Representation and Inference for Natural Language

A First Course in Computational Semantics

Volume II

Working with Discourse Representation Structures

Patrick Blackburn & Johan Bos

September 3, 1999

Contents

Preface

In Volume II 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. In spite of this change, we will be able to reuse the grammars and tools developed in Volume I without difficulty. We then investigate some interesting semantic phenomena, namely pronoun resolution and presupposition projection, and integrate—at least partially—our work on inference with our work on semantic construction. Chapter by chapter, Volume II looks like this:

> **Chapter 1. Discourse Representation Theory.** This chapter introduces DRT and prepares the way for our later computational work. We discuss the notion of context change potential, define DRS languages, discuss accessibility and the standard DRS construction algorithm, give two (equivalent) semantics for DRS languages, and show how to translate DRSs into first-order logic. We then implement a simple model checker, and a DRS to first-order logic compiler.

> **Chapter 2. Building Discourse Representations.** We first examine three distinct techniques for constructing the DRSs. First, we discuss threading, an elegant and efficient approach that is essentially the standard DRS construction algorithm viewed declaratively. Second, we show how straightforward it is to transplant the lambda calculus technology developed in Volume I to the setting of DRT; we call the resulting blend λ-DRT. Third, we integrate DRS construction with underspecification. Finally, we discuss the merging of DRSs.

> **Chapter 3. Pronoun Resolution.** Here we turn to a key topic for DRT: pronoun resolution. We develop a Prolog program which determines which antecedent NPs are accessible, and performs pronoun resolution on the basis of this information. We will then extend this program to deal with reflexive pronouns. Finally, we use a focusing algorithm to get—in case ambiguities arise—a preferred reading.

Chapter 4. Presupposition Resolution. What exactly are presuppositions, and how can we solve the projection problem for presupposition? In this chapter we discuss and implement an elegant DRT-based approach to these problems due to Rob van der Sandt. This approach treats presuppositions as anaphors which may be accommodated, offering a novel perspective on presupposition projection. In this chapter we show how the pronoun resolution techniques introduced in Chapter 3 can be extended to handle presupposition accommodation.

Chapter 5. Acceptability Constraints. The algorithms we have supplied for dealing with pronouns and presupposition are genuinely over-generating. But on the basis of a simple maxims of conversations, such as consistency and informativity, some of them can be ruled out. In this chapter we show how to do this.

Chapter 6. Putting It All Together. Here we combine the techniques we have acquired in this book. We'll try to put together a system that integrates our techniques for dealing with scope, anaphora, and presupposition. We show that *hole semantics* developed in Volume I adapts straightforwardly to DRT, thus giving us a mechanism for coping with scope ambiguitiesin DRT.

Each chapter concludes with two sections: *Software Summary* lists the programs developed in the chapter, and *Notes* lists references the reader may find helpful, and discusses more advanced topics.

This book developed out of material for a course on Computational Semantics we regularly offer at the Department of Computational Linguistics, University of the Saarland. We also taught a preliminary version (essentially the material that now makes up Volume I) as an introductory course at ESSLLI'97, the *European Summer School in Logic, Language and Information* held at Aix-en-Provence, France in August 1997. When designing these courses, we found no single source which contained all the material we wanted to present. At that time, the only notes solely devoted to computational semantics we knew of were Cooper et al. 1993. These notes, which we recommend to our readers, were developed at the University of Edinburgh, and are probably the first systematic introduction to modern computational semantics. Like the present book they are Prolog based, and cover some of the same ground using interestingly different tools and techniques. However we wanted to teach the subject in a way that emphasized such ideas as inference, underspecification, and architectural issues. This led us to a first version of the book, which was heavily influenced by Pereira and Shieber 1987 for semantic construction, Fitting 1996 and Smullyan 1995 for tableaux systems, and Kamp and Reyle 1993 for DRT. Since then, the project has taken on a life of its own, and grown in a variety of (often unexpected) directions. Both the code and the text has been extensively rewritten and we are now (we hope!) in the final stretch of producing the kind of introduction to computational semantics that we wanted all along.

Acknowledgments

We would like to thank Manfred Pinkal and all our colleagues at the Department of Computational Linguistics, University of Saarland, Saarbrücken, Germany. Thanks to Manfred, the department has become an extremely stimulating place for working on computational semantics; it was the ideal place to write this book.

Johan Bos would like to thank his colleagues in the Verb*mobil* Project (grant 01 IV 701 R4 and 01 IV 701 N3) at IMS-Stuttgart, TU-Berlin, CSLI Stanford, the DFKI, and the Department of Computational Linguistics at Saarbrücken, and those in the Trindi Project (Task Oriented Instructional Dialogue; LE4-8314) at the University of Gothenburg, SRI Cambridge, University of Edinburgh, and Xerox Grenoble.

Conversations with Paul Dekker, Andreas Franke, Martin Kay, Hans Kamp, Michael Kohlhase, Karsten Konrad, Christof Monz, Reinhard Muskens, Maarten de Rijke, and Henk Zeevat were helpful in clarifying our goals. We're grateful to David Beaver, Judith Baur, Björn Gambäck, Ewan Klein, Emiel Krahmer, Rob van der Sandt, and Frank Schilder for their comments on an early draft of this book. We're also very grateful to our students both at Saarbrücken and at ESSLLI, who has provided us with invaluable feedback. In particular we would like to thank Aljoscha Burchardt, Gerd Fliedner, Malte Gabsdil, Kristina Striegnitz, Stefan Thater, and Stephan Walter. Patrick Blackburn would like to thank Aravind Joshi and the staff of Institute for Research in Cognitive Science at the University of Pennsylvania for their hospitality while the Montreal version was being prepared.

Finally, we are deeply grateful to Robin Cooper, who taught a course based on the previous draft of this book, and provided us with extremely detailed feedback on what worked and what didn't. His comments have greatly improved the book.

Patrick Blackburn and Johan Bos,
Computerlinguistik, Universität des Saarlandes,
September 1999.

Chapter 1

Discourse Representation Theory

In Part I we used first-order formulas as our underlying representations; in Part II we switch to *Discourse Representation Structures (DRSs)*, the representations used in *Discourse Representation Theory (DRT)*. DRSs will enable us to encode the information contained in multi-sentence discourses, and to do so in a way that yields a natural treatment of such phenomena as anaphoric pronouns and presuppositions. As we shall see, DRT is compatible with a wide range of implementation strategies, and we will be able to reuse much of our work from Part I with little or no fuss.

This chapter introduces the basic ideas of DRT. The first section gives a swift overview of DRT that encourages the reader to think of DRSs (or *boxes*) as pictures. We show that it is useful to view discourse comprehension as a process of constructing a picture of the changing context, define *DRS languages*, informally sketch the *embedding semantics*, introduce the concept of *accessibility*, and discuss the *standard construction algorithm*. The remaining sections then explore semantics of DRS languages in more detail, both theoretically and computationally. First, we present two alternative interpretations of DRS languages: the embedding semantics, which emphasizes the intuition that DRSs are pictures, and *dynamic semantics*, which emphasizes the view that DRSs are context-transforming programs. We then examine the relationship between first-order languages and DRS languages. We conclude the chapter with two simple Prolog implementations: a model checker for DRSs, and a DRS to first-order logic compiler. By the end of the chapter it will be clear that DRT offers an entire architecture for thinking about semantics, an architecture that blends formal, empirical, and computational ideas in a flexible and suggestive way.

1.1 An Overview of DRT

In this section we introduce a number of key concepts of DRT. Much of our discussion revolves around the representation language DRT employs, a language based on box-like structures called DRSs. We will be making heavy use of DRSs in subsequent chapters, for many different purposes, so it is important that the reader has a good grasp of what they are and why they are so useful. The present section is largely based around the following intuition:

DRSs are Pictures.

That is, here we shall present DRT from a *representational* perspective, encouraging the reader to view DRSs as something like mental models constructed during the process of discourse comprehension. This isn't the only way of thinking about DRSs; there is also a natural *dynamic* perspective which insists that

DRSs are Programs.

This perspective will be introduced in the following section.

DRT and Context Change Potential

Imagine some agent—perhaps a little robot—about to receive an important message in natural language. Unless the message is very simple, a sequence of several natural language sentence will be required, and we immediately enter into a fascinating new area: *discourse*.

