DRS. The bottom component is a list of discourse conditions. Note that we stipulate that this list always contains at least one item. The conditions in this component may be either basic or complex.

How will we use this representation language? Roughly speaking, as follows. Discourse referents and constants are normally introduced by NPs. Conditions of the form defined in clause 2 are introduced by common nouns or verbs. Pronouns introduce conditions of the form defined in clauses 3 and 4. (The '=' symbol establishes the link between the discourse referent introduced by the pronoun and the discourse referent or constant introduced by its antecedent.) Universally quantifying NPs (for example, every man), and conditional sentences (that is, sentences of the form If S_1, S_2.) introduce conditions of the form defined in clause 5. Clauses 6 and 7 enable us to represent negative and disjunctive information.

What about clause 1? Such DRS are introduced by indefinite NPs. For example, the DRS for a woman snorts is:

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{snort}(\mathrm{x}) \\ \hline\end{array}$$

The box and the discourse referent x are contributed by the NP a woman. The VP snorts then fills in the box with the condition SNORT(x). Intuitively, this representation captures the fact that the sentence introduces some entity into the discourse (the discourse referent x is the formal counterpart of this entity) and asserts of this entity that it is a woman who snorts. In standard DRT, this DRS would be built using a top-down process that starts at the root node of the parse tree for a woman snorts and works it way down assigning the needed discourse referents.

Proper names also introduce discourse referents. For example, the DRS for vincent does not die is:

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \mathrm{x}{=}\textsc{vincent} \\ \neg \begin{array}{|l|}\hline \\ \hline \textsc{die}(\mathrm{x}) \\ \hline\end{array} \\ \\ \hline\end{array}$$

Again, in standard DRT, this DRS would be built using a top-down process that works its way down the parse tree assigning the needed boxes and discourse referents.

DRT has a number of interesting things to say about how pronouns and their antecedent NPs are related. One of the key ideas underlying the DRT account is the idea of *accessibility*. In DRT, the discourse referent representing a pronoun (say x) can only be resolved to the discourse referent introduced by some previously encountered NP (say y) if y is in a DRS that is accessible from the DRS containing x. Accessibility is the concept that grounds many of DRT's empirical claims about anaphora. Accessibility is a 'geometrical' concept; it has to do with the way DRSs are stacked one inside the other.

Accessibility is defined as follows. Discourse referents of DRS K_1 are accessible from DRS K_2 when K_1 *subordinates* K_2, or when K_1 *equals* K_2. Subordination is defined as follows:

> Let K_1 and K_2 be DRSs. K_1 subordinates K_2 if and only if
>
> 1. K_1 contains a condition of the form $\neg K_2$.
> 2. K_1 contains a condition of the form $K_2 \rightarrow K$, where K is some DRS.
> 3. $K_1 \rightarrow K_2$ is a condition in some DRS K.
> 4. K_1 contains a condition of the form $K_2 \vee K$ or $K \vee K_2$, for some DRS K.
> 5. Some DRS K subordinates K_2, and K_1 subordinates K.

This concept allows DRT to correctly predict that an anaphoric link is allowed between she and a woman in the first little discourse that follows, but not between she and every woman in the second:

(1) A woman snorts. She collapses.

$$\begin{array}{|l|}\hline \mathrm{x\ y} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{snorts}(\mathrm{x}) \\ \textsc{collapse}(\mathrm{y}) \\ \mathrm{y=x} \\ \hline \end{array}$$

Here x is accessible to y (they are part of the same DRS).

(2) Every woman snorts. ? She collapses.

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \hline \end{array} \rightarrow \begin{array}{|l|}\hline \\ \hline \textsc{snorts}(\mathrm{x}) \\ \hline \end{array} \\ \textsc{collapse}(\mathrm{y}) \\ \mathrm{y=?} \\ \hline \end{array}$$

There is no accessible discourse referent for y.

To conclude this section, lets consider how DRT represents the infamous "donkey sentences"; for example, if a man eats a big kahuna burger, he enjoys it.[1]

Why are such sentences considered difficult for formal semantics? Because it is not clear how the required semantic representations can be built compositionally using the traditional tools of Montague grammar. The correct semantic representation of the previous sentence is

[1] Donkey sentences are so-called because the standard examples are if a farmer owns a donkey, he beats it and every farmer who owns a donkey beats it.

$$\forall x \forall y[\textsc{man}(x) \wedge \textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \textsc{enjoy}(x,y)]$$

But how can we build up this representation using the tools developed in Chapter 2? For a start, how have the *existential* NPs a man and a big kahuna burger given rise to the *universal* quantifiers $\forall x$ and $\forall y$? And how did these two quantifiers come to take wide scope over the $\rightarrow$ contributed by the "If..." that glues the antecedent and consequent sentences together? If one attempts to extend our previous translations to cover these new sentences, one is likely to wind up generating

$$\exists x[\textsc{man}(x) \wedge \exists y[\textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y)] \rightarrow \textsc{enjoy}(x,y)]$$

Unfortunately, under the standard semantics of first-order logic, this 'representation' does not correctly capture the meaning of the sentence.

In DRT the meaning of the sentence would be represented as follows:

$$\begin{array}{|c|}\hline \\ \hline \begin{array}{|l|}\hline x\ y \\ \hline \textsc{man}(x) \\ \textsc{big\_kahuna\_burger}\ (y) \\ \textsc{eat}(x,y) \\ \hline \end{array} \rightarrow \begin{array}{|l|}\hline v\ w \\ \hline \textsc{enjoy}(v,w) \\ v=x \\ w=y \\ \hline \end{array} \\ \\ \hline \end{array}$$

This representation is natural in several ways. For a start, note that outermost level of structure is a box containing a condition of the form $K_1 \rightarrow K_2$, which clearly mirrors the fact that this sentence is a conditional. Moreover, note that the effect of each of the existential NPs in the antecedent (that is, a man and a big kahuna burger) is to contribute a discourse referent (namely x and y) just as they are supposed to. Further, the pronouns he and it in the consequent do this too (they introduce v and w). The linkage between the pronouns and their antecedents is neatly captured by the two equalities.

Does this representation really capture the meaning of the donkey sentence? And can such representations really be built systematically? The answer to both questions is "yes". As we shall later see, there is a systematic translation of the DRS language into first-order logic with equality, and applying the translation yields to the above DRS yields

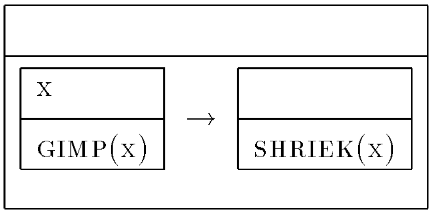
$$\forall x \forall y[\textsc{man}(x) \wedge \textsc{big\_k\_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \exists v \exists w(\textsc{enjoy}(x,y) \wedge v=x \wedge w=y)]$$

This clearly captures the intended meaning of the donkey sentence. Further, there are (at least) two methods of systematically generating such representations, namely the top-down algorithm standardly employed in DRT, or the bottom-up, Montague-flavoured, λ-employing method that we shall explore below.

7.2 Representing DRSs in Prolog

We 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.

Let's consider an example. In box notation, the DRS for every gimp shrieks is



In Prolog we represent this DRS as:

```
drs([],[drs([X],[gimp(X)]) > drs([],[shriek(X)])]).
```

Note that discourse referents are represented as Prolog variables.

Exercise 7.2.1 *What is the Prolog notation for the DRS representing* vincent does not die*?*

Exercise 7.2.2 *Define a predicate* `wfdrs/1` *that is true if and only if its argument is a well formed DRS.*

Exercise 7.2.3 *What are the DRSs for (1)* vincent dances and mia dances*, (2)* vincent cleans the back seat or jules cleans the back seat*, and (3)* if butch wins, marsellus loses*? Give the Prolog representations for all three DRSs as well.*

Exercise 7.2.4 *(This exercise is only for those readers who have had had some prior exposure to DRT. It presupposes that the reader is familiar with the semantics of DRSs, which isn't discussed in this version of the notes.) Define a simple model checker for DRSs. Do so by dividing the* `sem` *predicate defined in Chapter 1 into* `sem_drs/2` *and* `sem_condition/2`*. Here is required code for* `sem_drs/2`*:*

```
sem_drs(drs(D,C),Model):-
   constants(D),
   sem_conditions(C).
```

```
constants([]).
constants([C|Constants]):-
   constant(C),
   constants(Constants).

sem_conditions([]).
sem_conditions([C|Conditions]):-
    sem_condition(C),
    sem_conditions(Conditions).
```

Finish the program off by defining `sem_condition` *for all types of DRS condition.*

7.3 λ-DRT

In Chapter 2 we took first-order logic as our basic representation language, and allowed ourselves to mark 'missing information' with the help of the λ-operator. This enabled us to give a compositional semantics to sentences containing quantified NPs, and opened the door to more sophisticated ideas, such as the use of nested Cooper storage in Chapter 3. Let's explore a parallel strategy here: we will build up sentence representations with the help of λ, but we will take our nice new box language, rather than first-order logic, as our basic representation language. This combination will enable us to give a natural treatment of a wider range of semantic phenomena using only relatively minor adaptations of our previous work.

Actually, we shall also make use of one additional ingredient: the *merge* operation $\oplus$, which combines two DRSs by concatenating their lists of discourse referents and their lists of conditions. For example,

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{boxer}(\mathrm{x}) \\ \textsc{lose}(\mathrm{x}) \\ \hline \end{array} \oplus \begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{die}(\mathrm{y}) \\ \mathrm{y}=\mathrm{x} \\ \hline \end{array} = \begin{array}{|l|}\hline \mathrm{x}\ \mathrm{y} \\ \hline \textsc{boxer}(\mathrm{x}) \\ \textsc{lose}(\mathrm{x}) \\ \textsc{die}(\mathrm{y}) \\ \mathrm{y}=\mathrm{x} \\ \hline \end{array}$$

There are two reasons for adding merge. Perhaps the most fundamental is that it enables us to extend the idea of compositional semantics from single sentences to multi-sentence discourses in a natural way. For example, if a discourse consists of the sequence of sentences $S_1,\ldots,S_n$ and these sentences are represented by the DRSs $D_1,\ldots,D_n$ respectively, then the DRS associated with the entire discourse will be $D_1\oplus,\ldots,\oplus D_n$. In addition, merge plays a useful technical role when it comes to assigning representations to lexical items, as we shall shortly see.

What sort of λ-DRSs should be assigned to lexical items? Let's consider some examples.

Nouns are associated with the following kind of representation:

(3) boxer: λy. $\begin{array}{|l|}\hline \quad \\ \hline \text{BOXER(y)} \\ \hline \end{array}$

(4) woman: λy. $\begin{array}{|l|}\hline \quad \\ \hline \text{WOMAN(y)} \\ \hline \end{array}$

As for the determiners a and every, these are assigned the following λ-DRSs:

(5) a: λP.λQ. $\begin{array}{|l|}\hline \text{z} \\ \hline \quad \\ \hline \end{array} \oplus \text{P(z)} \oplus \text{Q(z)}$

(6) every: λP.λQ. $\begin{array}{|l|}\hline \quad \\ \hline \begin{array}{|l|}\hline \text{z} \\ \hline \quad \\ \hline \end{array} \oplus \text{P(z)} \rightarrow \text{Q(z)} \\ \hline \end{array}$

These are clearly analogous to our earlier representations. For example, in Chapter 2 we made the following assignment to every:

every: λP.λQ.$\forall$z(P(z)$\rightarrow$Q(z))

In both representations, the abstracted variable P marks the missing restriction information, while Q marks the missing nuclear scope. Moreover, the 'quantificational force' in the λ-DRS representation is obtained by using a combination of $\begin{array}{|l|}\hline \text{z} \\ \hline \quad \\ \hline \end{array}$ and $\rightarrow$, in a fashion that clearly matches the use of $\forall z$ and $\rightarrow$ in the other. Note the way that $\oplus$ is used to help state which DRS need to be 'glued' together in the course of semantic construction. (That is, $\oplus$ has a role to play in λ-DRT somewhat akin to that played by λ.)

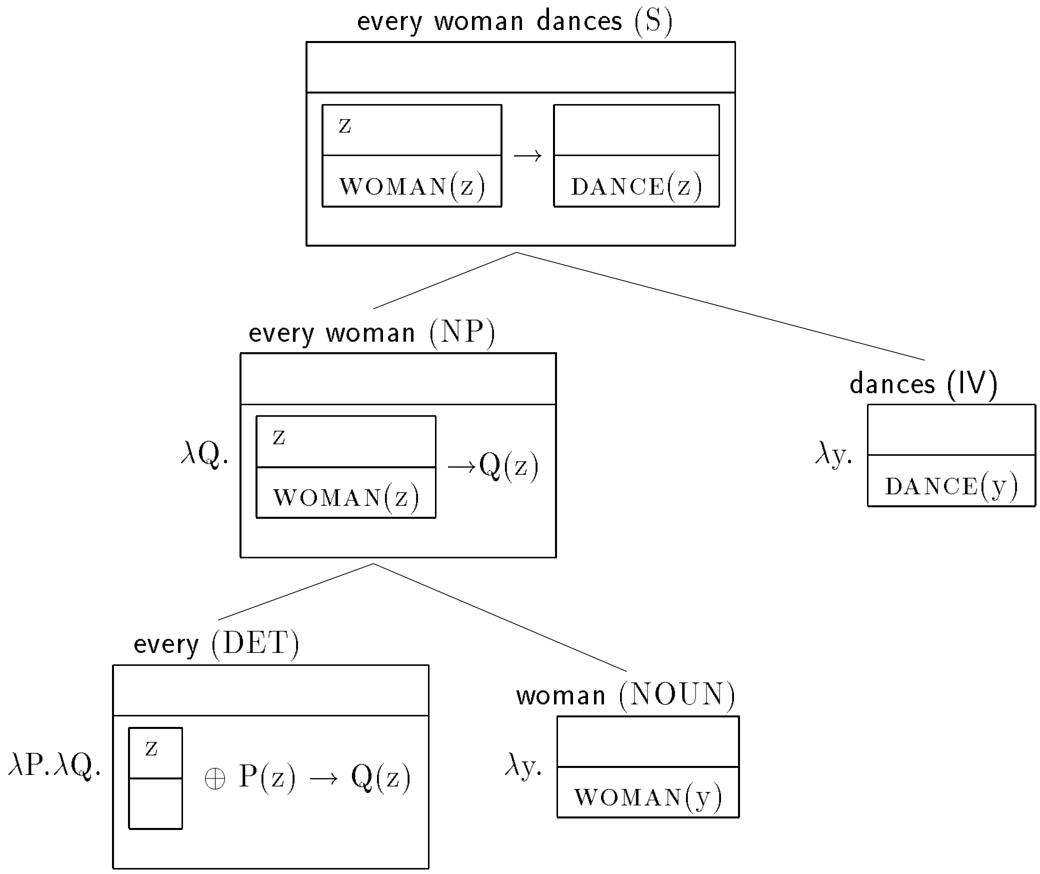
Finally, we consider intransitive and transitive verbs:

(7) dances: $\lambda y.\begin{array}{|c|}\hline \ \\ \hline \text{dance(y)} \\ \hline\end{array}$

(8) loves: $\lambda X.\lambda y.X(\lambda x.\begin{array}{|c|}\hline \ \\ \hline \text{love(y,x)} \\ \hline\end{array})$

How do we use these entries? As a sample, here's how we build a DRS for every woman dances:

(9)



Note that the fundamental mechanism that drives this construction is the one we are familiar with from Chapter 2: the lambda calculus plugs things together in the expected way. Note, however, that merge plays a supporting role. For example, the λ-DRS associated with every women in the above tree is formed from the λ-DRSs associated with every and woman by a lambda conversion (using the bound variable P), followed by another lambda conversion (using the bound variable z), followed by a merge.

Exercise 7.3.1 *Write out the steps involved in obtaining the λ-DRS associated with* every women *in the above tree.*

Exercise 7.3.2 *Build the DRS representing* a woman snorts *using the lexical representations given above.*

Exercise 7.3.3 *State the needed lexical representations, and then build the DRS representation of the "donkey sentence"* if a boxer loses, he dies.

Exercise 7.3.4 *Suppose that we tried to use the following assignment for* a *instead of the one given above:*

(10) a: $\lambda \mathrm{P}.\lambda \mathrm{Q}. \begin{array}{|l|}\hline \mathrm{x} \\ \hline \mathrm{P(x)} \\ \mathrm{Q(x)} \\ \hline \end{array}$

Why won't this work?

7.4 Implementing λ-DRT in Prolog

Actually, we practically have such an implementation already! We know how to represent DRSs in Prolog, and we already have the predicates needed for working with lambda calculus. So let's get to work and see what else we need.

Lexical entries will look like this:

```
noun(lambda(X,drs([],[boxer(X)]))) --> [boxer].

pn(lambda(P,drs([X|D],[X=mia|C]))) --> [mia],{fa(P,X,drs(D,C))}.

iv(lambda(X,drs([],[dance(X)]))) --> [dances].
```

Note that in the λ-DRS for the proper name mia we coded the merge operation directly using the list operator. But we face a problem if there are merge operations to perform which are not possible to implement in this way. This is the case for the determiner a. One option would be to make use of `append/3` in the lexicon:

```
det(lambda(P,lambda(Q,drs(D3,C3))))  --> [a],
    {fa(P,X,drs(D1,C1)),fa(Q,X,drs(D2,C2)),
     append([X|D1],D2,D3),append(C1,C2,C3)}.
```

However, this is inefficient and causes unwanted backtracking. Another possibility would be to make use of difference lists, and the reader is asked to explore this option in an exercise. Here we shall explore a more straightforward alternative. We shall decide on a PROLOG notation for the merge of two DRSs, build PROLOG representations which contain such explicit merge markings, and then define a predicate which carries out the (explicitly marked) merges required.

Suppose `B1` and `B2` are DRSs. We can represent the merge of these DRSs as `merge(B1,B2)`. Then the revised entry for `a` looks like:

```
det(lambda(P,lambda(Q,merge(drs([X|D],C),B)))) --> [a],
    {fa(P,X,drs(D,C)),fa(Q,X,B)}.
```

What happens when we query a DCG containing such entries? Here's the output that we get for a simple sentence with two indefinite noun phrases:

```
?- s(B,[a,boxer,eats,a,big,kahuna,burger],[]).

B = merge(drs([X],[boxer(X)]),
          merge(drs([Y],[big_kahuna_burger(Y)]),drs([],[eat(X,Y)])))
```

That is, we have a merge of a DRS with a DRS that is the result of another merge. This is correct, but not very readable, so let's now write a predicate `merge/2` that will actually carry out the stipulated merges. As merges can also appear in complex conditions of DRSs, this predicate must be recursive.

```
mergeDrs(drs(D,C1),drs(D,C2)):-
   mergeDrs(C1,C2).
mergeDrs(merge(B1,B2),drs(D3,C3)):-
   mergeDrs(B1,drs(D1,C1)),mergeDrs(B2,drs(D2,C2)),
   append(D1,D2,D3),append(C1,C2,C3).
mergeDrs([B1 > B2|C1],[B3 > B4|C2]):-!,
   mergeDrs(B1,B3),mergeDrs(B2,B4),mergeDrs(C1,C2).
mergeDrs([B1 v B2|C1],[B3 v B4|C2]):-!,
   mergeDrs(B1,B3),mergeDrs(B2,B4),mergeDrs(C1,C2).
mergeDrs([~ B1|C1],[~ B2|C2]):-!,
   mergeDrs(B1,B2),mergeDrs(C1,C2).
mergeDrs([C|C1],[C|C2]):-
   mergeDrs(C1,C2).
mergeDrs([],[]).
```

To make life easier, we directly incorporate the merging in the grammar rules. Adding the following rule with a new 'category' `m` (for *merge*) will do.

```
m(Drs) --> s(D), {mergeDrs(D,Drs)}.
```

Now we get 'real' DRSs from the parser. For example:

```
?- m(K,[a,boxer,dies],[]).

K = drs([X],[boxer(X),die(X)])
```

And that's all there is to it. The complete program is in appendix B.7.

Exercise 7.4.1 *Extend program* `lambdaDRT.pl` *with a treatment of restricted relative clauses.*

Exercise 7.4.2 *Extend the program to handle adjectives. (Try to handle sentences like* a rich boxer retires.*) [Easy]*

Exercise 7.4.3 *Extend the program with ditransitive verbs (for example,* to give*).*

Exercise 7.4.4 *There is another solution to the merge-problem: using difference lists. Represent DRSs as a pair of* difference lists *and reimplement* $\lambda - DRT$ *using difference lists techniques. [Hard]*

7.5 Anaphora Resolution

As we have already mentioned, in DRT, resolution of anaphora is restricted by the geometrical structure of DRSs: a discourse referent can play the role of antecedent for an anaphor only if it is accessible. In this section we develop a program for handling anaphora resolution.

As a first step, let us add grammar rules to let us handle multi-sentence discourses. Here are the two new DCG rules we need:

```
d(drs([],[])) --> [].
d(merge(Drs1,Drs2)) --> s(Drs1), d(Drs2).
```

The first rule states that a syntactic category D (for discourse) can be empty: the domain and conditions of the corresponding DRS are then empty lists. The second rule says that a discourse can consist of a sentence followed by another (possibly empty) discourse. This recursive rule allows us to parse discourses of arbitrary length. (Note that in this rule, the DRSs of the S and D clause are merged to form one big DRS representing the discourse as a whole.)

While we're at it, we may as well add a clause for conditionals:

```
d(drs([],[Drs1 > Drs2])) --> [if], s(Drs1), s(Drs2).
```

Now for the second step: handling the pronouns. Here we face a practical problem: pronouns are introduced in the lexicon, and at that point we don't know what their antecedents are. How are we to cope with this?

Our strategy will be the following: we will 'mark' the free variables the pronouns introduce, and try to unify them with referents at a later stage (for example, when we have parsed the entire discourse). So let's extend our PROLOG DRS language with the notation `alfa(X,Drs)`, where `X` is the (still unknown) value of an antecedent discourse referent. We'll call such a term an α-DRS. For example, `she snorts` is represented by the following α-DRS:

```
alfa(Y,drs([X],[X=Y,snorts(X)])).
```

Now this should remind the reader of something — namely, the use of the `lambda` predicate! In fact, we are really just re-applying the strategy that grounded our earlier work on first order representations: we are carefully marking the 'missing information' which later will later need to be filled in. Here, of course, the missing information is of a rather special type (namely anaphoric information) so we mark it with a special new `alpha` (for 'anaphoric') binder rather than with `lambda`.

Before we start thinking about how to implement anaphora resolution, let's try out the parser to work by posing the following PROLOG query:

```
?- d(B,[a,woman,snorts,she,collapses],[]).

B = merge(merge(drs([X],[woman(X)]),drs([],[snort(X)])),
          merge(merge(alfa(Z,drs([Y],[Y=Z])),
                      drs([],[collapse(Y)])),
                drs([],[])))
```

There are a lot of DRSs in the output that still have to be merged, and there is also an α-DRS introduced by the pronoun `she`. We have to resolve both kind of conditions. Note that it is sensible to first resolve the pronouns before performing the merge, as otherwise we would have to revise the `mergeDrs/2` predicate. So let's define a predicate `resolveDrs/2` that will do this for us.

Recall that a discourse referent of a DRS K_1 is accessible from DRS K_2 when K_1 subordinates K_2, or when K_1 equals K_2. (For the definition of subordination, see the beginning of this chapter.) The predicate `resolveDrs` encodes the concept of accessibility in a fairly transparent way. We make use of a second predicate `resolveDrs/3`, that has a difference list as extra argument for collecting all accessible discourse referents:

```
resolveDrs(Drs,Resolved):-
   resolveDrs(Drs,[]-_Acc,Resolved).
resolveDrs(drs(D,C1),Acc1-Acc3,drs(D,C2)):-
   append(D,Acc1,Acc2),
   resolveDrs(C1,Acc2-Acc3,C2).
resolveDrs(merge(B1,B2),Acc1-Acc3,merge(B3,B4)):-
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-Acc3,B4).
resolveDrs(alfa(Pron,B1),Acc1-Acc2,B2):-
   member(Pron,Acc1),
   resolveDrs(B1,Acc1-Acc2,B2).
resolveDrs([B1 > B2|C1],Acc1-Acc3,[B3 > B4|C2]):-!,
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-_,B4),
   resolveDrs(C1,Acc1-Acc3,C2).
resolveDrs([B1 v B2|C1],Acc1-Acc2,[B3 v B4|C2]):-!,
   resolveDrs(B1,Acc1-_,B3),
   resolveDrs(B2,Acc1-_,B4),
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([~ B1|C1],Acc1-Acc2,[~ B2|C2]):-!,
   resolveDrs(B1,Acc1-_,B2),
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([C|C1],Acc1-Acc2,[C|C2]):-
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([],Acc-Acc,[]).
```

This predicate, conforming to the nature of DRSs, is recursively defined. If an α-DRS is encountered, we use the standard `member/2` predicate (which succeeds when its first argument is a member of its second argument, which is a list) to unify an accessible discourse referents with the referent introduced by the pronoun. (Note since we make use of `member/2`, `resolveDrs` fails if there are no accessible referents, and will find more solutions on backtracking if there are more accessible referents.)

The final step is to integrate resolution in the DCG. Let's do this by introducing a new category R (for "resolved discourse"), that takes the DRS of a discourse (category D), and applies anaphora resolution and, after that, performs the merge operation. This is done by adding the following rule to the DCG.

```
r(Drs) --> d(D1), {resolveDrs(D1,D2),mergeDrs(D2,Drs)}.
```

Now for a test. Let's parse: A boxer collapses. He dies.

```
?- r(K,[a,boxer,collapses,he,dies],[]).

K = drs([X,Y],[boxer(X),collapse(X),Y=X,die(Y)])
```

In short, we no have a simple implementation of the basic ideas of DRT, including pronoun resolution. The complete program is in appendix B.7.

Exercise 7.5.1 *Try the program with* if a man eats a big kahuna burger, he enjoys it. *How many readings do you think it will generate? Why?*

Exercise 7.5.2 *Extend the program by including additional constraints on resolution, like gender.*

7.6 From Discourse Representations to First-Order Logic

We now show how to translate DRSs into formulas of first-order logic with equality. The translation is performed by a translation function t which recursively maps DRSs and conditions to formulas of first-order logic with equality. We will write $(arg)^t$ to denote the application of function t to argument arg.

First, we need to know how to translate boxes. Given a DRS whose universe is non-empty, we translate it as follows:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}\right)^t = \exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n((\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t)$$

That is, this clause maps the discourse referents in the universe of the box to existentially quantified variables, and then recursively translates the conditions. If there is only one condition γ_1 in the condition set, then the translation simplifies to $\exists \mathrm{x}_1(\gamma_1)^t$.

On the other hand, given an DRS with an empty universe, we translate it as follows:

$$\left(\begin{array}{|l|}\hline \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}\right)^t = (\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t$$

If there is only one condition γ_1 in the condition set, then the translation simplifies to $(\gamma_1)^t$.

The translation of the basic conditions couldn't be simpler: it's simply the identity translation. That is, basic conditions and atomic formulas are identical:

$$\begin{array}{rcl}(R(\mathrm{x}_1,\ldots,\mathrm{x}_n))^t & = & R(\mathrm{x}_1,\ldots,\mathrm{x}_n)\\ (\mathrm{x}_1=\mathrm{x}_2)^t & = & \mathrm{x}_1=\mathrm{x}_2\\ (\mathrm{x}=\mathrm{c})^t & = & \mathrm{x}=\mathrm{c}\\ (\mathrm{c}=\mathrm{x})^t & = & \mathrm{c}=\mathrm{x}\end{array}$$

Complex conditions formed using $\neg$ and $\vee$ are also straightforward to translate:

$$\begin{array}{rcl}(\neg\mathrm{K})^t & = & \neg(\mathrm{K})^t\\ (\mathrm{K}_1\vee\mathrm{K}_2)^t & = & (\mathrm{K}_1)^t\vee(\mathrm{K}_2)^t)\end{array}$$

That is, we simply drive the translation function t in over the connective, leaving the connective unchanged.