In this book we adopt a very simple model of discourse: discourses will simply be sequences of natural language sentences. (Of course, we certainly *don't* assume the converse: not every sequence of natural language sentences counts as a genuine discourse.) And the question that interests us is: how can we represent the meaning of a discourse? This is difficult to answer, but one negative observation can be made immediately: whatever the meaning of a discourse is, it *isn't* simply the conjunction of the first-order representations of its individual sentences. To see this, let's assume that our little robot is programmed to build first-order representations using the kind of tools developed in Part I. Now consider the following discourse:

'Mia is a woman. She loves Vincent'.

Our robot correctly parses the first sentence and obtains the representation WOMAN(MIA). Then along comes the second sentence, and with it one of the most obvious problems of discourse representation: how to cope with pronouns. Now, as we mentioned in Chapter **??**, in certain respects free variables are like pronouns, thus a first attempt at representation might be $\text{LOVE}(x, \text{VINCENT})$, and we certainly could build this using the kinds of techniques developed in Part I. However the meaning of the (first two items of) this discourse is *not* correctly captured by the following conjunction:

$$\text{WOMAN}(\text{MIA}) \wedge \text{LOVE}(x, \text{VINCENT}).$$

This representation misses the obvious fact that the pronoun 'she' 'refers back' to Mia. To use the linguistic terminology, 'she' is an *anaphoric* pronoun that needs to be *resolved*; we need to link it to a suitable discourse *antecedent* which will tell us what this pronoun is meant to refer to.

What are we to do? Well, perhaps we could could simply bite the bullet and say that this free-variable-containing conjunction really *is* the underlying representation, and that pronoun resolution is essentially a post-processing (or transformational) step that converts this crude, unresolved, formula into the final representation. That is, perhaps we should simply form the above conjunction and then find a strategy for converting it into

$$\text{WOMAN}(\text{MIA}) \wedge \text{LOVE}(\text{MIA}, \text{VINCENT}).$$

This strategy is just about plausible for simple discourses like the previous one, but it soon collapses. Consider instead the following discourse:

'A woman snorts. She collapses'.

Our naive first-order approach would build the following representation:

$$\exists z(\text{WOMAN}(z) \wedge \text{SNORT}(z)) \wedge \text{COLLAPSE}(x).$$

Again, this representation contains a free variable x, so our post-processing will need to do something about this. The obvious strategy is to try converting it into

$$\exists z(\text{WOMAN}(z) \wedge \text{SNORT}(z) \wedge \text{COLLAPSE}(z)).$$

This at least gets the truth conditions right—but there are two problems. For a start, our post-processing step is beginning to look rather complex: sometimes it has to to substitute a constant (such as MIA) for a free variable, while sometimes it has to play around with quantifier scopes. It isn't particularly plausible that transforming first-order representations in this way is compatible with the demands of good grammar engineering. As we saw in Chapter **??** (where we developed the lambda calculus after realizing that naive computational approaches to noun phrase representations led to trouble) and Chapter **??**

(where we examined a number of increasingly sophisticated approaches to quantifier scope) the key to good grammar engineering is to start with the right representation. The two (extremely simply) discourses we have examined make it clear that merging the information contained in successive utterances is a complex process, one that seems to involves both quantifier scope and free variable manipulation—and that's simply to allow us to cope with anaphoric pronouns! Once (say) presuppositions have been added to the brew, it becomes obvious that we need to adopt a more disciplined approach to representing and merging the information delivered in the course of a discourse. First order languages simply weren't designed with these kinds of issues in mind.

But there's a second problem: neither the alleged underlying representation

$$\exists z(\textsc{woman}(z) \land \textsc{snort}(z)) \land \textsc{collapse}(x)$$

nor the final resolved representation

$$\exists z(\textsc{woman}(z) \land \textsc{snort}(z) \land \textsc{collapse}(z))$$

corresponds well with our intuitions about the way the second discourse actually *works*. Intuitively, the sentence 'A woman snorts' sets the agenda. It paints a simple picture: we are talking about a woman, a woman who snorts, and presumably the subsequent discourse will tell us more about her (perhaps it will give her a name). The sentence 'She collapses' then goes on to exploit this: the 'she' in a very real sense *refers* to the woman we're talking about.

These are strong intuitions, and first-order representations simply don't get to grips with them. The first-order representation of 'A woman snorts' is $\exists z(\textsc{woman}(z) \land \textsc{snort}(z))$. This is truth-conditionally adequate—that is, it captures the fact that the sentence asserts that a woman exists and that she snorts—but it gives us no grip on the fact that a woman has—somehow or other—been made contextually relevant. That is, although the first-order representation handles the truth-conditional dimension of meaning well, something is missing.

This missing aspect of meaning is often called *Context Change Potential (CCP)*. When we utter a 'A woman snorts' we don't simply make a claim about the world, we also *change the context in which subsequent utterances will be interpreted*. If we are to make headway in computing discourse representations, we need to take account of the context change potential of sentences right from the start. That is, instead of starting with naive representations and hoping to post-process the contextual effects back in, we need to build more sophisticated representations which mirror context change potential in a natural way. Of course, we want these more sophisticated representations to get the truth conditions right as well—but, intuitively, it should not be too difficult to ensure this. It certainly seems far more sensible to try making truth conditional omelettes out of sophisticated eggs rather than first making the omelette and then trying get the eggs back!

Summing up, we would like a semantic architecture that enables us to build representations that naturally mirror CCP, but which also allows us easy access to the truth-conditional dimension of meaning. And in fact, this is precisely what Discourse Representation Theory offers. For our first encounter with the world of DRT, let's consider how a DRT-based robot would handle the previous discourse.

As soon as it realized that a new discourse had started, our robot would open a DRS, or box. To keep things simple, let's assume the robot is programmed to open the *empty box*, that is, the box containing no prior information:

In contrast, to the first-order robot, designed to operate on a "sentence-by-sentence first-order representations + post-processing" approach, the DRT robot, right from the start, is programmed to take an inter-sentential perspective on the discourse: it will progressively expand this empty box with information from the entire discourse. From the DRT perspective, discourse processing is all about carefully filling this initial box, observing various constraints while doing so. Moreover, the box filling process is highly intuitive: it builds a sort of picture (or perhaps, a mental model) of the incoming information. This picture records how the initial context (represented by the empty box) is changed in the course of the discourse.

When the 'A woman snorts' arrives, the robot parses it, and uses the parse tree to guide the way it expands the empty DRS. (We'll explain *how* the parse tree guides DRS construction when we discuss the standard construction algorithm; for the time being we'll ignore the underlying details.) The result of this process is the following DRS:

x
WOMAN(x) SNORT(x)

The x in the top compartment of this box is a *discourse referent*. The expressions WOMAN(x) and SNORT(x) in the bottom compartment are *conditions*. The discourse referent x is placed in the top compartment by the NP 'a woman'; in fact, the primary function of NPs in DRT is to introduce new discourse referents. The conditions WOMAN(x) and SNORT(x) were placed in the bottom compartment by the NP 'a woman' and the VP 'snorts' respectively. The reader should think of the top half of this DRS as the domain of a little model (the discourse model, if you like) and the discourse referents that sit there as entities we introduce to help us record the way the context changes as the information streams in; they are potentially salient objects, available for later anaphoric reference. The conditions in the bottom half of the picture record what we learn about these entities and their interrelationships as the discourse proceeds. The following metaphor is quite useful: think of the discourse referents as *pegs*, and the conditions as coats we hang on these pegs as best we can.

But to return to our example: what happens when our robot then receives the sentence 'She snorts'? The robot parses it and does three things. First, it adds a new discourse referent (y, say) to the top part of the DRS; it does this because 'she' is an NP, and the primary function of NPs is to introduce new discourse referents. Second, it adds the condition COLLAPSE(y); it is the VP that contributes this condition. Thirdly, and most interestingly, it adds a further condition, x=y. That is, after the second sentence has been processed, we have the following DRS:

(1) A woman snorts. She collapses.

x y
WOMAN(x) SNORT(x) COLLAPSE(y) y=x

It should be intuitively clear that the condition y=x is doing something very sensible: it identifies the collapsing entity with the snorting woman introduced in the first sentence, thus correctly resolving the pronoun 'she'. But *why* did the robot do this? Well, 'she' is not only an NP, it is a pronoun. Let us assume our robot is programmed to treat all pronouns as anaphoric. In DRT, anaphoric pronouns are handled as followed: the discourse referent introduced by the pronoun must be identified with an *accessible* discourse referent. Accessibility is a key concept in DRT and we'll be working with it a lot in this book—but for now it's enough to say that there is only one other discourse referent in our example DRS (namely x), and as a matter of fact x *is* accessible (we'll see why later), thus in this example the robot is *forced* to add the condition x=y. In short, our robot resolves pronouns by exploiting the picture of the context it has built.