Now for complex conditions formed using $\rightarrow$. First, we treat the case where the antecedent DRS has a non-empty universe. In such a case we translate as follows:

$$(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n\\ \hline \gamma_1\\ \cdot\\ \cdot\\ \cdot\\ \gamma_m\\ \hline\end{array}\rightarrow\mathrm{K})^t \quad = \quad \forall\mathrm{x}_1\cdots\forall\mathrm{x}_n(((\gamma_1)^t\wedge\cdots\wedge(\gamma_m)^t)\rightarrow(\mathrm{K})^t)$$

If there is only one condition γ_1 in the condition set of the antecedent, then the translation simplifies to $\forall\mathrm{x}_1((\gamma_1)^t\rightarrow(\mathrm{K})^t)$.

On the other hand, when the antecedent DRS has an empty universe, we translate as follows:

$$(\begin{array}{|l|}\hline \\ \hline \gamma_1\\ \cdot\\ \cdot\\ \cdot\\ \gamma_m\\ \hline\end{array}\rightarrow\mathrm{K})^t \quad = \quad ((\gamma_1)^t\wedge\cdots\wedge(\gamma_m)^t)\rightarrow\mathrm{K}^t$$

Again, if there is only one condition γ_1 in the condition set of the antecedent, then the translation simplifies to $(\gamma_1)^t\rightarrow\mathrm{K}^t$.

This translation gives us a semantics for our DRS language. As should be clear from our discussion so far, DRSs built using a vocabulary V are a language for talking about models of that vocabulary. But what exactly do they say about such models? Well, we could define an interpretation for the DRS language over models of this vocabulary and in fact there are two (equivalent, but interestingly different) ways of doing this. However, another way is simply to associate each DRS with a formula of first-order logic with equality (after all, we know what these mean) and then to say that the DRS has the same meaning as the formula.

As this translation makes clear, the 'quantificational force' of the DRS language is packaged very differently from that of first order logic: it resides in the boxes (or more accurately, in the discourse referents in the universe of a box) and in the connective $\rightarrow$. The translation is straightforward to implement, and a PROLOG program called `drs2pl.pl` which does this is given in Appendix B.

Exercise 7.6.1 *Translate the DRS representing* if a man eats a big kahuna burger, he enjoys it *into first-order logic with equality.*

Notes

DRT was originally developed by Kamp [Kam81] and Heim [Hei82] (who called it "File Change Semantics"). DRT is blessed with an excellent textbook, namely Kamp and Reyle [KR93]. The book patiently describes the underlying ideas and formal details of the theory, and applies it to a wide range of natural language phenomena. The top-down DRS construction mentioned in the text is developed in detail and a direct semantics of DRT is given (the *embedding semantics* is used). The reader who wants to find out more about DRT is advised to start here. Another textbook level account well worth looking at may be found in the second volume of Gamut [Gam91b]. This account is more critical of DRT, and develops an alternative called Dynamic Predicate Logic (DPL). This uses the syntax of ordinary first order logic but gives it an interesting new semantics (the *relational semantics*). Under this semantics, first order syntax behaves like languages of DRSs equipped with the merge. The relational semantics can also be used to interpret the DRS language itself. For a more advanced discussion of DRT, the reader should consult van Eijck and Kamp [vEK97].

Many versions of DRT don't make use of a merge operator. (It's irrelevant if one employs the top-down DRS construction algorithm, only becoming important if one switches to a bottom-up, λ-based approach, as we have done here.) Merge seems to have been introduced into DRT by Zeevat [Zee89]. Zeevat used it as a tool to devise an unarguably compositional version of DRT. It occurs in a rather different guise in Groenendijk and Stokhof's [GS91] DPL, where it is called dynamic conjunction.

There are a number of "Box + λ" systems, including λ-DRT [BMM+94, KKP95], and the system of Asher [Ash93]. (Moreover, Dynamic Montague Grammar [GS90] has much in common with these systems; roughly speaking, DMG is type theoretic extension of DPL.) Combining boxes and lambdas in one formalism has proved a good approach to defining and implementing compositional semantics for large scale grammars. For example, λ-DRT is used as a semantic representation formalism for VERBMOBIL, a large spoken-language machine-translation system (see Bos *et al* [BMM+94] for an account of this application). However relatively little theoretic work has been done on such systems, and their logical properties remain poorly understood. Recent work by Muskens [Mus96] clarifies the situation somewhat. Essentially, Muskens shows how to define boxes in standard type theory. This (very readable) paper is probably the best starting point for readers interested in learning more about the theory of λ-DRT like systems.

The translation of DRSs into first-order logic with equality given in the text was taken from Kamp and Reyle [KR93]. It's probably the best known translation, but there are others; Muskens [Mus96], for example, gives an interesting alternative.

Bibliography

[Apt95] Krzysztof Apt. *From Logic Programming to Prolog*. International series in computer science. Prentice Hall, 1995.

[Ash93] N. Asher. *Reference to Abstract Objects in Discourse*. Kluwer, Dordrecht, 1993.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[Bar91] H. Barendregt. Lambda calculi with types. In D. Gabbay S. Abramsky and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 118–279. Oxford University Press, 1991.

[BM77] John Bell and Moshé Machover. *A Course in Mathematical Logic*. North Holland, 1977.

[BMM+94] Johan Bos, Elsbeth Mastenbroek, Scott McGlashan, Sebastian Millies, and Manfred Pinkal. A compositional drs-based formalism for nlp applications. In Harry Bunt, Reinhard Muskens, and Gerrit Rentier, editors, *International Workshop on Computational Semantics*, 1994. University of Tilburg.

[Bra90] Ivan Bratko. *Prolog; Programming for Artificial Intelligence*. Addison-Wesley, second edition edition, 1990.

[CGR89] Partee B. Chierchia G. and Turner R., editors. *Properties, Types and Meaning. Volume 1. Foundational Issues*. Kluwer, Dordrecht, 1989.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, third, revised and extended edition edition, 1987.

[Coo75] Robin Cooper. *Montague's Semantic Theory and Transformational Syntax*. PhD thesis, University of Massachusetts, 1975.

[Coo83] Robin Cooper. *Quantification and Syntactic Theory*. Reidel, Dordrecht, 1983.

[D'A92] Marcello D'Agostino. Are tabeleaux an improvement on truth tables? *Journal of Logic, Language and Information*, 1:235–252, 1992.

[DDS81] Wall R. Dowty D. and Peters S. *Introduction to Montague Semantics*. Reidel, Dordrecht, 1981.

[DMV97] Pereira F. Dalrymple M., Lamping J. and Saraswat V. Quantifiers, anaphora, and intentionality. *Journal of Logic, Language, and Information*, 6:219–273, 1997.

[End72] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York and London, 1972.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, second edition, 1996.

[Gal75] D. Gallin. *Intentional and Higher-Order Modal Logic*. North Holland, 1975.

[Gal86] J Gallier. *Logic for Computer Science*. Harper Row, New York, 1986.

[Gam91a] L.T.F. Gamut. *Logic, Language, and Meaning. Volume I. Introduction to Logic*. The University of Chicago Press, Chicago and London, 1991.

[Gam91b] L.T.F. Gamut. *Logic, Language, and Meaning. Volume II. Intensional Logic and Logical Grammar*. The University of Chicago Press, Chicago and London, 1991.

[GM89] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog*. Addison-Wesley Publishing Company, 1989.

[GS90] Jeroen Groenendijk and Martin Stokhof. Dynamic montague grammar. *Papers from the Second Symposium on Logic and Language*, 1990.

[GS91] Jeroen Groenendijk and Martin Stokhof. Dynamic Predicate Logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[Hei82] Irene Heim. *The Semantics of Definite and Indefinite Noun Phrases*. PhD thesis, University of Massachusetts, 1982.

[Hod83] Wilfired Hodges. Elementary predicate logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosphical Logic. Volume I.*, pages 1–131. Reidel, 1983.

[HS86] J. Hindley and J. Seldin. *Introduction to Combinators and the Lambda Calculus*. London Mathematical Society Student Texts vol. 1. Cambridge University Press, 1986.

[HS87] Jerry R. Hobbs and Stuart M. Shieber. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13(1-2):47–55, 1987.

[HS94] R Hähnle and P. Schmitt. The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning*, 13:211–221, 1994.

[Jan86a] Theo Janssen. *Foundations and Applications of Montague Grammar, Part 1: Philosophy, Framework, Computer Science.* CWI Tracts no. 19. Centre for Mathematics and Computer Science, Amsterdam, 1986.

[Jan86b] Theo Janssen. *Foundations and Applications of Montague Grammar, Part 2: Applications to Natural Language.* CWI Tracts no. 28. Centre for Mathematics and Computer Science, Amsterdam, 1986.

[Jan97] Theo Janssen. Compositionality. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 417–473. Elsevier, 1997.

[Kam81] Hans Kamp. A Theory of Truth and Semantic Representation. *Formal Methods in the Study of Language*, 1, 1981.

[Kel88] William R. Keller. Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories.* Reidel, Dordrecht, 1988.

[KKP95] Michael Kohlhase, Susanna Kuschert, and Manfred Pinkal. A Type-Theoretic Semantics for λ-DRT. In *Proceedings of the Tenth Amsterdam Colloquium*, 1995.

[KR93] Hans Kamp and Uwe Reyle. *From Discourse to Logic.* Kluwer, Dordrecht, 1993.

[Lew90] Ian Lewin. A quantifier scoping algorithm without a free variable constraint. *Coling*, pages 190–193, 1990.

[Moo89] Robert Moore. Unification-Based Semantic Interpretation. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

[Mus96] Reinhard Muskens. Combining montague semantics and discourse representation. *Linguistics and Philosophy*, 19:143–186, 1996.

[Par97] Barbara Partee. Montague grammar. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 5–91. Elsevier, 1997.

[PS87] Fernando Pereira and Stuart Shieber. *Prolog and Natural Language Analysis.* CSLI Lecture Notes 10. Chicago University Press, Stanford, 1987.

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

[Rob65] J. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–61, 1965.

[Smu95] Raymond Smullyan. *First-Order Logic.* Dover Publications, 1995. This is a reprinted and corrected version of the 1968 Springer-Verlag edition.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog; Advanced Programming Techniques.* The MIT Press, 1986.

[Sun83] Göran Sundholm. Systems of deduction. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosphical Logic. Volume I.*, pages 133–188. Reidel, 1983.

[Sun86] Göran Sundholm. Proof theory and meaning. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosphical Logic. Volume III.*, pages 471–506. Reidel, 1986.

[Tho74] Richmond Thomason. *Formal Philosophy. Selected Papers of Richard Montague.* Yale University Press, New Haven, 1974.

[Tur97] Raymond Turner. Types. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 535–586. Elsevier, 1997.

[Urq95] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1:425–467, 1995.

[vDP96] K van Deemter and S Peters, editors. *Semantic Ambiguity and Underspecification.* CSLI Lecture Notes 55. Chicago University Press, Stanford, 1996.

[vEK97] Jan van Eijk and Hans Kamp. Representing discourse in context. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 179–237. Elsevier, 1997.

[Zee89] Henk Zeevat. A compositional approach to discourse representation theory. *Linguistics and Philosophy*, 12:95–131, 1989.

Appendix A

Prolog in a Nutshell

A.1 What is Prolog?

Prolog is one of the most important programming languages in Computational Linguistics and Artificial Intelligence. Two key features distinguish Prolog from other programming languages: its *declarative* nature, and its extensive use of *unification*. Ideally, in Prolog we simply state the nature of the problem, and let the Prolog unification-driven inference engine search for a solution.

A.2 Basics

There are actually only three basic constructs in Prolog: *facts*, *rules*, and *queries*. A set of facts and rules — that is, a *knowledge base* — is all a Prolog program consists of. Facts are used to state things that are unconditionally true of the domain of interest. For example, we can state that Mia and Jody are women by putting the facts

```
woman(mia).
woman(jody).
```

in our knowledge base. Rules relate facts by logical implications. We can add to our knowledge base the conditional information that Mia plays air-guitar *if* she listens to music as follows:

```
plays_air_guitar(mia):-
   listens_to_music(mia).
```

The `:-` should be read as "if". The part on the left hand side of the `:-` is called the *head* of the rule, the part on the right hand side the *body*. (Incidentally, note that we can view a fact as a rule with an empty body. That is, we can think of facts as 'conditionals that don't have any antecedent conditions'.) The facts and rules contained in a knowledge base are called *clauses*. The collection of all rules in a knowledge base that have the same head is called a *predicate*.

Posing queries makes the Prolog inference engine try to deduce a positive answer from the information contained in the knowledge base. There are two basic circumstances under which Prolog can return a positive answer. The first, and simplest, is when the question posed is one of the facts listed in the knowledge base. The second is when Prolog can deduce the positive answer by using the deduction rule called modus ponens. That is, if `head :- body` and `body` are both in the knowledge base, then prolog can deduce that `head` is true.

Let's consider an example. We can ask Prolog whether Mia is a woman by posing the query

```
?- woman(mia).
```

and Prolog will answer "yes", since this is a fact in the knowledge base. However, if we ask whether Mia plays air-guitar by posing the query

```
?- plays_air_guitar(mia).
```

its answer is "no". First, this fact is not recorded in the knowledge base. Second, it cannot infer that Mia plays air-guitar as there is nothing in the knowledge base stating that Mia is listening to music (thus Prolog assumes that this is false) and hence we cannot make use of the only rule we have in our knowledge base. On the other hand, if the knowledge base had contained the additional fact

```
listens_to_music(mia).
```

then Prolog would have responded "yes", since it could then have deduced by modus ponens that Mia plays air-guitar.

Jumping ahead slightly, there are two things that make Prolog so powerful. The first is that it is capable of 'chaining together' uses of modus ponens. The second is that Prolog has a powerful mechanism (described later) called *unification*, which lets it handle variables. This combination of chained modus ponens and sophisticated variable handling enables it to draw far more interesting inferences than our rather trivial examples might suggest.