The idea of building pictures of the changing context by introducing discourse referents and stating constraints on them is fundamental to DRT. For example, had the first sentence been 'Mia snorts', the robot would have built the following DRS:

x
x=MIA SNORT(x)

That is, the DRT perspective is that this sentence introduces a new discourse referent (namely x) and simultaneously records that this new 'peg' has to be identified with the entity named Mia. If you compare this example with the previous one, you will see that

DRT handles proper names and existentially quantified NPs in essentially the same way: both introduce a discourse referent which is then constrained by adding conditions. Moreover, note that there is an even closer parallelism between the treatment of anaphoric pronouns and proper names: both introduce new discourse referents, and both constrain them *equationally* (that is, using the equality symbol).

Summing up, DRT offers a natural architecture for thinking about the way information is accumulated in the course of discourse processing: essentially it allows us to draw pictures of the changing context. DRSs distinguish two types of information: information about which discourse entities we have at our disposal (recorded in the top compartment of DRSs) and information about the properties these entities have and the way they are interrelated (recorded in the bottom). As we have already seen, such pictures give us a natural way of thinking about anaphoric dependencies—but the DRT perspective turns out to be useful for a lot more besides: because it focuses on modeling how the context is transformed as the discourse proceeds, DRT is a good framework for thinking about many devices in natural language (such as presupposition, ellipsis, tense, and temporal reference) which give natural language discourses their rich intersentential structure. Switching to DRSs as the basic building blocks of our semantic representations will enable us to take some important first steps in discourse processing.

DRS Languages

We have just given our first examples of DRSs, and encouraged the reader to think of them as pictures—but clearly the simple DRSs we have seen so far can't cope with such obvious features of natural language as universal quantification and negation. How are we to deal with these? In fact, there are entire *languages* of DRSs: DRSs can be combined using various connectives and thus nested one inside another; these nested pictures offer us all representational power we would expect. So before going any further, let's be precise about the facilities DRS languages offer us.

Discourse Representation Structure languages (or *DRS languages*, or *box languages*) share many of the ingredients of first-order languages. Like first-order languages, they are built over *vocabularies* (see Chapter **??**). Like first-order languages, they contain the symbols $\neg$, $\vee$, $\rightarrow$ (though they normally don't contain $\wedge$), and like first-order languages with equality, they contain a special symbol $=$. In addition, DRS languages contain the symbols x, y, z, and so on, though these are called *discourse referents*, not variables. Nonetheless, there are important differences. Most importantly, DRS languages *don't* contain the symbols $\forall$ or $\exists$. Instead, they approach quantification via the idea of *Discourse Representation Structures (DRSs)* or *boxes*.

DRSs are pairs consisting of a finite set of discourse referents and a finite set of conditions. Some of these conditions are primitive—these are the only kind of conditions we have seen so far—but it is also possible to build complex conditions; these are built out of boxes using the connectives $\neg$, $\vee$, and $\Rightarrow$. As boxes may contain complex conditions, and as complex conditions are themselves defined in terms of boxes, boxes and conditions must be defined by mutual induction; let us give this crucial definition right away.

Suppose we have chosen some vocabulary of interest. We assume that the vocabulary contains no function symbols (this is simply because we won't be needing function symbols in what follows; it's easy to add them if they're wanted), thus a *term* τ is either a constant (if the vocabulary contains constants, which for natural language applications it normally would) or a discourse referent. We then build DRSs and conditions over our chosen vocabulary as follows:

1. If $\mathrm{x}_1, \ldots, \mathrm{x}_n$ are discourse referents (here $n \geq 0$) and $\gamma_1, \ldots, \gamma_m$ (here $m \geq 0$) 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 $\tau_1,\ldots,\tau_n$ are terms, then $R(\tau_1,\ldots,\tau_n)$ is a condition.
3. If τ_1 and τ_2 are terms then $\tau_1 = \tau_2$ is a condition.
4. If B is a DRS, then $\neg$B is a condition.
5. If B_1 and B_2 are DRSs, then $\mathrm{B}_1 \vee \mathrm{B}_2$ is a condition.
6. If B_1 and B_2 are DRSs, then $\mathrm{B}_1 \Rightarrow \mathrm{B}_2$ is a condition.
7. Nothing else is a DRS or a condition.

Clause 1 specifies that a DRS is a structure consisting of two components. The top component (which may be empty) consists of a set of discourse referents; we call this component the *universe*. The bottom component (which may also be empty) consists of a set of conditions. The DRS

$$\begin{array}{|c|}\hline \quad \\ \hline \quad \\ \hline \end{array}$$

(that is, the DRS with the empty universe that contains no conditions) is called the *empty DRS*, or the *empty box*.

Conditions licensed by clauses 2 and 3 are called *primitive* conditions; they are obvious analogs of first-order atomic formulas, and we have already seen examples of them. Conditions licensed by clauses 4, 5 and 6 are called *complex* conditions. As the reader probably suspects, $\neg$ handles negative information and $\vee$ handles disjunctive information. The $\Rightarrow$ construct handles both conditional and universally quantified information. Note that any DRS will have an outermost box (we call this the *main DRS*) and may in addition have a number of *sub-DRSs*; these sub-DRSs (if any) are the DRSs used to build complex conditions.

The definition of DRS languages may well have raised more questions than it answers; there's no denying that—at least at first glance—its two-dimensional box-based format is somewhat unusual. However, as will become increasingly apparent, DRS languages are extremely natural. For a start, the intuition that DRSs are pictures is a robust one; it extends in a natural way to DRSs containing complex conditions, and gives us a good way of defining the semantics of DRSs. Furthermore, the way boxes are nested one inside another gives rise to a geometrical notion called *accessibility*, and this will play an important role in later work.

Boxes as Pictures

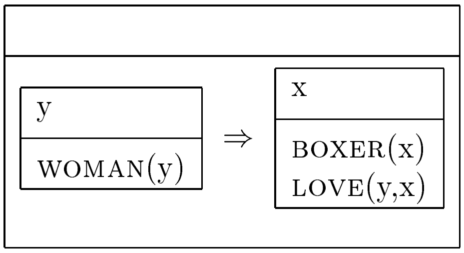
When is a DRS satisfied? Given that a DRS is supposed to be a picture, it seems natural to say that a DRS is satisfied in a model if and only if it is an accurate image of the information recorded inside the model. For example, the DRS

$$\begin{array}{|l|}\hline \mathrm{x}\ \mathrm{y} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{boxer}(\mathrm{y}) \\ \textsc{admire}(\mathrm{x},\mathrm{y}) \\ \hline \end{array}$$

should be satisfied with respect to a model if and only if it is possible to associate the discourse referents x and y with a pair of entities in the model such that the first entity is a woman, the second is a boxer, and the first stands in the admires relation to the second. If we can associate the discourse referents with such a pair of entities, it is as if the little picture the DRS gives us is "embedded" inside the (possibly very large) model, and it is natural to regard such a picture as satisfied.

In fact, the idea of associating discourse referents with model-theoretic entities, thereby embedding DRS in models, is the cornerstone of the embedding semantics we shall develop in detail in the following section. But the reader may be skeptical. For sure, thinking in terms of pictures is a natural way of viewing boxes that contain only basic conditions—but does it help us with the semantics of complex conditions?

Actually, it does. Roughly speaking, a negated DRS will be satisfied if it is *not* possible to find the picture the DRS gives us embedded inside the model, and a pair of DRSs joined by a disjunction symbol will be satisfied if at least one of these pictures can be found embedded inside the model. But what about conditions of the form $B_1 \Rightarrow B_2$? This is nice: such conditions will be satisfied if every way of embedding the antecedent picture B_1 gives rise to an embedding of the consequent picture B_2. For example, the DRS



which captures the narrow scope reading of the sentence 'Every woman loves a boxer', will be satisfied if no matter what entity we choose to associate with the discourse referent y (for example, let's suppose we choose an entity w), we will always be able associate x with an entity (let's call it b) such that w loves b. In short, no matter which entities we use to embed the antecedent picture, we will be able to embed the consequent picture too.

Thus DRS languages can be given a semantics that is faithful to the pictorial intuitions we have been emphasizing, and this is one good reason for being interested in them. But there is another: box syntax gives rise to the important notion of *accessibility*.

Accessibility

Our little DRT robot resolved the anaphoric pronoun 'she' by adding an equality condition to the picture it was building. Equality conditions are the mechanism used in DRT to resolve anaphors, and their use is subject to an interesting constraint: if y is the discourse referent introduced by a pronoun, and x is a previously introduced discourse referent, then we are only allowed to add the condition y=x if x is in the universe of a DRS that is *accessible* from the DRS whose universe contains y. This accessibility constraint not only grounds many of DRT's empirical claims about anaphora, it will also play an important role in our later discussion of presupposition.

What is accessibility? Actually, it's a simple geometric concept, defined in terms of the way DRSs are nested one inside another. Here's the definition. DRS B_1 is accessible from DRS B_2 when B_1 *equals* B_2 or when B_1 *subordinates* B_2. The subordinates relation is defined as follows: B_1 subordinates B_2 if and only if

1. B_1 *immediately subordinates* B_2; or
2. There is some DRS B such that B_1 *subordinates* B and B *subordinates* B_2.

That is, the subordinates relation is the transitive closure of the immediately subordinates relation. So to complete the definition we need only stipulate how immediate subordination is defined. We say: B_1 immediately subordinates B_2 if and only if

1. B_1 contains a condition of the form $\neg B_2$; or
2. B_1 contains a condition of the form $B_2 \vee B$ or $B \vee B_2$, for some DRS B; or
3. B_1 contains a condition of the form $B_2 \Rightarrow B$, for some DRS B; or
4. $B_1 \Rightarrow B_2$ is a condition in some DRS B.

Exercise 1.1.1 Draw the configurations of DRSs listed in the definition of immediate subordination. Suppose that B is a DRS and that B_s is a sub-DRS of B (that is, B_s is nested, perhaps very deeply, somewhere inside B). Prove that B accessible to B_s.

Now, the previous definition tells us what it means for one DRS to be accessible from another—but we really want accessibility to be a relation between *occurrences of discourse referents in universes*. However it is now straightforward to define this concept: we say that an occurrence of a discourse referent (say x) in the universe of some box (say B_1) is accessible from an occurrence of a discourse referent (say y) in the universe of some box (say B_2) if and only if B_1 is accessible from B_2. And given this concept we can now state the accessibility constraint on pronoun resolution more precisely:

> *Suppose a pronoun has introduced a new discourse referent (say y) into the universe of a DRS B_2. Then we are only free to add the condition $y=x$ to the condition set of B_2 if some occurrence of x in some universe is accessible from the occurrence of y in the universe of B_2.*

This can be simplified. When using DRSs to analyze natural language, we usually end up working with DRSs in which each discourse referent occurs in exactly one universe. When this is the case we don't need to bother talking about occurrences of discourse referents in universes—in effect we can identify a discourse referent with unique universe it occurs in, and simply talk of one discourse referent being accessible from another. This enables us to restate the accessibility constraint as follows:

> *Suppose a pronoun has introduced a new discourse referent (say y) into the universe of some DRS B. Then we are only free to add the condition $y=x$ to the condition set of B if x is accessible from y.*

We'll usually talk about accessibility in this simpler way in what follows.

Let's see some examples of the accessibility constraint at work. First, it enables DRT to correctly predict that an anaphoric link is permitted between 'she' and 'a woman' in the first little discourse that follows, but not in the second:

(2) A woman snorts. She collapses.

<table>
<tr><td>x y</td></tr>
<tr><td>WOMAN(x)
SNORT(x)
COLLAPSE(y)
y=x</td></tr>
</table>

Here we are free to add the constraint y=x, for the (unique) occurrences of x and y belong to the same universe, and thus x is accessible to y. On the other hand, consider the following two sentence sequence and its (only partially completed) DRS representation:

(3) Every woman snorts. ? She collapses.

<table>
<tr><td colspan="3">y</td></tr>
<tr><td><table><tr><td>x</td></tr><tr><td>WOMAN(x)</td></tr></table></td><td>⇒</td><td><table><tr><td></td></tr><tr><td>SNORT(x)</td></tr></table></td></tr>
<tr><td colspan="3">COLLAPSE(y)
y=?</td></tr>
</table>

In this example there is no accessible discourse referent for y—the only candidate is x, and x is not accessible from y. Hence we cannot complete the representation, and thus (correctly) predict that the pronoun 'she' does not have an anaphoric interpretation. Here are some other examples for which DRT correctly predicts felicity or infelicity on the basis of accessibility arguments:

(4) Mia ordered a five dollar shake. Vincent tasted it.

(5) Mia didn't order a five dollar shake. ∗ Vincent tasted it.

(6) Butch stole a chopper. It belonged to Zed.

(7) Butch stole a chopper or a motor cycle. ? It belonged to Zed.

(8) Butch stole a chopper or a motor cycle. ∗ The chopper belonged to Zed.

Exercise 1.1.2 Use accessibility arguments to explain why anaphoric reference is or is not possible in these examples.

To conclude this discussion, let's consider how DRT represents the infamous "donkey sentences"; for example, 'If a man eats a big Kahuna burger, he enjoys it'. (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'.)

Why are such sentences considered difficult? Because it is not clear how the required semantic representations can be built compositionally using the traditional tools of Montague grammar (roughly speaking, the λ-based approach to semantic construction introduced in Chapter **??**). Now, it's no problem finding a first-order formula which captures the truth conditions of this sentence; the following formula clearly gets it right:

$$\forall x \forall y[\textsc{man}(x) \wedge \textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \textsc{enjoy}(x,y)].$$

But how can we build this representation? 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 you attempt to build first-order representations using the methods of Chapter **??**, it's quite likely that you'll wind up generating:

$$\exists x[\textsc{man}(x) \wedge \exists y[\textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y)] \rightarrow \textsc{enjoy}(x,y)].$$

Exercise 1.1.3 Under the standard semantics of first-order logic, this 'representation' does not correctly capture the meaning of the sentence. Explain why.

In DRT this 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 looks more promising. 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), and the linkage between the pronouns and their antecedents is neatly captured by the two equalities. Moreover, this representation really does capture the meaning of the donkey sentence: it should be clear that this DRS will be satisfied if and only if whenever we find a little picture in which a man is eating a big Kahuna burger, we can extend this to a picture in which that same man is enjoying that same Kahuna burger. Furthermore, we will soon have a DRS to first-order logic compiler at out disposal, and this compiler will translate the above DRS to:

$$\forall x \forall y[\textsc{man}(x) \land \textsc{big\_k\_burger}(y) \land \textsc{eat}(x,y) \rightarrow \exists v \exists w(\textsc{enjoy}(v,w) \land v{=}x \land w{=}y)],$$

which is easily seen to be equivalent to the earlier first-order representation.

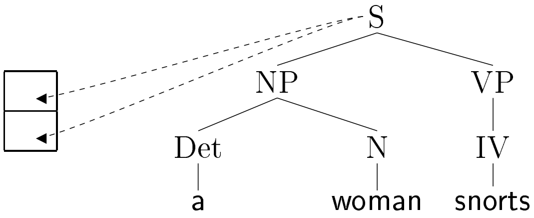
So the only question that remains to be answered is: can this DRS be built systematically? The answer is “yes”—indeed, from the DRT perspective there really isn’t anything particularly special about donkey sentences at all; they are handled using exactly the same construction methods that handle other sentence types. So the really important question is not “How do we handle donkey sentences in DRT?” but “How do we set about building DRSs at all?”. The following chapter is entirely devoted to this question, but it’s high time we had a first glimpse of the issues involved.

The Standard Construction Algorithm

In this section we’ll show, step-by-step, how the DRS corresponding to the discourse ‘A woman snorts. She collapses’ can be constructed using a simple top-down strategy. This strategy, which we call the *standard construction algorithm*, is the account of DRS construction the reader is most likely to meet in other presentations of DRT.