The comma `,` expresses logical conjunction in Prolog. We can change the rule above by adding another condition to its body:

```
plays_air_guitar(mia):-
   listens_to_music(mia),
   happy(mia).
```

Now the rule reads: Mia plays air-guitar if she listens to music, *and* if she is happy. We can also express disjunction in PROLOG. Let's change the rule to:

```
plays_air_guitar(mia):-
   listens_to_music(mia);
   happy(mia).
```

The ; should be read as *or*, so here we have a rule stating that Mia plays air-guitar if she listens to music, *or* if she is happy.

However there is another (much more commonly used) way of expressing disjunctive conditions: we simply list a number of clauses that have the same head. For example, the following two rules mean exactly the same as the previous rule:

```
plays_air_guitar(mia):-
   listens_to_music(mia).
plays_air_guitar(mia):-
   happy(mia).
```

In fact, disjunctions are almost always expressed using such multiple rules; extensive use of semicolon makes PROLOG code pretty hard to read.

A.3 Syntax

What exactly are the syntactic entities such as `woman(jody)` and `plays_air_guitar(mia)` that we use to build facts, rules, and queries? They are called *terms*, and there are three kinds of terms in PROLOG: atoms, variables, and complex terms. An atom is a sequence of characters starting with a lowercase character. A variable is also a sequence of characters, but it must start with an uppercase character or an underscore. So `mia` and `air_guitar` are atoms, while `X`, `Mayonnaise`, and `_mayonnaise` are variables.

Complex terms are build out of a *functor* and a sequence of *arguments*. The arguments are put in ordinary brackets, separated by commas, and placed after the functor. The functor *must* be an atom. That is, variables *cannot* be used as functors. On the other hand, arguments can be any kind of term. For example

```
hide(X,father(butch))
```

is a complex term. Its functor is `hide`, and it has two arguments: the variable `X`, and the complex term `father(butch)`.

The number of arguments that a complex term has is called its *arity*. For instance, `woman(mia)` is a complex term with arity 1, while `loves(vincent,mia)` is a complex term with arity 2.

Arity is important to PROLOG. PROLOG would be quite happy for us to use two predicates with the same functor but with a different number of arguments (for example, `love(vincent,mia)`, and `love(vincent,marcellus,mia)`) in the same program. However, if we insisted on doing this, PROLOG would treat the two place `love` and the three place `love` as completely different predicates.

When we need to talk about predicates and how we intend to use them (for example, in documentation) it is usual to use a suffix `/` followed by a number to indicate the predicate's arity. For example, if we were talking about the `plays_air_guitar` predicate we would write is as `plays_air_guitar/1` to indicate that it takes one argument. We make use of this convention in the book.

A.4 Unification

Variables allow us to make general statements in PROLOG. For example, to declare that every woman likes a foot massage, we add the following rules to the database:

```
like_foot_massage(X):-
   woman(X).
```

It can be read as: `X` likes a foot massage, if `X` is a woman. The nice thing about variables is that they have no fixed values. They can be instantiated with a value by the process of unification. If we pose the query

```
?- like_foot_massage(mia).
```

the variable `X` is *unified* with the atom `mia`, and PROLOG tries to prove that `woman(mia)` can be inferred from the knowledge base. Two terms are unifiable if they are the same atoms, or if one of them is a variable, or if they are both complex terms with the same functor name and arity, and all corresponding arguments unify. Unification makes terms identical. It is the heart of the engine that drives PROLOG.

PROLOG has a built-in operator for unification: the `=`. By querying the goal

```
?- X = butch.
```

the variable `X` gets unified with the atom `butch` and the goal succeeds.

A.5 Recursion

Predicates can be defined recursively. Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself. Here's an example:

```
more_expensive(X,Y):-
   costs_a_little_more(X,Y).

more_expensive(X,Y):-
   costs_a_little_more(X,Z),
   more_expensive(Z,Y).
```

The definition of the `more_expensive/2` predicate is fairly typical of the way recursive predicates are defined in PROLOG. Clearly, `more_expensive/2` is (at least partially) defined in terms of itself, as the more `more_expensive` functor occurs on both the left and right hand sides of the second rule. Note, however, that there is an 'escape' from this circularity. This is provided by the `costs_a_little_more` predicate, which occurs in both the first and second rules. (Significantly, the right hand side of the first rule makes no mention of `more_expensive`.)

Let's see how PROLOG makes use of such definitions. Suppose we had the following facts in our knowledge base:

```
costs_a_little_more(royale_with_cheese,big_kahuna_burger).

costs_a_little_more(five_dollar_shake,royale_with_cheese).
```

If we pose the query

```
?- more_expensive(five_dollar_shake,big_kahuna_burger).
```

then PROLOG goes to work as follows. First, it tries to make use of the first rule listed concerning `more_expensive`. This tells it that X is more expensive than Y if X costs a little more than Y, but as the knowledge base doesn't contain the information that a five dollar shake costs a little more than a big kahuna burger, this is no help. So, PROLOG tries to make use of the second rule. By unifying `X` with `five_dollar_shake` and `Y` with `big_kahuna_burger` it obtains the following goal:

```
?- costs_a_little_more(five_dollar_shake,Z),
   more_expensive(Z,big_kahuna_burger).
```

PROLOG deduces `more_expensive(five_dollar_shake,big_kahuna_burger)` if it can find a value for `Z` such that, firstly,

```
?- costs_a_little_more(five_dollar_shake,Z).
```

is deducible, and secondly,

```
?- more_expensive(Z,big_kahuna_burger).
```

is deducible too. But there is such a value for `Z`: `royale_with_cheese`. It is immediate that

```
?- costs_a_little_more(five_dollar_shake,royale_with_cheese).
```

will succeed, for this fact is listed in the knowledge base. Deducing

```
?- more_expensive(royale_with_cheese,big_kahuna_burger).
```

is almost as simple, for the first clause of `more_expensive/2` reduces this goal to deducing

```
?- costs_a_little_more(royale_with_cheese,big_kahuna_burger).
```

and this is a fact listed in the knowledge base.

There is one very important thing you should bear in mind when working with recursion: a recursive predicate should always have at least two clauses: a non-recursive one (the clause that stops the recursion at some point, otherwise PROLOG would never be able to finish a proof!), and one that contains the recursion. In our example, the first clause of `more_expensive/2` is the non-recursive clause, and the second clause contains the recursion. (Note that the order of these two clauses in the knowledge base is not important!)

A.6 Lists

Lists are recursive data structures in PROLOG. The recursive definition of a list runs as follows. First, the empty list is a list. Second, a complex term is a list if it consists of two items, the first of which is a term (often refered to as *first*), and the second of which is a list (often referred to as *rest* or *the rest list*).

Square brackets indicate lists. The empty list is written as `[]`. The list operator `|` separates the first item of a list from the rest list. For example, here is a list with three items:

```
[butch|[pumpkin|[marsellus|[]]]]
```

When working with lists, PROLOG always makes use of such recursive first/rest representations, and for some purposes it is important to know this. Mercifully, however, PROLOG also offers a more readable form of list representation. The same list can be declared as:

```
[butch,pumpkin,marsellus]
```

PROLOG will quite happily accept lists in this more palatable notation as input, and moreover, it does its best to uses this notation for its output.

Note that the items in a list need not only be atoms: they can be any PROLOG term, including lists. For example

```
[vincent,[honey_bunny,pumpkin],[marsellus,mia]]
```

is a perfectly good list.

Since lists are recursive data structures, most of the predicates that work on lists, are defined using recursive predicates. The simplest example is the `member/2` predicate, which is given in the program library.

A.7 Operators

Many newcomers to PROLOG find the word *operator* rather misleading, for PROLOG's operator's don't actually operate on anything, or indeed do anything at all. They are simply a notational device that PROLOG offers to represent complex terms in a more readable fashion.

For example, suppose that we want to use `not` as the functor expressing sentence negation, and `and` as the functor expressing sentence conjunction. Then the term representing Butch boxes and Vincent doesn't dance would be:

```
and(butch_boxes,not(vincent_dances)).
```

It would be nicer if we could use the more familiar notation in which the conjunction symbol stands between the two sentences it conjoins. That is, we would prefer the following representation:

```
butch_boxes and not vincent_dances
```

The following *operator definitions* let us do precisely this.

```
:- op(30,yfx,and).
:- op(20,fy,not).
```

The `and` is declared as an infix operator by `yfx` with precedence value of 30. The `y` represents an argument (in this case the left argument of &) whose precedence is lower or equal of the operator. The `x` represents an argument (the right argument of `and`) whose precedence value must be strictly lower than that of the operator (30, in this case). The `not` is defined as a prefix operator with precedence 20, and an argument that should have a precedence value lower or equal to 20.

More generally, PROLOG allows us to define our own infix, prefix and postfix operators. Operator names must be atoms. When we declare a new operator, we also have to state its precedence and those of its arguments (in order for PROLOG to disambiguate expressions with more than one operator).

A very important final remark about operators: PROLOG is featured with a set of special predefined operators, that *have* a meaning. Among these is the `=` for unification as we saw before and `\+` for negation as we will see later. Other predefined operators we use in the programs of this book are the infix operators `==` and `\==`. The goal X `==` Y succeeds if X and Y are identical terms (That is, the should have the same structure, even variables should have the same name — unification plays no role here!). On the other hand, `\==` checks whether its argument are *not* identical.

A.8 Arithmetic

PROLOG contains some built-in operators for handling integer arithmetic. These include `*`, `/` `+`, `-` (for multiplication, division, addition, and subtraction, respectively) and `>`, `<` for comparing numbers.

These symbols, however, are just ordinary PROLOG operators. That is, they are just a user friendly notation for writing arithmetic expressions: they *don't* carry out the actual arithmetic. For example, posing the query

```
?- X = 1 + 1.
```

unifies the variable `X` with the complex term `1 + 1`, not with `2`, which, for people unused to PROLOG's little ways, tends to come as a bit of a surprise.

If we want to carry out the actual arithmetic involved, we have to explicitly force evaluation by making use of the very special inbuilt 'operator' `is/2`. This calls an inbuilt mechanism which carries out the arithmetic evaluation of its second argument, and then unifies the result with its first argument.

```
?- X is 1 + 1.
X = 2
yes
```

A.9 Negation

There is no explicit negation in Prolog. Something is regarded as false if it cannot be proved. This is the so-called closed world assumption of Prolog.

Prolog does have an inbuilt mechanism for "negation as failure". That is, we can ask it whether something cannot be proved by using the built-in prefix operator `\+`. The goal `?- \+ man(jules)` succeeds if and only if the goal `?- man(jules)` fails.

Variables in negated goals are not instantiated. Therefore, the following two goals are not fully equivalent.

```
?- like_foot_massage(Who)
Who = mia
yes

?- \+ \+ like_foot_massage(Who)
yes
```

A.10 Backtracking and the Cut

When Prolog tries to prove a goal, the structure underlying it attempts can be reagarded as a tree: a tree with branches that lead (or do not lead) to possible proofs. The Prolog inference engine essentially searches for a branch that makes the main goal true. Of course it may, and usually does, find itself in a branch that does not lead to a proof (that is, a branch that *fails*). Then Prolog automatically *backtracks* to the previous node in the tree and tries another (the next) branch. (If there is no previous node, this means that the goal is not provable, and Prolog tells us "no".)

Backtracking can be forced by the user by entering a semicolon after Prolog gives a solution. This allows us to try generating more than one solution to a query. For example (reverting to our original knowledge base) we can demand that Prolog finds all the women as follows:

```
?- woman(Who).
Who = mia;
Who = jody;
```

```
no
```

By making use of the *cut* (written `!`) we can control the backtracking mechanism in a far more drastic way. The cut removes certain branches from the proof tree. If a cut is put in a clause, and PROLOG encounters it during a proof, it removes all the clauses of the same predicate that are listed further down in the knowledge base, and moreover, removes all alternative solutions to conjuncts to the left of the cut in its clause.

A.11 Built-in Predicates

There is a set of built-in predicates available in PROLOG. We briefly discuss the ones that we use in our programs. It should be noted though, that it depends on the PROLOG version that you use, which predicates are built-in.

First we have the ones for controlling output. The predicate `write/1` displays a term to the current output device, and `nl/0` creates a new line.

Then there is whole family of predicates that are used for term manipulation. With these you can break complex terms into pieces or build new ones. The predicate `functor(T,F,A)` succeeds if `F` is the functor of term `T` with arity `A`. And `arg(N,T,A)` is true if `A` is the `N`th argument of `T`. In some case we prefer to make use of the so-called "univ" predicate, weirdly written as "`=..`", that transforms a complex term into a list consisting of the functor followed by its arguments. For instance:

```
?- love(pumpkin,honey_bunny) =.. List.
List = [love,pumpkin,honey_bunny]
yes
```

For checking the types of terms: `nonvar(X)` is true if `X` is instantiated, `var(X)` is true if `X` is not instantiated. The predicate `simple/1` succeeds if its argument is either an atom or a variable, and `compound/1` if its argument is a complex term.

With `assert/1` and `retract/1` it is possible to change the knowledge base while executing a goal. The former asserts a clause to the database, the latter removes it. Many versions of PROLOG require a `dynamic` declaration of those predicates that are modified by other predicates, as we do with the counter for the skolem index in the library file.

Finally, there is some predefined stuff that gives controll over backtracking. The very rude `fail/0` predicate causes PROLOG to backtrack (this is used to generate more solutions). With `bagof/3` it is possible to generate all solutions of a specific goal, and put the instantiations of a certain variable (or complex term with variables) in a list. E.g., the goal

```
?- bagof(Who,woman(Who),Answer).
Answer = [mia,jody].
yes
```

tries to satisfies the goal in the second argument of `bagof`, and for each solution it puts the value of its first argument (`Who`) in its third argument, the list `Answer`. The predicate `setof/3` functions similarly, but it removes duplicate answers.

A.12 Difference Lists

A difference list `List1-List2` is a way of using two lists (namely, `List1` and `List2`) to represent one single list (namely, the *difference* of the two lists).

For example, the three element list `[pumpkin,butch,jimmy]` can be represented as the following difference list:

```
[pumpkin,butch,jimmy,lance]-[lance]
```

or as the following one:

```
[pumpkin,butch,jimmy]-[]
```

Indeed, there are infinitely many difference list representations of `[pumpkin,butch,jimmy]`. In all of them, the first list in the difference list representation consists of the list we are interested in (namely `[pumpkin,butch,jimmy]`) followed by some suffix, while the second list is that suffix. So a difference list is simply a pair of list structures, the second of which is a suffix from the other. We follow the usual convention of using the built-in operator `-` to group the two list together. (However, that we don't have to do this: any other operator would do.)

Let's go a step further. Suppose we take the suffix to be a variable. Then any other difference-list encoding of `[pumpkin,butch,jimmy]` is an instance of the following *most general difference list*.

```
[pumpkin,butch,jimmy|X]-X
```

In practice, we will use the term *difference list* to mean the most general difference list. The empty list is represented by the difference list `X-X`.

Why on earth should anyone want to represent lists as difference lists? There is a simple answer: efficiency. In particular, when we use the difference list representation, Prolog can perform concatenation of lists must faster. For example, the usual append/3 for normal lists is a recursively defined predicate that can be inefficient for large lists. (This is because PROLOG must recursively work its way down the entire collection of first/rest pairs.) On the other hand, when we make use of difference lists append/3 can be defined as follows.

```
append(X-Y,Y-Z,X-Z).
```

Consider how this works. Suppose we want to append [pumpkin,butch] to [jody,lance]. Then we make the following query:

```
?- append([pumpkin,butch|E]-E,[jody,lance|F]-F,A-B).
```

This causes A to unify with [pumpkin,butch|E], E with [jody,lance|F], and B with F. As a result, A-B unifies with [pumpkin,butch,jody,lance|B]-B, which is a difference list of the four items pumpkin, butch, jody, and lance.

The reader who works through this example will see that what makes difference list representations so efficient is that the suffix variable gives us direct access to the end of the list. In the conventional first/rest representation we have to work our way recursively down towards the end of the list. Difference list representations avoid this overhead. We have to pay a price for this gain in efficiency (difference list representation is less transparent) but in many circumstances this is a price worth paying.

A.13 Definite Clause Grammars

Definite clause grammars (DCGs) are the in built PROLOG mechanism for defining grammars. Actually, they are really a sytactically sugared way of working with certain difference lists. With DCGs you can kill two birds with one stone: if you define the grammar rules you'll get the parser for free!

Here is a DCG for a very small fragment of english grammar. We have defined syntactic categories s, np, vp, det, noun, and tv, standing for sentence, noun phrase, verb phrase, determiner, common noun, and transitive verb. These are also called the *non-terminal* symbols. Every rule in the DCG has a non-terminal symbol on its left hand side.

```
s --> np, vp.
np --> det, noun.
np --> [mia].
det --> [a].
noun --> [man].
noun --> [five,dollar,shake].
```

```
vp --> tv, np.
vp --> [drinks].
tv --> [loves].
```

On the right hand side of the rules, you'll either find one or more non-terminal symbols, or a terminal symbol. The terminal symbols are the lexical entries, the actual words of the language of our interest.

Now, what do these rules mean? They are very intuitive indeed. The rule `s --> np, vp` says that a syntactic category called `s`, consists of an `np`, followed by a `vp`. Similarly, according to this DCG, the category `noun` is either the string `man`, or the sequence of strings `five dollar shake` (represented in lists). And so on.

This DCG covers sentences like Mia loves a five dollar shake, A man drinks, but not: A woman loves or Mia drinks a one dollar shake.

To parse sentences, we can pose a query like:

```
?- s([mia,loves,a,five,dollar,shake],[]).
```

This goal is satisfied if the sequence of words in the first argument belongs to `s`. There is only one rule for `s` in our DCG. It says that an `s` can be replaced by an `np` followed by a `vp`. That is, we have to take some items of the input list that form a noun phrase, in such a way that the rest of the items on the list form a verb phrase. Since there is the rule

```
np --> [mia].
```

in the knowledge base we now have to proof whether

```
?- vp([loves,a,five,dollar,shake],[]).
```

which is provable indeed (we leave this is an exercise to the reader). And this is basically how parsing with DCGs takes place on the surface level. However, PROLOG doesn't use our DCG rules directly for the purpose of parsing. The DCG is translated internally into the following clauses:

```
s(A,B) :- np(A,C), vp(C,B).
np(A,B) :- det(A,C), noun(C,B).
np([mia|B],B).
det([a|B],B).
noun([man|B],B).
noun([five,dollar,shake|B],B).
vp(A,B) :- tv(A,C), np(C,B).
vp([drinks|B],B).
tv([loves|B],B).
```

The alert reader will notice that these clauses look surprisingly familiar. In fact, they are the direct encodings of difference lists (in the facts, for example noun([man|B],B)) and the appending of two difference lists (in the rules, for instance

```
s(A,B) :- np(A,C), vp(C,B).
```

can be read as: s is a difference list A-B if we can prove that is the result of appending difference list A-C to C-B).

Since, as we just noticed, DCG rules are normal PROLOG clauses, it is perfectly allowed to add arguments to the rules. Some useful stuff we can add to our grammar is information on agreement. Suppose we want to include noun phrases like all boxers in our grammar by adding entries for all and boxers:

```
det --> [all].
noun --> [boxers].
```

Be careful though! Adding these clauses make it possible to parse Mia loves all boxers, but also non-grammatical All man drinks or A boxers loves all woman. Clearly, our grammar lacks information about agreement. However, this information can be added very easily to the rules. Let's do it for the determiners and nouns first (and why not add some extra entries at the same time):

```
det(plural) --> [all].
det(singular) --> [a].
noun(singular) --> [boxer].
noun(plural) --> [boxers].
noun(singular) --> [man].
noun(plural) --> [men].
noun(singular) --> [five,dollar,shake].
noun(plural) --> [five,dollar,shakes].
```

Since det and noun have an extra argument now, all the rules that have these symbols at their right hand side (this is the np --> det, noun. rule in our current grammar) should get an extra argument as well. As can be seen from the entries above, the value of this argument is either singular or plural. Since we want the determiner to have the same agreement as the noun when they are combined to a noun phrase, we could write the following code:

```
np --> det(singular), noun(singular).
np --> det(plural), noun(plural).
```

But there is much more elegant way of encoding this. We can bring in a variable that, during parsing, gets instantiated with the agreement value of the determiner and noun, and collapse the above rules into one:

```
np --> det(Agr), noun(Agr).
```

Nevertheless, from the examples given earlier, we also want to add agreement features to noun phrases and verb phrases. We won't do explain this step by step — the principle should be clear now — but list the entire rewritten grammar including agreement, and extended with some new entries, below.

```
s --> np(Agr), vp(Agr).
np(Agr) --> det(Agr), noun(Agr).
np(singular) --> [mia].
det(plural) --> [all].
det(singular) --> [a].
noun(singular) --> [boxer].
noun(plural) --> [boxers].
noun(singular) --> [man].
noun(plural) --> [men].
noun(singular) --> [five,dollar,shake].
noun(plural) --> [five,dollar,shakes].
vp(Agr) --> tv(Agr), np(_).
vp(singular) --> [drinks].
vp(plural) --> [drink].
tv(singular) --> [loves].
tv(plural) --> [love].
```

Now look at our grammar. From an aesthetic point of view, there certainly is some space for improvement! Everything is mixed up: the rules and the lexical entries together form one chaotic DCG. Consider what these grammar would look like if we extend its coverage to a more serious fragment of english!

One way to bring in some organisation in the grammar, is to make a physical distinction between the lexicon and the grammar rules. The lexicon might be designed as follows:

```
lexicon(np,singular,mia).
lexicon(det,plural,all).
lexicon(det,singular,a).
lexicon(noun,singular,boxer).
lexicon(noun,plural,boxers).
lexicon(noun,singular,man).
```

```
lexicon(noun,plural,men).
lexicon(vp,singular,drinks).
lexicon(vp,plural,drink).
lexicon(tv,singular,loves).
lexicon(tv,plural,love).
```

That is, normal Prolog facts of the form `lexicon/3`, with the first argument stating the syntactic category, the second argument the agreement value, and the third the word itself. The only thing left to do is to make a connection between this lexicon and the grammar rules. DCGs have a neat way to do this, normal Prolog goals can be included in the rules included in curly brackets. This is the result:

```
s --> np(Agr), vp(Agr).
np(Agr) --> [X], {lexicon(np,Agr,X)}.
np(Agr) --> det(Agr), noun(Agr).
det(Agr) --> [X], {lexicon(det,Agr,X)}.
vp(Agr) --> tv(Agr), np(_).
vp(Agr) --> [X], {lexicon(vp,Agr,X)}.
tv(Agr) --> [X], {lexicon(tv,Agr,X)}.
```

Notes

In this appendix we summarized the basic concepts of Prolog. We hope it will be a handy reference, however it is *not* intended as a substitute for good introduction to Prolog. The reader who wants to learn about computational semantics, but who knows no Prolog, is strongly advised to put this book aside for a while and study one of the many excellent Prolog texts currently available. We particularly recommend the following ones. For a succinct, no-frills overview, try Clocksin and Mellish [CM87]. For a leisurely, in-depth introduction to programming in Prolog, try Bratko [Bra90]. For a more theoretically oriented introduction, try Sterling andShapiro [SS86]. Finally, for an introduction specially geared towards computational linguistics, try Pereira and Shieber [PS87].

Appendix B

Listing of Programs

This appendix includes the full program listings that are developed in this book. The programs can be used independently, but make sure you have always consulted the library file ComSemLib.pl (see B.8). This file contains definitions of general predicates and definitions of the operators that we use.

B.1 Programs for Chapter 1

```
/*************************************************************************

         name: modelPL.pl
      version: June 19, 1997
  description: Model Interpretation
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
     Vocabulary
========================================================================*/

relation(love,2).
relation(hate,2).
relation(customer,1).
relation(robber,1).
constant(mia).
constant(vincent).
constant(honey_bunny).
constant(pumpkin).

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

```
     Semantic Interpretation
========================================================================*/

sem(exists(X,Formula),Model):-
   constant(X),
   sem(Formula,Model).

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

sem(Formula1 & Formula2,Model):-
   sem(Formula1,Model),
   sem(Formula2,Model).

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

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

sem(~ Formula,Model):-
   \+ sem(Formula,Model).

sem(Formula,Model):-
   member(Formula,Model).

/*========================================================================
     Example Models
========================================================================*/

example(1,[customer(mia),customer(vincent),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).

example(2,[customer(mia),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).

example(3,[customer(mia),customer(vincent),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny),love(mia,vincent)]).

/*========================================================================
     Evaluation
========================================================================*/

evaluate(Formula,Example):-
   example(Example,Model),
```

```
   sem(Formula,Model).

/***********************************************************************

         name: wff.pl
      version: July 15, 1997
  description: well formed formulas
      authors: Patrick Blackburn & Johan Bos

***********************************************************************/

/*======================================================================
     Evaluation
======================================================================*/

%%% wff(Formula)
%%%
%%%    Checks whether Formula is a well formed formula
%%%    built out of the vocabulary.

wff(Formula):-
   var(Formula),
   !,fail.

wff(Formula):-
   functor(Formula,Symbol,Arity),
   relation(Symbol,Arity),
   good_arguments(Arity,Formula).

wff(forall(X,Formula)):-
   var(X),
   wff(Formula).

wff(exists(X,Formula)):-
   var(X),
   wff(Formula).

wff(Formula1 > Formula2):-
   wff(Formula1),wff(Formula2).

wff(Formula1 & Formula2):-
   wff(Formula1),wff(Formula2).

wff(Formula1 v Formula2):-
   wff(Formula1),wff(Formula2).

wff(~ Formula):-
   wff(Formula).
```

```
%%% good_arguments(Number,Formula)
%%%
%%%    Checks whether the arguments of Formula
%%%    are either variables or contants declared
%%%    in the vocabulary.

good_arguments(0,_).
good_arguments(N,Formula):-
   arg(N,Formula,Arg),
   first_order_term(Arg),
   M is N - 1,
   good_arguments(M,Formula).

%%% first_order_term(Term)
%%%
%%%    Term is a variable or a constant in
%%%    the vocabulary

first_order_term(X):-
   var(X).
first_order_term(X):-
   constant(X).

/*======================================================================
     Evaluation
======================================================================*/

evaluate(Formula,Example):-
   wff(Formula),
   example(Example,Model),
   sem(Formula,Model).

evaluate(Formula,_):-
    \+ wff(Formula),
    nl,write('Not a wff over the given vocabulary.'),
    nl,write('Cannot be evaluated.'),nl.
```

B.2 Programs for Chapter 2

```
/**********************************************************************

         name: experiment1.pl
      version: July 1, 1997
  description: contains the code of the first experiment of Chapter 2
      authors: Patrick Blackburn & Johan Bos
```

```
*********************************************************************/

/*======================================================================
     DCG rules
======================================================================*/

s(Sem) --> np(Sem), vp(SemVP),
           {arg(1,SemVP,X),arg(1,Sem,X),
            arg(2,Sem,Matrix),arg(2,Matrix,SemVP)}.

s(Sem) --> np(SemNP), vp(Sem), {arg(1,Sem,SemNP)}.

np(Sem) --> pn(Sem).

np(Sem) --> det(Sem), noun(SemNOUN),
            {arg(1,SemNOUN,X),arg(1,Sem,X),
             arg(2,Sem,Matrix),arg(1,Matrix,SemNOUN)}.

vp(Sem) --> iv(Sem).

vp(Sem) --> tv(SemTV), np(Sem),
            {arg(2,SemTV,X),arg(1,Sem,X),
             arg(2,Sem,Matrix),arg(2,Matrix,SemTV)}.

vp(Sem) --> tv(Sem), np(SemNP), {arg(2,Sem,SemNP)}.

/*======================================================================
     Proper Names
======================================================================*/

pn(vincent) --> [vincent].
pn(mia) --> [mia].

/*======================================================================
     Transitive Verbs
======================================================================*/

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

/*======================================================================
     Intransitive Verbs
======================================================================*/

iv(snort(_)) --> [snorts].