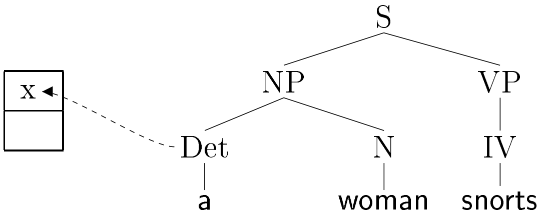
First we start feeding information from the initial sentence into a box:

Step 1:



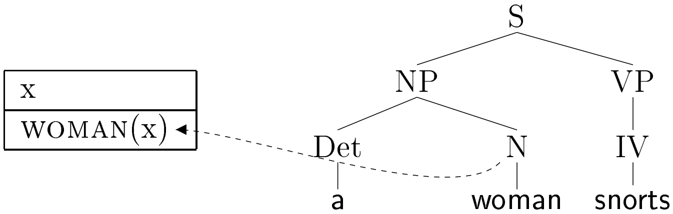
We proceed is follows: we are going to work our way around the tree in a top-down, left-to-right fashion, adding discourse referents as we go. So first we work down from the S node to the NP node. We then peek inside the NP: what sort of NP is it? In fact, it’s an indefinite noun phrase; indefinites introduce a new discourse referent into the upper part of the box, so we introduce the discourse referent x:

Step 2:



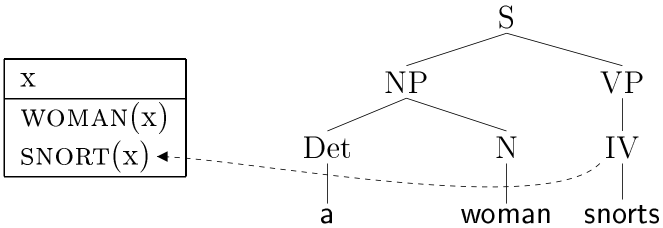
What conditions attach to this discourse referent? To find out, we move around the NP subtree to the N node. We find the word ‘woman’, thus we constrain x by putting the condition WOMAN(x) in the lower part of the box (Step 3).

Step 3:



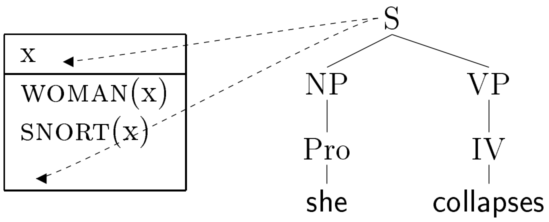
We’ve now explored the entire NP subtree; there’s nothing left to do here. So it’s time to move back up to S and then down into the VP subtree. Here we encounter the verb phrase that consists of the intransitive verb ‘snorts’. The standard algorithm has kept track of the fact that we’ve basically moved from the subject position of the tree to the predicate position, so it instructs us to view as a further constraint on x. Accordingly we add the condition SNORT(x) to the DRS:

Step 4:



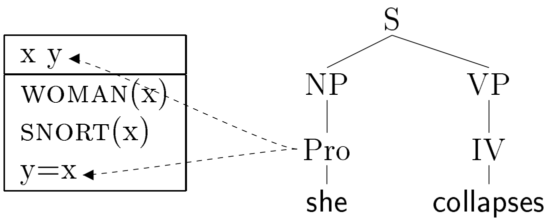
This is the DRS for the first sentence of discourse (2). Let’s continue straight on with the second sentence. As with the first sentence, we will explore its parse tree in a top-down, left-to-right fashion. Crucially, the standard construction algorithm insists that we add all the information we obtain from the second sentence straight into the DRS we have already built using the first sentence (after all, the main DRS is meant to be a picture of the entire discourse). This will make it very easy to perform the required anaphora resolution.

Step 5:



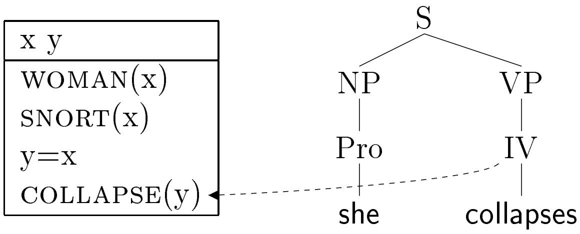
Again we work our way down from S, to NP, and then peek to see what kind of NP we are dealing with. It's a pronoun, so this means we must introduces a new discourse referent (we have chosen y) and a condition of the form y=?, where the question mark will be identified with an accessible, earlier introduced, discourse referent. (Actually, the standard algorithm insists on a further constraint: the discourse referent we use must not only be accessible, it must also be *suitable*—but we'll defer our discussion of suitability till Chapter 9.) There is only one accessible discourse referent available, namely the discourse referent x, which was introduced by 'a woman', so we are forced to build the following DRS:

Step 6:



This finishes the NP subtree. We then work our way round to the VP subtree and handle it in essentially the same way as did in the first sentence, so this is the final DRS we obtain:

Step 7:



This DRS captures the truth conditions of the discourse appropriately. Two entities are introduced, x and y, where x possesses the properties of being a woman and snorting, and y the property of collapsing, and these two entities are identified, just as the the anaphoric link between the pronoun 'she' and 'a woman' would lead us to expect.

The standard construction algorithm is simple and easy to understand. But a worrying thought may have occurred to some readers: is this is the *only* way we have of constructing DRSs? If so, then DRT isn't very computationally promising. The heart of computational semantics is developing flexible semantic construction strategies; we simply can't afford to get locked into a top-down (or indeed, any other) strait-jacket.

In fact, there's no problem: DRT turns out to be compatible with a wide range of implementation strategies. Nor is the standard construction algorithm a dead end: on the contrary, appropriately viewed it provides the basis for an efficient approach to DRS construction known as threading. These issues will be explored in detail in the following chapter.

1.2 Interpreting DRSs

It's time to be precise about the interpretation of DRSs. There are two popular ways of doing this: one is to use embedding semantics, which we have already briefly discussed, the other is to use dynamic semantics. These interpretations are equivalent—at least for the 'core' language of DRT we are discussing here—but both are worth getting to know, for each emphasizes different intuitions about what DRSs are.

Embedding Semantics

Suppose we are working with some DRS language, and that $\mathbf{M} = (D, F)$ is a model for that language (that is, the language and the model have the same vocabulary). An *embedding* is a partial function from the set of discourse referents to D, the domain of $\mathbf{M}$. Note that as they are *partial* functions, embeddings need not associate *all* discourse referents with elements of the model. Indeed there is a special embedding, the *empty embedding*, which does not associate *any* discourse referents with elements of the model.

Suppose i and j are embeddings in some model $\mathbf{M}$. We say that j *extends* i if and only if whenever $i(x)$ is defined for some discourse referent x, then $j(x)$ is defined too, and $j(x) = i(x)$. In set-theoretic terms this simply means that viewed as sets of ordered pairs, $i \subseteq j$. It is also useful to think of this concept in information-oriented terms: intuitively, j extends i means that j contains at least as much information as i. Note that every embedding extends itself, and that every embedding extends the empty embedding (think of the empty embedding as carrying no information).

We can now say what it means for an i embedding to satisfy a DRS B in a model $\mathbf{M}$:

$$\mathbf{M}, i \models \begin{array}{|l|}\hline \mathrm{x}_1,\ldots,\mathrm{x}_n \\ \hline \mathrm{K}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathrm{K}_m \\ \hline \end{array} \quad \textit{iff} \quad \begin{array}{l} \text{there is an embedding } j \text{ such that } j \text{ extends } i, \text{ and} \\ j(\mathrm{x}_1),\ldots,j(\mathrm{x}_n) \text{ are all defined, and} \\ j \text{ satisfies conditions } \mathrm{K}_1,\ldots,\mathrm{K}_m \text{ in } \mathbf{M}. \end{array}$$

This definition captures the intuition of "finding the picture inside the model" in a fairly obvious way: it demands that the embedding i matches discourse referents with elements of the model in such a way that all the conditions in the DRS are satisfied. (Note, incidentally, that this clause explains why we don't need an explicit conjunction symbol $\wedge$ in DRT; any conditions listed in the same DRS are, in effect, conjoined.) There is only one complication: why do we shift from the original embedding i to an extended embedding j? Because some of the conditions listed in the DRS we are interpreting (let's call it B) may be complex, and the DRSs these complex conditions are formed from may contain discourse referents that do not belong to the universe of the original DRS; we need to be able to successfully extend embedding i to cover all the discourse referents introduced in sub-DRSs.

Now for the next step: we need to say what it means for a model $\mathbf{M}$ and an embedding i to satisfy a condition K. For basic conditions, this is straightforward. First, for any term τ, by the interpretation of τ with respect to $\mathbf{M}$ and i we mean $F(\tau)$ if τ is a constant, and $i(\tau)$ if τ is a discourse referent; we denote the interpretation of τ by $I^i_F(\tau)$. Note that as embeddings are partial, $i(\tau)$ may be undefined; if this is the case we say that $I^i_F(\tau)$ is undefined too. We can now state what it means for basic condition to be satisfied:

$$\begin{array}{lll} \mathbf{M}, i \models R(\tau_1, \cdots, \tau_n) & \textit{iff} & I^i_F(\tau_1), \ldots, I^i_F(\tau_n) \text{ are all defined, and} \\ & & (I^i_F(\tau_1), \cdots, I^i_F(\tau_n)) \in F(R) \\ & & \\ \mathbf{M}, i \models \tau_1 = \tau_2 & \textit{iff} & I^i_F(\tau_1) \text{ and } I^i_F(\tau_2) \text{ are defined, and} \\ & & I^i_F(\tau_1) = I^i_F(\tau_2) \end{array}$$

Note that neither of these clauses makes use of the idea of extending embeddings: to interpret basic conditions we simply compare condition and model in the obvious way.

Now for the complex conditions. Let us say that an embedding j extends an embedding i *on* a box B if the arguments j is defined on are precisely the arguments i is defined on together with the discourse referents in the universe of B. Using this terminology we define:

$$\begin{array}{lll} \mathbf{M}, i \models \neg B & \textit{iff} & \text{for all } j \text{ such that } j \text{ extends } i \text{ on } B, \\ & & \text{it is not the case that } \mathbf{M}, j \models B \\ & & \\ \mathbf{M}, i \models B_1 \vee B_2 & \textit{iff} & \text{there is an } j \text{ such that } j \text{ extends } i \text{ on } B_1, \text{ and} \\ & & \mathbf{M}, j \models B_1; \text{ or} \\ & & \text{there is an } k \text{ such that } k \text{ extends } i \text{ on } B_2, \text{ and} \\ & & \mathbf{M}, k \models B_2 \\ & & \\ \mathbf{M}, i \models B_1 \Rightarrow B_2 & \textit{iff} & \text{for all } j \text{ such that } j \text{ extends } i \text{ on } B_1 \text{ and} \\ & & \mathbf{M}, j \models B_1, \\ & & \text{there is a } k \text{ such that } k \text{ extends } j \text{ on } B_2 \text{ and} \\ & & \mathbf{M}, k \models B_2 \end{array}$$

That is, a negative condition is satisfied precisely when we have painted ourselves into a corner: no matter how we extend the embedding i on the box in question—note that this may include doing nothing at all to i, since every embedding extends itself—we simply can't embed the DRS behind the negation symbol. Similarly, a disjunctive condition is satisfied precisely if it is possible to extend i so that at least one of the disjoined DRSs is embedded. Finally, a conditional condition is satisfied if every successful embedding of the antecedent leads to a successful embedding of the consequent, just as we discussed earlier.

We are almost there. We now know what it means for a DRS to be satisfied in a model with respect to a given assignment function—but what does it mean simply to say that a DRS is satisfied in a model? The following:

> *A DRS B is* embedding satisfied *in a model* $\mathbf{M}$ *if and only if there is an embedding i such that* $\mathbf{M}, i \models B$.

And that's the embedding semantics. As we promised, it makes precise the intuition that DRSs are pictures in a very direct way. Nonetheless some readers may have detected a second idea emerging in the above discussion. Many of the key definitions—including the one for boxes—made use of the idea of *extending* an embedding. There is something very procedural about this idea: indeed extending an embedding i to an embedding j sounds very like "adding information by performing a computation". In fact, the dynamic semantics we shall now define has its roots in this observation.

Exercise 1.2.1 Both the $\Rightarrow$ symbol and the $\vee$ symbol are *eliminable* from languages of DRSs. That is, we can throw away *both* the $\Rightarrow$ and $\vee$ from the language of DRSs without losing expressive power for we can express both types of condition using boxes and negative conditions. Prove this with the help of the embedding semantics. (We return to the eliminability of $\Rightarrow$ and $\vee$, from a somewhat different perspective, when we discuss the relationship of DRS languages and first-order languages; see in particular Exercise 1.3.5.)

Dynamic Semantics

The embedding semantics explains what it means for a picture to be verified. But although this tells us something about the notion of context change potential—after all, these pictures are pictures of changing context—it is based on an essentially static intuition. Dynamic semantics foregrounds the idea of change: it insists that DRSs are programs that modify context, and that only conditions should be viewed statically.

What kind of mathematical object could we use to model contexts? The traditional answer given in dynamic semantics is: *total embeddings*. Suppose we have fixed some (function symbol free) signature, and suppose that $\mathbf{M} = (D, F)$ is a model of this signature. A total embedding in $\mathbf{M}$ is a (total) function from the set of all discourse referents to D.

A moment's thought shows this is a reasonable notion of context. Given that our task is to explain the kind of context change potential that brings objects into salience and enables anaphoric links to be established, contexts are essentially objects that link discourse referents with model theoretic entities—and this is precisely what total embeddings are. In fact we've met this idea before. In Chapter 1 we noted that an assignment of values to variables could be viewed as a context; but of course, 'discourse referent' is simply DRT terminology for 'variable', hence a total embedding is nothing but an assignment function.

There is one point which may be bothering some readers: do we really need to work with *total* embeddings? The answer is, "No, we don't"; but doing so simplifies the technicalities, and it is pleasant to have an interpretation for DRS languages that makes use of *exactly* the same semantic machinery—models and assignment functions—as first order languages.

So how does the the dynamic put this machinery to work? First, it draws a distinction between the semantics of conditions and the semantics of DRSs. Conditions are regarded as *tests* and treated statically: $\mathbf{M}, g \models K$ will mean that K (a test) is satisfied in a model $\mathbf{M}$ (a situation) under a total embedding g (a context). Conceptually and notationally this is like the treatment of satisfaction in first-order logic; tests don't change contexts—their semantics is static. The dynamics comes in the treatment of DRSs: we are going to think of DRSs as non-deterministic programs which act on contexts to produce new contexts. That is, we will define what it means for a DRS B (a *program*) to be satisfied in a model $\mathbf{M}$ (a situation) with respect to a *pair* of total embeddings g and h, (the *input context* and the *output context*); this is written $\mathbf{M}, g, h \models B$. Thus dynamic semantics will blend four ideas: programs, tests, situations, and contexts. Incidentally, *blends* is the key word here. Much of the interest of dynamic semantics lies in the harmonious way it combines a dynamic analysis of context change with traditional static ideas.

So let's turn to the key definition. The following piece of notation will be helpful. If g and h are total embeddings on the same model we shall write $g[\mathrm{x}_1, \ldots, \mathrm{x}_n]h$ to indicate that g differs from h, if at all, only in the values it assigns to the discourse referents $\mathrm{x}_1, \ldots, \mathrm{x}_n$. (Note that this is just an n-place version of the idea of variant assignments used in the definition of first-order satisfaction in Chapter **??**.) As before, we'll let $I_F^g(\tau_n)$, the denotation of τ be $F(\tau)$ if τ is a constant, and $g(\tau)$ if τ is a discourse referent. We can now define the static and dynamic notions we need by mutual indication:

$$\mathbf{M}, g \models R(\tau_1, \cdots, \tau_n) \quad \textit{iff} \quad (I_F^g(\tau_1), \cdots, I_F^g(\tau_n)) \in F(R)$$

$$\mathbf{M}, g \models \tau_1 = \tau_n \quad \textit{iff} \quad I_F^g(\tau_1) = I_F^g(\tau_n)$$

$$\mathbf{M}, g \models \neg B \quad \textit{iff} \quad \text{for all } h, \text{ it is not the case that } \mathbf{M}, g, h \models B$$

$$\mathbf{M}, g \models B_1 \vee B_2 \quad \textit{iff} \quad \begin{array}{l}\text{there is an } h \text{ such that} \\ \mathbf{M}, g, h \models B_1 \text{ or } \mathbf{M}, g, h \models B_2\end{array}$$

$$\mathbf{M}, g \models B_1 \Rightarrow B_2 \quad \textit{iff} \quad \begin{array}{l}\text{for all } f \text{ such that } \mathbf{M}, g, f \models B_1 \\ \text{there is an } h \text{ such that } \mathbf{M}, f, h \models B_2\end{array}$$

$$\mathbf{M}, g, h \models \begin{array}{|l|}\hline \mathrm{x}_1, \ldots, \mathrm{x}_n \\ \hline \mathrm{K}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathrm{K}_m \\ \hline \end{array} \quad \textit{iff} \quad \begin{array}{l} g[\mathrm{x}_1, \ldots, \mathrm{x}_n]h \text{ and} \\ \mathbf{M}, h \models \mathrm{K}_1 \;\&\cdots\&\; \mathbf{M}, h \models \mathrm{K}_m \end{array}$$

The first two clauses simply say that basic conditions are satisfied in a situation $\mathbf{M}$ given contextual information g precisely when the interpretations of the discourse reference guarantees that we get the answer "Yes" to these tests.