/*======================================================================
     Determiners
======================================================================*/

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

```
det(forall(_,_ > _)) --> [every].

/*======================================================================
     Nouns
======================================================================*/

noun(woman(_)) --> [woman].
noun(footmassage(_)) --> [foot,massage].



/**********************************************************************

         name: experiment2.pl
      version: July 2, 1997
  description: contains the code of the second experiment of Chapter 2
      authors: Patrick Blackburn & Johan Bos

**********************************************************************/

/*======================================================================
     DCG rules
======================================================================*/

s(Sem) --> np(X,SemVP,Sem), vp(X,SemVP).

np(X,Scope,Sem) --> det(X,SemNoun,Scope,Sem), noun(X,SemNoun).

np(SemPN,Sem,Sem) --> pn(SemPN).

vp(X,Sem) --> iv(X,Sem).

vp(X,Sem) --> tv(X,Y,SemTV), np(Y,SemTV,Sem).

/*======================================================================
     Proper Names
======================================================================*/

pn(vincent) --> [vincent].
pn(mia) --> [mary].

/*======================================================================
     Transitive Verbs
======================================================================*/

tv(Y,Z,love(Y,Z)) --> [loves].

/*======================================================================
     Intransitive Verbs
======================================================================*/
```

```
iv(Y,snort(Y)) --> [snorts].

/*========================================================================
     Determiners
========================================================================*/

det(X,Rest,Scope,exists(X,Rest & Scope)) --> [a].
det(X,Rest,Scope,forall(X,Rest > Scope)) --> [every].

/*========================================================================
     Nouns
========================================================================*/

noun(X,woman(X)) --> [woman].
noun(X,footmassage(X)) --> [foot,massage].

/***********************************************************************

         name: lambdaPL.pl
      version: May 15, 1997
  description: Lambda Calculus for a fragment of English
      authors: Patrick Blackburn & Johan Bos

***********************************************************************/

/*========================================================================
     Functional Application (FA)
========================================================================*/

fa(lambda(Argument,Result),Argument,Result).

/*========================================================================
     DCG rules
========================================================================*/

s(Sem) --> np(SemNP), vp(SemVP), {fa(SemNP,SemVP,Sem)}.
np(Sem) --> det(SemDet), noun(SemNoun), {fa(SemDet,SemNoun,Sem)}.
np(Sem) --> pn(Sem).
vp(Sem) --> iv(Sem).
vp(Sem) --> tv(SemTV), np(SemNP), {fa(SemTV,SemNP,Sem)}.
vp(Sem) --> [is,a], noun(Sem).
vp(Sem) --> dv(SemDV), np(SemNP1), np(SemNP2),
            {fa(SemDV,SemNP2,SemTV),fa(SemTV,SemNP1,Sem)}.

/*========================================================================
     Determiners
========================================================================*/

det(lambda(P1,lambda(P2,forall(X,R1 > R2)))) --> [every],
```

```
                                                {fa(P1,X,R1),fa(P2,X,R2)}.

det(lambda(P1,lambda(P2,exists(X,R1 & R2)))) --> [a],
                                                {fa(P1,X,R1),fa(P2,X,R2)}.

/*========================================================================
     Proper Names
========================================================================*/

pn(lambda(P,R)) --> [vincent], {fa(P,vincent,R)}.
pn(lambda(P,R)) --> [mia], {fa(P,mia,R)}.

/*========================================================================
     Nouns
========================================================================*/

noun(lambda(X,man(X))) --> [man].
noun(lambda(X,boxer(X))) --> [boxer].
noun(lambda(X,woman(X))) --> [woman].
noun(lambda(X,footmassage(X))) --> [foot,massage].

/*========================================================================
     Intransitive Verbs
========================================================================*/

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

/*========================================================================
     Transitive Verbs
========================================================================*/

tv(lambda(K1,lambda(Y,K2))) --> [loves], {fa(K1,lambda(X,love(Y,X)),K2)}.
tv(lambda(K1,lambda(Y,K2))) --> [likes], {fa(K1,lambda(X,like(Y,X)),K2)}.
tv(lambda(K1,lambda(Y,K2))) --> [is,loved,by],
            {fa(K1,lambda(X,love(X,Y)),K2)}.

/*========================================================================
     Ditransitive Verbs
========================================================================*/

dv(lambda(K1,lambda(K2,lambda(X,K4)))) --> [gives],
     {fa(K1,lambda(Z,give(X,Y,Z)),K3),fa(K2,lambda(Y,K3),K4)}.
```

B.3 Programs for Chapter 3

```
/************************************************************************
```

```
        name: quantMontague.pl
     version: June 21, 1997
 description: Montagues Rule of Quantification
     authors: Patrick Blackburn & Johan Bos

**********************************************************************/

/*=====================================================================
    Functional Application (FA)
=====================================================================*/

fa(lambda(Argument,Result),Argument,Result).

/*=====================================================================
     DCG Rules
=====================================================================*/

s([],Sem) --> s([(SemNP,I)],SemS), {fa(SemNP,lambda(I,SemS),Sem)}.
s(Q,Sem) --> np([],SemNp), vp(Q,SemVp), {fa(SemNp,SemVp,Sem)}.

np([],SemNP) --> det(SemDet), noun(SemNoun),
                 {fa(SemDet,SemNoun,SemNP)}.

np([(SemNP,I)],lambda(P,R)) --> det(SemDet), noun(SemNoun),
                            {fa(SemDet,SemNoun,SemNP),fa(P,I,R)}.
np([],Sem) --> pn(Sem).
vp([],Sem) --> iv(Sem).
vp(Q,SemVP) --> tv(SemTV), np(Q,SemNP), {fa(SemTV,SemNP,SemVP)}.


/*=====================================================================
     Determiners
=====================================================================*/

det(lambda(P1,lambda(P2,forall(X,R1 > R2)))) --> [every],
                                                 {fa(P1,X,R1),fa(P2,X,R2)}.

det(lambda(P1,lambda(P2,exists(X,R1 & R2)))) --> [a],
                                                 {fa(P1,X,R1),fa(P2,X,R2)}.

/*=====================================================================
     Proper Names
=====================================================================*/

pn(lambda(P,R)) --> [vincent], {fa(P,vincent,R)}.
pn(lambda(P,R)) --> [mia], {fa(P,mia,R)}.

/*=====================================================================
     Nouns
=====================================================================*/
```

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

/*========================================================================
     Intransitive Verbs
========================================================================*/

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

/*========================================================================
     Transitive Verbs
========================================================================*/

tv(lambda(K1,lambda(Y,K2))) --> [loves], {fa(K1,lambda(X,love(Y,X)),K2)}.


/**********************************************************************

         name: cooperStorage.pl
      version: June 21, 1997
  description: Cooper Storage Implementation
      authors: Patrick Blackburn & Johan Bos

**********************************************************************/

/*========================================================================
    Functional Application (FA)
========================================================================*/

fa([lambda(Argument,Result)|Store1],[Argument|Store2],[Result|Store]):-
     append(Store1,Store2,Store).

/*========================================================================
    Storage Operations
========================================================================*/

np_storage([lambda(Argument,Result)|Store1],
           [Argument|Store2],
           [lambda(P,R),bo(Result,X)|Store]):-
    append(Store1,Store2,Store),
    fa([P],[X],[R]).

s_retrieval([S],S).
s_retrieval([Sem|Store],S):-
   remove_from_store(bo(Q,X),Store,NewStore),
   fa([Q],[lambda(X,Sem)],[Result]),
   s_retrieval([Result|NewStore],S).

remove_from_store(X,[X|T],T).
```

```
remove_from_store(X,[Y|T],[Y|R]):-
   remove_from_store(X,T,R).

/*========================================================================
     DCG Rules
========================================================================*/

s(Sem) --> np(SemNp), vp(SemVp),
           {fa(SemNp,SemVp,SemS),s_retrieval(SemS,Sem)}.

np(Sem) --> det(SemDet), noun(SemNoun),
            {np_storage(SemDet,SemNoun,Sem)}.

np(Sem) --> pn(Sem).

noun(Sem) --> noun1(Sem).
noun(Sem) --> noun1(SemNoun), relpro(SemRP), vp(SemVP),
              {fa(SemRP,SemNoun,Sem1),fa(Sem1,SemVP,Sem)}.

vp(Sem) --> iv(Sem).
vp(Sem) --> [is,a], noun(Sem).
vp(Sem) --> tv(SemTV), np(SemNP), {fa(SemTV,SemNP,Sem)}.

/*========================================================================
     Determiners
========================================================================*/

det([lambda(P1,lambda(P2,forall(X,R1 > R2)))]) --> [every],
                     {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.

det([lambda(P1,lambda(P2,exists(X,R1 & R2)))]) --> [a],
                     {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.

/*========================================================================
     Relative Pronouns
========================================================================*/

relpro([lambda(P1,lambda(P2,lambda(X,R1 & R2)))]) --> [that],
     {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.

/*========================================================================
     Proper Names
========================================================================*/

pn(lambda(P,R)) --> [vincent], {fa([P],[vincent],[R])}.
pn(lambda(P,R)) --> [mia], {fa([P],[mia],[R])}.

/*========================================================================
     Nouns
========================================================================*/
```

```
noun1([lambda(X,man(X))]) --> [man].
noun1([lambda(X,boxer(X))]) --> [boxer].
noun1([lambda(X,woman(X))]) --> [woman].
noun1([lambda(X,piercing(X))]) --> [piercing].
noun1([lambda(X,needle(X))]) --> [needle].
noun1([lambda(X,footmassage(X))]) --> [foot,massage].

/*======================================================================
     Intransitive Verbs
======================================================================*/

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

/*======================================================================
     Transitive Verbs
======================================================================*/

tv([lambda(K1,lambda(Y,K2))]) --> [loves],
           {fa([K1],[lambda(X,love(Y,X))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [likes],
           {fa([K1],[lambda(X,like(Y,X))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [is,loved,by],
           {fa([K1],[lambda(X,love(X,Y))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [is,done,with],
           {fa([K1],[lambda(X,donewith(X,Y))],[K2])}.

/**********************************************************************

         name: nestedStorage.pl
      version: June 10, 1997
  description: Nested Cooper Storage
      authors: Patrick Blackburn & Johan Bos

**********************************************************************/

/*======================================================================
    Functional Application (FA)
======================================================================*/

fa([lambda(Argument,Result)|Store1],[Argument|Store2],[Result|Store]):-
     append(Store1,Store2,Store).

/*======================================================================
    Storage Operations
======================================================================*/

np_storage([lambda(Argument,Result)|Store1],
          [Argument|Store2],
```

```
           [lambda(P,R),bo([Result|Store],X)]):-
    append(Store1,Store2,Store),
    fa([P],[X],[R]).

s_retrieval([S],S).
s_retrieval([Sem|Store],S):-
   remove_from_store(bo(Q,X),Store,NewStore),
   fa([Q],[lambda(X,Sem)],[Result]),
   s_retrieval([Result|NewStore],S).

remove_from_store(bo(O,I),[bo([O|T1],I)|T2],T):-
   append(T1,T2,T).
remove_from_store(X,[Y|T],[Y|R]):-
   remove_from_store(X,T,R).

/*======================================================================
     DCG Rules
======================================================================*/

s(Sem) --> np(SemNp), vp(SemVp),
           {fa(SemNp,SemVp,SemS),s_retrieval(SemS,Sem)}.

np(Sem) --> det(SemDet), noun(SemNoun),
              {
                np_storage(SemDet,SemNoun,Sem)
                ;
                fa(SemDet,SemNoun,Sem)
              }.

np(Sem) --> pn(Sem).

noun(Sem) --> noun1(Sem).

noun(Sem) --> noun1(SemNoun), pp(SemPP),
              {fa(SemPP,SemNoun,Sem)}.

noun(Sem) --> noun1(SemNoun), relpro(SemRP), vp(SemVP),
              {fa(SemRP,SemNoun,Sem1),fa(Sem1,SemVP,Sem)}.

vp(Sem) --> iv(Sem).
vp(Sem) --> tv(SemTV), np(SemNP), {fa(SemTV,SemNP,Sem)}.

pp(Sem) --> prep(SemPrep), np(SemNP), {fa(SemPrep,SemNP,Sem)}.

/*======================================================================
     Determiners
======================================================================*/

det([lambda(P1,lambda(P2,forall(X,R1 > R2)))]) --> [every],
                     {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.
```

```
det([lambda(P1,lambda(P2,exists(X,R1 & R2)))]) --> [a],
                    {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.

/*======================================================================
     Relative Pronouns
======================================================================*/

relpro([lambda(P1,lambda(P2,lambda(X,R1 & R2)))]) --> [that],
     {fa([P1],[X],[R1]),fa([P2],[X],[R2])}.

/*======================================================================
     Proper Names
======================================================================*/

pn(lambda(P,R)) --> [vincent], {fa([P],[vincent],[R])}.
pn(lambda(P,R)) --> [mia], {fa([P],[mia],[R])}.

/*======================================================================
     Nouns
======================================================================*/

noun1([lambda(X,man(X))]) --> [man].
noun1([lambda(X,joke(X))]) --> [joke].
noun1([lambda(X,episode(X))]) --> [episode].
noun1([lambda(X,woman(X))]) --> [woman].
noun1([lambda(X,piercing(X))]) --> [piercing].
noun1([lambda(X,needle(X))]) --> [needle].
noun1([lambda(X,footmassage(X))]) --> [foot,massage].

/*======================================================================
     Intransitive Verbs
======================================================================*/

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

/*======================================================================
     Transitive Verbs
======================================================================*/

tv([lambda(K1,lambda(Y,K2))]) --> [loves],
           {fa([K1],[lambda(X,love(Y,X))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [likes],
           {fa([K1],[lambda(X,like(Y,X))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [told],
           {fa([K1],[lambda(X,told(X,Y))],[K2])}.
tv([lambda(K1,lambda(Y,K2))]) --> [is,done,with],
           {fa([K1],[lambda(X,donewith(X,Y))],[K2])}.
```

```
/*======================================================================
     Prepositions
======================================================================*/

prep([lambda(K1,lambda(P,lambda(Y,K3 & K2)))]) --> [in],
     {fa([K1],[lambda(X,of(Y,X))],[K2]),fa([P],[Y],[K3])}.
```

B.4 Programs for Chapter 4

```
/**********************************************************************

         name: propTabl.pl
      version: May 22, 1997
  description: Propositional Tableaux Program
      authors: Patrick Blackburn & Johan Bos

**********************************************************************/

%%% saturate(Tableau)
%%%
%%%    Expand Tableau until it is closed.

saturate(Tableau):-
   closed(Tableau).

saturate(OldTableau):-
   expand(OldTableau,NewTableau),
   saturate(NewTableau).