The third clause says that a condition $\neg B$ is satisfied in situation $\mathbf{M}$ in context g precisely when the *program* B cannot be successfully executed when g is used as the input context. (Note the flip-flop, characteristic of DRS languages, from conditions to programs.) That is, when we run program B on input g, there is no state h that is output by this computation. In the fourth clause we again see the same flip-flop: a disjunctive condition is satisfied precisely when at least one of its component programs can be executed on the input context g, producing a context h as output. (That is, at least one of the component programs is capable of transforming the input context in a satisfactory way.)

The clause for implicational conditions says that an implication is satisfied in situation $\mathbf{M}$ in context g precisely when every possible output f of this computation can be successfully acted on by the consequent program B_2 to produce output context. As a well known slogan puts it: *every successful execution of the antecedent leads to a successful execution of the consequent.* To appreciate the force of this definition the reader must recall that as we are thinking of DRSs as *non-deterministic* programs, a DRS may be able to modify the input context in various ways. This definition demands that no matter which context the DRS gives back, it must always be possible to execute the consequent program on it.

The last clause is the key one: it tells us what these programs actually do, and why they are non-deterministic. Given an input context g, a DRS must try to find interpretations for the discourse referents in its universe so that all its conditions are satisfied. That is, boxes are programs that update the links between discourse referents and the world. Note that such programs are intrinsically non-deterministic: after all, there may be many ways of updating these links that result in all conditions being satisfied.

One task remains: defining what it means for a DRS to be satisfied in a model.

> A DRS B is *dynamically satisfied* in a model $\mathbf{M}$ with respect to assignment f if there is an assignment f' such that $\mathbf{M}, f, f' \models$B.

And that's dynamic semantics. It should be quite clear that embedding semantics and dynamic semantics are closely linked—indeed they are essentially two different perspectives on the same set of ideas. The best way to get to grips with the way these two interpretations fit together technically is to attempt the following exercise right away.

Exercise 1.2.2 Show that any DRS B can be dynamically satisfied in a model $\mathbf{M}$ if and only if it can be embedding satisfied in model $\mathbf{M}$.

1.3 DRT and First-Order Logic

In spite of their differences, DRT and first-order logic are obviously related. Given any vocabulary without function symbols, we can use it to build either a DRS language or a first-order language. Moreover, both languages will interpreted on exactly the same models. Thus it is natural to try to relate DRS and first-order languages via satisfaction preserving translations. Such translations essentially tell us how to compile one language down into the other in a sensible way.

As we shall now show, any DRS language and its corresponding first-order language are intertranslatable: either can be compiled down into the other, and the translations involved are straightforward and efficient.

From DRT to First-Order Logic

We first show how to translate DRSs into formulas of first-order logic with equality. Suppose we are working with a vocabulary that contains no function symbols. We shall define a translation function *fo* which takes any DRS built over this vocabulary and maps it to a formula of the first-order language (with equality) built over the same vocabulary. This function works by recursively mapping the DRSs and conditions that make up the input DRS to first-order formulas. It is easy to understand and (as we shall see in the following section) easy to implement in Prolog.

First we need to know how to translate boxes. Here is the general translation schema:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}\right)^{fo} = \exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo})$$

That is, this clause maps the discourse referents in the universe of the box to existentially quantified variables, and then recursively translates the conditions. The basic idea should be clear, but to avoid misunderstanding it is worth spelling out what this schema means in the following special cases. First, if there is only one condition γ_1 in the condition set, then the translation is $\exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n(\gamma_1)^{fo}$. Second, if there are no conditions in the condition set, then the translation is $\exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n \top$ (recall that $\top$ stands for *true*). Third, if the DRS we are translating has an empty universe, then the translation is $(\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo}$. It follows from these conventions that the translation of the empty DRS is $\top$.

Now for the conditions. The treatment of the basic conditions couldn't be simpler; they simply map to themselves-viewed-as-first-order-atomic-formulas:

$$\begin{array}{rcl} (R(\mathrm{x}_1, \ldots, \mathrm{x}_n))^{fo} & = & R(\mathrm{x}_1, \ldots, \mathrm{x}_n) \\ (\tau_1 = \tau_2)^{fo} & = & \tau_1 = \tau_2 \end{array}$$

Moreover, complex conditions formed using $\neg$ and $\vee$ are also straightforwardly handled; we simply push the translation function in over the connective, leaving the connective unchanged:

$$\begin{array}{rcl} (\neg B)^{fo} & = & \neg(B)^{fo} \\ (B_1 \vee B_2)^{fo} & = & (B_1)^{fo} \vee (B_2)^{fo} \end{array}$$

Now for complex conditions formed using $\Rightarrow$. Here's the general schema:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array} \Rightarrow B\right)^{fo} = \forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n(((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo}) \rightarrow (B)^{fo})$$

This clause clearly captures the idea that every possible embedding of the antecedent DRS should lead to a embedding of the conclusion. However, as with the translation of boxes, it is a good idea to be explicit about what this schema means in the following special cases. First, if there is only one condition γ_1 in the condition set of the antecedent, then the translation is $\forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n((\gamma_1)^{fo} \rightarrow (B)^{fo})$. Second, if there are no conditions in the condition set of the antecedent, then the translation is $\forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n(\top \rightarrow (B)^{fo})$, which is logically equivalent to $\forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n(B)^{fo}$. Third, when the antecedent DRS has an empty universe, the translation is $(\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo} \rightarrow B^{fo}$. It follows from these conventions that if the antecedent DRS is the empty DRS, we obtain $\top \rightarrow (B)^{fo}$, which is logically equivalent to $(B)^{fo}$, as the translation.

This translation is not only semantically sensible, it is also *efficient*: the number of symbols in the first-order formula it returns as output is of the same order of magnitude as the number of symbols in the DRS. This means that we have an easy way of turning DRT inference problems into first-order inference problems, a possibility we shall exploit heavily in later chapters.

Exercise 1.3.1 Use *fo* to translate the DRS representing if a man eats a big Kahuna burger, he enjoys it into first-order logic with equality.

Exercise 1.3.2 We claimed that *fo* was a satisfaction preserving translation. Show that this claim can be made precise in two ways. Let B be a DRS. First, show that B is *dynamically* satisfied in a model $\mathbf{M}$ if and only if $\mathbf{M} \models (\mathrm{B})^{fo}$. Second, show that B is *embedding* satisfied in $\mathbf{M}$ if and only if $\mathbf{M} \models (\mathrm{B})^{fo}$.

From First-Order Logic to DRT

We now go in the reverse direction: we shall show how given any vocabulary at all (even one containing function symbols) we can map first-order formulas over this vocabulary to a DRS built over the same vocabulary (minus the function symbols) in a satisfaction preserving way. We shall present a *lean* version of the translation; that is, one that does not operate directly on all first-order connectives. The exercises below ask the reader to fatten it up, and also show why lean can be useful.

The translation requires a preprocessing stage, which performs three functions. First, it rewrites the formula to an equivalent formula that contains no function symbols. (We saw how to do this in Exercise **??**.) Having done this, it rewrites this function-free formula to an equivalent formula that contains only the connectives $\neg$, $\vee$ and $\exists$. (Again this is possible; see Exercise **??**.) Finally, the resulting formula is rewritten to an equivalent formula in which distinct occurrences of $\exists$ bind distinct variables, an easy (but important) step. We are only going to translate the formulas that are output by this three-step preprocessing stage.