%%% closed(Tableau)
%%%
%%%    Every branch of Tableau contains a contradiction.

closed([]).

closed([Branch|Rest]):-
    member(t(X),Branch),
    member(f(X),Branch),
    closed(Rest).

%%% expand(Oldtableau,Newtableau)
%%%
%%%    Newtableaux is the result of applying a tableaux expansion
%%%    rule to Oldtableaux.

expand([Branch|Tableau],[NewBranch|Tableau]):-
   unary_expansion(Branch,NewBranch).
```

```
expand([Branch|Tableau],[NewBranch|Tableau]):-
   conjunctive_expansion(Branch,NewBranch).

expand([Branch|Tableau],[NewBranch1,NewBranch2|Tableau]):-
   disjunctive_expansion(Branch,NewBranch1,NewBranch2).

expand([Branch|Rest],[Branch|Newrest]):-
   expand(Rest,Newrest).

%%%    The following three expansion predicates take a SignedFormula
%%%    and Branch as input, and return the new branch(es) that result
%%%    from applying the appropriate tableau rule.

unary_expansion(Branch,[Component|Temp]) :-
   unary(SignedFormula,Component),
   remove_first(SignedFormula,Branch,Temp).

conjunctive_expansion(Branch,[Comp1,Comp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Temp).

disjunctive_expansion(Branch,[Comp1|Temp],[Comp2|Temp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Temp).

%%% conjunctive(F,Comp1,Comp2)
%%%
%%%    F is a conjunctive signed formula with components
%%%    Comp1 and Comp2

conjunctive(t(X & Y),t(X),t(Y)).
conjunctive(f(X v Y),f(X),f(Y)).
conjunctive(f(X > Y),t(X),f(Y)).

%%% disjunctive(F,Comp1,Comp2)
%%%
%%%    F is a disjunctive signed formula with components
%%%    Comp1 and Comp2

disjunctive(f(X & Y),f(X),f(Y)).
disjunctive(t(X v Y),t(X),t(Y)).
disjunctive(t(X > Y),f(X),t(Y)).

%%% unary(F,Comp)
%%%
%%%    F is a signed formula with component Comp

unary(t(~X),f(X)).
unary(f(~X),t(X)).
```

```
%%% valid(F)
%%%
%%%    Try to create a closed tableau expansion for f(F).

valid(F):-
   saturate([[f(F)]]).
```

B.5 Programs for Chapter 5

```
/*********************************************************************

         name: freeVarTabl.pl
      version: July 22, 1997
  description: Free Variable Semantic Tableaux
               Uses a number of ideas from Melvin Fitting's
               implementation of an unsigned tableaux theorem prover
               for first-order logic, in "First-Order Logic and
               Automated Theorem Proving", Second Edition (1996),
               Graduate Texts in Computer Science, Springer.
               For more details, see the Notes to Chapter 5.
      authors: Patrick Blackburn & Johan Bos

*********************************************************************/

%%% saturate(Tableau,Qdepth)
%%%
%%%    Expand Tableau until it is closed, allowing Qdepth
%%%    applications of the universal rule.

saturate(Tableau,_):-
   closed(Tableau).

saturate(OldTableau,Qdepth):-
   expand(OldTableau,Qdepth,NewTableau,NewQdepth),!,
   saturate(NewTableau,NewQdepth).

%%% closed(Tableau)
%%%
%%%    Every branch of Tableau can be made to contain a
%%%    contradiction, after a suitable free variable
%%%    substitution.

closed([]).

closed([Branch|Rest]):-
    member(NotatedOne,Branch),
```

```
    notated_formula(NotatedOne,_,t(X)),
    member(NotatedTwo,Branch),
    notated_formula(NotatedTwo,_,f(Y)),
    unify(X,Y),
    closed(Rest).

%%% expand(Oldtableau,OldQdepth,Newtableau,NewQdepth)
%%%
%%%    Newtableaux with Q-depth NewQdepth is the result of applying
%%%    a tableaux expansion rule to Oldtableaux with a Q-depth of
%%%    OldQdepth.

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
   unary_expansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
   conjunctive_expansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch|Tableau],QD):-
   existential_expansion(Branch,NewBranch).

expand([Branch|Tableau],QD,[NewBranch1,NewBranch2|Tableau],QD):-
   disjunctive_expansion(Branch,NewBranch1,NewBranch2).

expand([Branch|Tableau],OldQD,NewTableau,NewQD):-
   universal_expansion(Branch,OldQD,NewBranch,NewQD),
   append(Tableau,[NewBranch],NewTableau).

expand([Branch|Rest],OldQD,[Branch|Newrest],NewQD):-
   expand(Rest,OldQD,Newrest,NewQD).

%%%    The following three expansion predicates take a SignedFormula
%%%    and Branch as input, and return the new branch(es) that result
%%%    from applying the appropriate tableau rule.

unary_expansion(Branch,[NotatedComponent|Temp]) :-
   unary(SignedFormula,Component),
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotatedComponent,Free,Component).

conjunctive_expansion(Branch,[NotatedComp1,NotatedComp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotatedComp1,Free,Comp1),
   notated_formula(NotatedComp2,Free,Comp2).

disjunctive_expansion(Branch,[NotComp1|Temp],[NotComp2|Temp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
```

```
   notated_formula(NotatedFormula,Free,SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   notated_formula(NotComp1,Free,Comp1),
   notated_formula(NotComp2,Free,Comp2).

existential_expansion(Branch,[NotatedInstance|Temp]):-
   notated_formula(NotatedFormula,Free,SignedFormula),
   existential(SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   sko_fun(Free,Term),
   instance(SignedFormula,Term,Instance),
   notated_formula(NotatedInstance,Free,Instance).

universal_expansion(Branch,OldQD,NewBranch,NewQD):-
   OldQD > 0, NewQD is OldQD - 1,
   member(NotatedFormula,Branch),
   notated_formula(NotatedFormula,Free,SignedFormula),
   universal(SignedFormula),
   remove_first(NotatedFormula,Branch,Temp),
   instance(SignedFormula,V,Instance),
   notated_formula(NotatedInstance,[V|Free],Instance),
   append([NotatedInstance|Temp],[NotatedFormula],NewBranch).

%%% conjunctive(F,Comp1,Comp2)
%%%
%%%    F is a conjunctive signed formula with components
%%%    Comp1 and Comp2

conjunctive(t(X & Y),t(X),t(Y)).
conjunctive(f(X v Y),f(X),f(Y)).
conjunctive(f(X > Y),t(X),f(Y)).

%%% disjunctive(F,Comp1,Comp2)
%%%
%%%    F is a disjunctive signed formula with components
%%%    Comp1 and Comp2

disjunctive(f(X & Y),f(X),f(Y)).
disjunctive(t(X v Y),t(X),t(Y)).
disjunctive(t(X > Y),f(X),t(Y)).

%%% unary(F,Comp)
%%%
%%%    F is a signed formula with component Comp

unary(t(~X),f(X)).
unary(f(~X),t(X)).

%%% universal(F)
%%%
```

```
%%%    F is a universal formula.

universal(t(forall(_,_))).
universal(f(exists(_,_))).

%%% existential(F)
%%%
%%%    F is an existential formula.

existential(t(exists(_,_))).
existential(f(forall(_,_))).

%%% instance(F, Term, Ins)
%%%
%%%    F is a signed quantified formula, and Ins is the result
%%%    of removing the quantifier and replacing all free
%%%    occurrences of the quantified variable by occurrences
%%%    of Term.

instance(t(forall(X,F)),Term,t(NewF)):-
    substitute(Term,X,F,NewF).

instance(f(exists(X,F)),Term,f(NewF)):-
    substitute(Term,X,F,NewF).

instance(t(exists(X,F)),Term,t(NewF)):-
    substitute(Term,X,F,NewF).

instance(f(forall(X,F)),Term,f(NewF)):-
    substitute(Term,X,F,NewF).

%%% sko_fun(Varlist,Skoterm)
%%%
%%%    Varlist is a list of free variables, and Skoterm is
%%%    a previously unused Skolem function symbol fun(N)
%%%    applied to those free variables.

sko_fun(Varlist,Skoterm) :-
   new_fun_count(N),
   Skoterm =.. [fun,N|Varlist].

%%% notated_formula(Notated,Free,SignedFormula)
%%%
%%%   Notated is a notated formula, Free its associated free
%%%   variable list, and SignedFormula the signed formula part.

notated_formula(n(Free,Formula),Free,Formula).

%%% valid(X,Qdepth)
%%%
```

```
%%%   Try to create a tableau expansion for f(X) that is
%%%   closed allowing Qdepth applications of the universal rule.

valid(X,Qdepth):-
   notated_formula(NotatedFormula,[],f(X)),
   saturate([[NotatedFormula]],Qdepth).
```

B.6 Programs for Chapter 6

```
/*********************************************************************

         name: putTogether.pl
      version: June 12, 1997
  description: Putting it all together
      authors: Patrick Blackburn & Johan Bos

*********************************************************************/

/*====================================================================
     In order to use this program, the programs lambdaPL.pl
     and propTabl.pl should be consulted as well.
====================================================================*/


%%% valid_argument(Argument,Qdepth)
%%%
%%%   Argument is a list, of which the first element is
%%%   the conclusion, and the rest of the list the premisses.
%%%   Try to create a tableau expansion for this argument,
%%%   that is closed allowing Qdepth applications of the gamma rule.

valid_argument([Conclusion|Premises],Qdepth):-
   notated_formula(NotatedConclusion,[],f(Conclusion)),
   premises(Premises,[],NotatedPremises),
   saturate([[NotatedConclusion|NotatedPremises]],Qdepth).

%%% premises(Premises,NotatedIn,NotatedOut)
%%%
%%%   turns the formulas in Premises into notated
%%%   formulas and adds these to NotatedIn, resulting
%%%   in NotatedOut

premises([],N,N).
premises([P|Premises],SoFar,NotatedPremises):-
   notated_formula(NotatedP,[],t(P)),
   premises(Premises,[NotatedP|SoFar],NotatedPremises).
```

```
%%% argument(P1,P2,C)
%%%
%%%    Succeeds if P1, P2, and C can be parsed and
%%%    are a valid argument (allowing 25 applications
%%%    of the gamma rule).

argument(P1,P2,C):-
   s(SemP1,P1,[]),
   s(SemP2,P2,[]),
   s(SemC,C,[]),
   valid_argument([SemC,SemP1,SemP2],25).
```

B.7 Programs for Chapter 7

```
/***********************************************************************

         name: wfdrs.pl
      version: July 24, 1997
  description: From DRSs to Predicate Logic
      authors: Patrick Blackburn & Johan Bos

***********************************************************************/

%%% wfdrs(DRS)
%%%
%%%    DRS is of the form drs(Dom,Cond), Dom is a list of discourse
%%%    referents, Cond is a list of well formed DRS-conditions.

wfdrs(drs(D,C)):-
   drsReferents(D),
   drsConditions(C).

drsReferents([]).
drsReferents([Referent|Referents]):-
   var(Referent),
   drsReferents(Referents).

drsConditions([]).
drsConditions([Cond|Conditions]):-
   drsCondition(Cond),
   drsConditions(Conditions).

drsCondition(Var):-
    var(Var),!,fail.
drsCondition(BasicCond):-
    BasicCond =.. [_Pred,_Arg1|Args],
```

```
    simpleTerms([Arg1|Args]).
drsCondition(Drs1 > Drs2):-
    wfdrs(Drs1),wfdrs(Drs2).
drsCondition(~ Drs):-
    wfdrs(Drs).
drsCondition(Drs1 v Drs2):-
    wfdrs(Drs1),wfdrs(Drs2).

%%% freeVars(DRS,Referents,FreeVars)
%%%
%%%    Referents  is a list of all discourse referents in DRS,
%%%    FreeVars is the list of all free variables in DRS.

freeVars(Drs,Referents,FreeVars):-
   wfdrs(Drs),
   free(Drs,[]-Referents,[]-FreeVars).

free(drs(D,C),BoundIn-BoundOut,FreeIn-FreeOut):-
   append(D,BoundIn,Bound),
   free(C,Bound-BoundOut,FreeIn-FreeOut).

free([],B-B,Free-Free).
free([Cond|Conds],BoundIn-BoundOut,FreeIn-FreeOut):-
   free(Cond,BoundIn-BoundOut,FreeIn-FreeOut).

free(BasicCondition,BoundIn-BoundOut,FreeIn-FreeOut):-
    BasicCondition =.. [_Predicate|Args],
    simpleTerms(Args),
    freeOrBound(Args,BoundIn-BoundOut,FreeIn-FreeOut).

free(~ Drs,BoundIn-BoundIn,FreeIn-FreeOut):-
    free(Drs,BoundIn-_,FreeIn-FreeOut).

free(Drs1 > Drs2,BoundIn-BoundIn,FreeIn-FreeOut):-
    free(Drs1,BoundIn-Bound,FreeIn-Free),
    free(Drs2,Bound-_,Free-FreeOut).

free(Drs1 v Drs2,BoundIn-BoundIn,FreeIn-FreeOut):-
    free(Drs1,BoundIn-_,FreeIn-Free),
    free(Drs2,BoundIn-_,Free-FreeOut).

freeOrbound([],_,F-F).
freeOrbound([X|Rest],Bound,FreeIn-FreeOut):-
    member(Y,Bound),
    X==Y,!,
    freeOrbound(Rest,Bound,FreeIn-FreeOut).
freeOrbound([X|Rest],Bound,FreeIn-FreeOut):-
    freeOrbound(Rest,Bound,[X|FreeIn]-FreeOut).

/*****************************************************************
```

```
        name: lambdaDRT.pl
     version: July 24, 1997
 description: compositional DRS construction for a fragment of English
     authors: Patrick Blackburn & Johan Bos

*******************************************************************/

/*======================================================================
     Functional Application (FA)
======================================================================*/

fa(lambda(Argument,Result),Argument,Result).

/*======================================================================
     DRS-merge
======================================================================*/

mergeDrs(drs(D,C1),drs(D,C2)):-
   mergeDrs(C1,C2).
mergeDrs(merge(B1,B2),drs(D3,C3)):-
   mergeDrs(B1,drs(D1,C1)),mergeDrs(B2,drs(D2,C2)),
   append(D1,D2,D3),append(C1,C2,C3).
mergeDrs([B1 > B2|C1],[B3 > B4|C2]):-!,
   mergeDrs(B1,B3),mergeDrs(B2,B4),mergeDrs(C1,C2).
mergeDrs([B1 v B2|C1],[B3 v B4|C2]):-!,
   mergeDrs(B1,B3),mergeDrs(B2,B4),mergeDrs(C1,C2).
mergeDrs([~ B1|C1],[~ B2|C2]):-!,
   mergeDrs(B1,B2),mergeDrs(C1,C2).
mergeDrs([C|C1],[C|C2]):-
   mergeDrs(C1,C2).
mergeDrs([],[]).

/*======================================================================
     DCG Rules
======================================================================*/

m(Drs) --> s(D), {mergeDrs(D,Drs)}.
s(Drs) --> np(DrsNp), vp(DrsVp), {fa(DrsNp,DrsVp,Drs)}.
np(DrsNP) --> det(DrsDet), noun(DrsNoun), {fa(DrsDet,DrsNoun,DrsNP)}.
np(Drs) --> pn(Drs).
np(Drs) --> pro(Drs).
vp(Drs) --> iv(Drs).
vp(DrsVP) --> tv(DrsTV), np(DrsNP), {fa(DrsTV,DrsNP,DrsVP)}.

/*======================================================================
     Determiners
======================================================================*/

det(lambda(P,lambda(Q,merge(drs([X|D],C),B)))) --> [a],
```

```
    {fa(P,X,drs(D,C)),fa(Q,X,B)}.

det(lambda(P,lambda(Q,drs([],[drs([X|D],C) > B])))) --> [every],
    {fa(P,X,drs(D,C)),fa(Q,X,B)}.

/*========================================================================
     Proper Names
========================================================================*/

pn(lambda(P,merge(drs([X],[X=vincent]),B))) --> [vincent],{fa(P,X,B)}.
pn(lambda(P,merge(drs([X],[X=mia]),B))) --> [mia],{fa(P,X,B)}.

/*========================================================================
     Common Nouns
========================================================================*/

noun(lambda(X,drs([],[boxer(X)]))) --> [boxer].
noun(lambda(X,drs([],[man(X)]))) --> [man].
noun(lambda(X,drs([],[woman(X)]))) --> [woman].
noun(lambda(X,drs([],[gimp(X)]))) --> [gimp].
noun(lambda(X,drs([],[big_kahuna_burger(X)]))) --> [big,kahuna,burger].

/*========================================================================
     Intransitive Verbs
========================================================================*/

iv(lambda(X,drs([],[snort(X)]))) --> [snorts].
iv(lambda(X,drs([],[die(X)]))) --> [dies].
iv(lambda(X,drs([],[collapse(X)]))) --> [collapses].
iv(lambda(X,drs([],[shriek(X)]))) --> [shrieks].
iv(lambda(X,drs([],[dance(X)]))) --> [dances].

/*========================================================================
     Transitive Verbs
========================================================================*/

tv(lambda(K,lambda(Y,B))) --> [eats],
   {fa(K,lambda(X,drs([],[eat(Y,X)])),B)}.

tv(lambda(K,lambda(Y,B))) --> [enjoys],
   {fa(K,lambda(X,drs([],[enjoy(Y,X)])),B)}.

tv(lambda(K,lambda(Y,B))) --> [loves],
   {fa(K,lambda(X,drs([],[love(Y,X)])),B)}.


/*************************************************************************

        name: anaRes.pl
     version: July 25, 1997
```

```
  description: Anaphora Resolution (Extension to lambdaDRT.pl)
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
     Pronoun Resolution
========================================================================*/

resolveDrs(Drs,Resolved):-
   resolveDrs(Drs,[]-_Acc,Resolved).
resolveDrs(drs(D,C1),Acc1-Acc3,drs(D,C2)):-
   append(D,Acc1,Acc2),
   resolveDrs(C1,Acc2-Acc3,C2).
resolveDrs(merge(B1,B2),Acc1-Acc3,merge(B3,B4)):-
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-Acc3,B4).
resolveDrs(alfa(Pron,B1),Acc1-Acc2,B2):-
   member(Pron,Acc1),
   resolveDrs(B1,Acc1-Acc2,B2).
resolveDrs([B1 > B2|C1],Acc1-Acc3,[B3 > B4|C2]):-!,
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-_,B4),
   resolveDrs(C1,Acc1-Acc3,C2).
resolveDrs([B1 v B2|C1],Acc1-Acc2,[B3 v B4|C2]):-!,
   resolveDrs(B1,Acc1-_,B3),
   resolveDrs(B2,Acc1-_,B4),
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([~ B1|C1],Acc1-Acc2,[~ B2|C2]):-!,
   resolveDrs(B1,Acc1-_,B2),
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([C|C1],Acc1-Acc2,[C|C2]):-
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([],Acc-Acc,[]).

/*========================================================================
     DCG Rules
========================================================================*/

r(Drs) --> d(D1), {resolveDrs(D1,D2),mergeDrs(D2,Drs)}.
d(drs([],[])) --> [].
d(merge(Drs1,Drs2)) --> s(Drs1), d(Drs2).
d(drs([],[Drs1 > Drs2])) --> [if], s(Drs1), s(Drs2).

/*========================================================================
     Pronouns
========================================================================*/

pro(lambda(P,merge(alfa(Y,drs([X],[X=Y])),B))) --> [he],{fa(P,X,B)}.
pro(lambda(P,merge(alfa(Y,drs([X],[X=Y])),B))) --> [she],{fa(P,X,B)}.
```

```
pro(lambda(P,merge(alfa(Y,drs([X],[X=Y])),B))) --> [it],{fa(P,X,B)}.

/*************************************************************************

         name: drs2pl.pl
      version: July 25, 1997
  description: From DRSs to Predicate Logic
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

%%% drs2pl(Drs,F)
%%%
%%%    converts DRS to formula F. DRS is a term drs(Dom,Cond), where
%%%    Dom is a list of discourse referents, and Cond is a non empty
%%%    list of DRS-conditions. The first argument of drs2pl/2 should
%%%    be instantiated!

%%% if there is exactly one condition

drs2pl(drs([],[Cond]),Formula):-
    cond2pl(Cond,Formula).

%%% if there are two or more conditions

drs2pl(drs([],[Cond1,Cond2|Conds]),Formula1 & Formula2):-
    cond2pl(Cond1,Formula1),
    drs2pl(drs([],[Cond2|Conds]),Formula2).

%%% translate the discourse referents

drs2pl(drs([X|Referents],Conds),exists(X,Formula)):-
    drs2pl(drs(Referents,Conds),Formula).

%%% negation

cond2pl(~ Drs, ~ Formula):-
    drs2pl(Drs,Formula).

%%% disjunction

cond2pl(Drs1 v Drs2, Formula1 v Formula2):-
    drs2pl(Drs1,Formula1),
    drs2pl(Drs2,Formula2).

%%% implication (empty domain)

cond2pl(drs([],Conds) > Drs2, Formula1 > Formula2):-
    drs2pl(drs([],Conds),Formula1),
    drs2pl(Drs2,Formula2).
```

```
%%% implication (non-empty domain)

cond2pl(drs([X|Referents],Conds) > Drs2, forall(X,Formula)):-
    cond2pl(drs(Referents,Conds) > Drs2,Formula).

%%% basic conditions

cond2pl(BasicCondition,AtomicFormula):-
    BasicCondition =.. [_Predicate|[FirstArg|OtherArgs]],
    simpleTerms([FirstArg|OtherArgs]),
    AtomicFormula=BasicCondition.
```

B.8 The library file: ComSemLib.pl

```
/*************************************************************************

         name: ComSemLib.pl
      version: July 22, 1997
  description: general Prolog predicates and operator definitions
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
     Operators
========================================================================*/

:- op(40,yfx,>).         % implication
:- op(30,yfx,v).         % disjunction
:- op(30,yfx,&).         % conjunction
:- op(20, fy,~).          % negation

/*========================================================================
     General Predicates
========================================================================*/

%%% member(Item, List)
%%%
%%%   Item occurs in List.

member(X,[X|_]).
member(X,[_|Tail]):-
   member(X, Tail).

%%% append(X,Y,Z)
```

```
%%%
%%%    appends list X to Y resulting in list Z

append([],Y,Y).
append([X|Tail1],Y,[X|Tail2]) :-
   append(Tail1,Y,Tail2).

%%% remove(Item, List, Newlist)
%%%
%%%    Newlist is the result of removing all occurrences of
%%%    Item from List.

remove(_,[],[]).
remove(X,[X|Tail],Newtail):-
   remove(X,Tail,Newtail).
remove(X,[Head|Tail],[Head|Newtail]):-
   X \== Head,
   remove(X,Tail,Newtail).

%%% remove_first(Item,List,Newlist)
%%%
%%%    Newlist is the result of removing the first occurrence of
%%%    Item from List. Fails when Item is not member of List.

remove_first(X,[X|Tail],Tail) :- !.
remove_first(X,[Head|Tail],[Head|NewTail]):-
   remove_first(X,Tail,NewTail).

%%% simpleTerms(L)
%%%
%%%    L is a list of simple terms
%%%    (simple/1 is a built-in quintus predicate)

simpleTerms([]).
simpleTerms([X|Rest]):-
    simple(X),
    simpleTerms(Rest).

%%% unify(Term1,Term2)
%%%
%%%    Unify Term1 with Term2 including occurs check.
%%%    Adapted from "The Art of Prolog" by Sterling & Shapiro,
%%%    MIT Press 1986, page 152.

unify(X,Y):-
   var(X),var(Y),X=Y.
unify(X,Y):-
   var(X),nonvar(Y),not_occurs_in(X,Y),X=Y.
unify(X,Y):-
   var(Y),nonvar(X),not_occurs_in(Y,X),X=Y.
```

```
unify(X,Y):-
   nonvar(X),nonvar(Y),atomic(X),atomic(Y),X=Y.
unify(X,Y):-
   nonvar(X),nonvar(Y),compound(X),compound(Y),term_unify(X,Y).

%%% not_occurs_in(X,Term)
%%%
%%%    variable X does not occur in Term

not_occurs_in(X,Term):-
   var(Term), X \== Term.
not_occurs_in(_,Term):-
   nonvar(Term),atomic(Term).
not_occurs_in(X,Term):-
   nonvar(Term),compound(Term),
   functor(Term,_,Arity),not_occurs_in_complex_term(Arity,X,Term).

%%% not_occurs_in_complex_term(N,X,Term)
%%%
%%%    variable X does not occur in complex Term with arity N

not_occurs_in_complex_term(N,X,Y):-
   N > 0, arg(N,Y,Arg),
   not_occurs_in(X,Arg),
   M is N - 1,
   not_occurs_in_complex_term(M,X,Y).
not_occurs_in_complex_term(0,_,_).

%%% term_unify(Term1,Term2)
%%%
%%%    unify the complex terms Term1 and Term2

term_unify(X,Y):-
   functor(X,Functor,Arity),
   functor(Y,Functor,Arity),
   unify_args(Arity,X,Y).

unify_args(N,X,Y):-
   N > 0,
   M is N - 1,
   arg(N,X,ArgX),
   arg(N,Y,ArgY),
   unify(ArgX,ArgY),
   unify_args(M,X,Y).

unify_args(0,_,_).

%%% compound(Term)
%%%
%%%    Term is a complex term
```

```
compound(Term):-
   functor(Term,_,Arity), Arity > 0.


%%% substitute(+Term,+Variable,+Formula,-NewFormula)
%%%
%%%    NewFormula is the result of substituting
%%%    occurrences of Term for each free
%%%    occurrence of Variable in Formula.

substitute(Term,Var,A,Term):-
   Var == A,!.

substitute(_Term,_Var,A,A):-
   atomic(A),!.

substitute(_Term,_Var,A,A):-
   var(A),!.

substitute(_Term,Var,forall(A,Y),forall(A,Y)):-
   Var == A,!.

substitute(Term,Var,forall(X,Y),forall(X,Z)):-
   !,substitute(Term,Var,Y,Z).

substitute(_Term,Var,exists(A,Y),exists(A,Y)):-
   Var == A,!.

substitute(Term,Var,exists(X,Y),exists(X,Z)):-
   !,substitute(Term,Var,Y,Z).

substitute(Term,Var,Formula,NewFormula):-
   Formula =.. [Functor|Arglist],
   substitute_list(Term, Var, Arglist, Newarglist),
   NewFormula =.. [Functor|Newarglist].

substitute_list(_Term,_Var,[],[]).

substitute_list(Term,Var,[Head|Tail],[Newhead|Newtail]) :-
   substitute(Term,Var,Head,Newhead),
   substitute_list(Term,Var,Tail,Newtail).

%%% fun_count(N)
%%%
%%%    N is the current Skolem function index.
%%%    Declared as dynamic, and set to value 1.

:- dynamic(fun_count/1).
```

```
fun_count(1).

%%% new_fun_count(N)
%%%
%%%    N is the current Skolem function index, and as a
%%%    side effect, the remembered value is incremented.

new_fun_count(N):-
   fun_count(N),
   retract(fun_count(N)),
   M is N+1,
   assert(fun_count(M)).

%%% mysetof(X,Goal,Solutions)
%%%
%%%    Solutions is a list of X elements for all
%%%    solutions of Goal, after removing duplicates.

mysetof(X,Goal,Solutions):-
   bagof(X,Goal,Temp),
   remove_duplicates(Temp,Solutions).

%%% remove_duplictes(List,NewList)
%%%
%%%    NewList is List with duplicate terms removed.
%%%    Note that two different variables are considered
%%%    as duplicate!

remove_duplicates([],[]).
remove_duplicates([X|L1],L2):-
   member(X,L1),!,
   remove_duplicates(L1,L2).
remove_duplicates([X|L1],[X|L2]):-
   remove_duplicates(L1,L2).
```