Given a preprocessed first-order formula ϕ, how do we translate it? First we form the following structure:

$$\begin{array}{|l|}\hline \quad \\ \hline (\phi)^{dr} \\ \hline \end{array}$$

That is, we take an empty box, and put the expression $(\phi)^{dr}$ in the compartment reserved for conditions. The superscripted *dr* is our translation function, and we shall simply keep recursively applying this function; as we do so we will fill the box with conditions and discourse referents until we finish building the DRS we require.

So, what does *dr* let us do? Obviously we need to specify a translation step for each kind of first-order formula we could encounter, and as ϕ has been preprocessed this means atomic formulas, conjunctions, negations, and existential quantifications. Let's go through these possibilities in turn, starting with the atomic formulas.

Suppose we see a box of the following form somewhere in the DRS we are building (it may be the main box itself, or a sub-box):

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\tau_1 = \tau_2)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

(The notation is meant to emphasize that the important thing is that we see a box containing $(\tau_1 = \tau_2)^{dr}$: it *doesn't* matter if this box contains other conditions, either before or after the $(\tau_1 = \tau_2)^{dr}$, and it *doesn't* matter which—if any—discourse referents this box has in its universe.) If we find such a box, it should be clear how to handle it: we simply drop the brackets. Doing so yields a legitimate DRS condition and gives us the following box:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ \tau_1 = \tau_2 \\ \textit{Conditions} \\ \hline \end{array}$$

In a similar fashion, if we see a box containing the expression $(R(\mathrm{x}_1, \ldots, \mathrm{x}_n))^{dr}$, then we simplify this to $R(\mathrm{x}_1, \ldots, \mathrm{x}_n)$ and carry on.

So what happens if we see a box containing a conjunction that needs to be translated? That is, what do we do if we find something of the following form:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\phi \wedge \psi)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

Fairly obviously, we simply break this expression in two, forming the following box:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\phi)^{dr} \\ (\psi)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

So let's consider the clause for negation, which is rather more interesting. Suppose we find a box of the following form:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\neg\phi)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

This triggers sub-DRS construction—we form a negative condition by opening up a negated DRS:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ \neg \begin{array}{|c|}\hline \quad \\ \hline (\phi)^{dr} \\ \hline \end{array} \\ \textit{Conditions} \\ \hline \end{array}$$

Finally, suppose we encounter an existentially quantified expression that needs translating:

$$\begin{array}{|l|}\hline \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\exists x\phi)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

We simply add the variable x to the universe of the DRS and form:

$$\begin{array}{|l|}\hline \text{x}, \ \textit{Discourse Referents} \\ \hline \textit{Conditions} \\ (\phi)^{dr} \\ \textit{Conditions} \\ \hline \end{array}$$

Note that this step exploits our preprocessing: we know that each quantifier bind a *unique* variable, so, given the inductive nature of our construction, x is guaranteed to be distinct from any of the discourse referents already in this box's universe.

And that's the translation process—we simply start at the top and carry out this simplification process until we bottom out in the atomic formulas; it easy to show that this process terminates yielding a legitimate DRS. Note that this translation is *non-deterministic*: we are allowed to simplify in any order we like, wherever we see an appropriate configuration. It should also be clear that this translation is just begging to be implemented in Prolog, and after we have discussed how to represent DRS in Prolog we shall get the reader to do this (see Exercise1.4.4).

Between them, *fo* and *dr* unambiguously show that—*regarded simply as tools for talking about models*—DRS languages and first-order languages are expressively equivalent. This strongly underlines the point made at the start of the chapter: the move to DRT does *not* reflect dissatisfaction with first-order logic as a tool for representing truth conditions. First-order logic is excellent at this task (it's not called classical logic for nothing) and DRS languages add *nothing* on this level. Rather, the gains from DRS languages reflect a completely different set of ideas. Although they offer the same model-theoretic expressivity as first order languages, they package this expressivity differently. As both embedding and dynamic semantics in their different ways show, this repackaging allows us to model the idea of context change potential in a straightforward and natural way. DRS languages and first-order languages don't differ in what they say about models; they differ in what they teach us about context.

Exercise 1.3.3 Preprocess $\forall x(\textsc{woman}(x) \rightarrow \textsc{snort}(x))$ and use *dr* to translate it into a DRS.

Exercise 1.3.4 As the previous exercise shows, a lean translation has its unpleasant aspects: it requires us to eliminate symbols before we can use it. Fatten *dr* up by adding clauses that operate directly on $\vee$, $\rightarrow$, and $\forall$ formulas, and then retranslate $\forall x(\textsc{woman}(x) \rightarrow \textsc{snort}(x))$.

Exercise 1.3.5 But lean translations can be useful. For example, using the fact that *fo* and *dr* are both satisfaction preserving, it is easy to show that the symbols $\vee$ and $\Rightarrow$ can be eliminated from DRS languages without losing expressivity. Indeed we can even pull out explicit paraphrases of $\vee$ and $\Rightarrow$ conditions. Why is this? [Hint: suppose a DRS B contains occurrences of $\Rightarrow$ or $\vee$. What do you get if you first apply *fo* and then apply *dr* to the result?]

1.4 DRT in Prolog

The architecture that is offered by DRT is gradually coming into focus—but so far our discussion has been purely theoretical. What does DRT look like from the perspective of *computational* semantics? The remainder of this book is essentially an extended answer to this question; in the present section we lay the computational foundations for the chapters that follow.

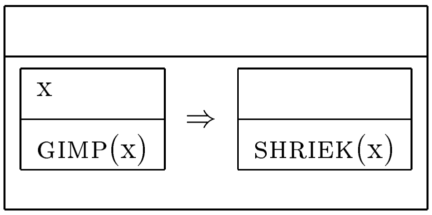
We shall do two things. First, we define a simple model checker for DRS languages. This will enable us to fix our basic Prolog conventions for DRT, and gives us an interesting new perspective on the semantics of DRSs. Following this, we implement the translation of DRT into first-order logic given in Section 1.3. This simple program opens the door to the the world of first-order inference techniques, and underpins much of our later work.

A Simple Model Checker

We want to define a model checker for DRSs. As we know from Chapter 1, where we defined a model checker for first-order logic, there are four things on our to-do list. We have to: (1) decide how to represent vocabularies in Prolog, (2) decide how to represent models, (3) decide how to represent DRSs, and (4) specify the model checker for DRSs. The good news is that we can reuse the vocabularies and model representations that we introduced in Chapter 1, so let's forget (1) and (2) and turn straight to (3): fixing a Prolog notation for boxes.

We will represent DRSs in Prolog as terms of the form `drs(D,C)`, where `D` is a list of terms representing the discourse referents, and `C` is a list of other terms representing the DRS conditions. To represent complex conditions we use the same operator definitions as for first-order logic (but note that we are not going to use the operator for conjunction, since we don't need it in DRT).

Let's consider an example. The DRS for 'Every gimp shrieks' is:



And in Prolog we represent this DRS as:

```
drs([],[drs([X],[gimp(X)]) > drs([],[shriek(X)])]).
```

Our choice of representation is actually rather arbitrary. For a start, we didn't have to use `drs` as the functor name—pretty much anything else would have done. Moreover, we could have devised a glossy operator notation for DRSs had we wanted. Basically, we chose this aspect of our representation because it is simple and fairly readable, but nothing much hangs on it. Note, however, that discourse referents are represented as Prolog variables. That is, we have made the same choice here that we made when deciding how to represent first-order variables. As we have already discussed, this choice has a number of advantages (in particular, we won't have to worry about α-conversion, for Prolog handles that automatically) but it can be dangerous if used carelessly.

Exercise 1.4.1 What are the Prolog representations of the DRSs for (1) 'Vincent does not die', (2) 'Vincent dances and Mia dances', (3) 'Vincent cleans the back seat or Jules cleans the back seat', and (4) 'If Butch wins, Marsellus loses'?

We now turn to task (4): defining a simple model checker for DRSs. We do so in the spirit of Chapter 1, where we first defined a (rather naive) `satisfy/2` predicate that succeeded if its first argument, a formula of first-order logic, could be satisfied in the model specified by the second argument.

Now, DRSs and conditions are mutually recursive constructs. Unsurprisingly, this is reflected in the implementation of our model checker: we will specify a predicate `satisfyDrs` that succeeds if a DRS can be satisfied in a model, and a predicate `satisfyCondition` that succeeds if a DRS-condition can be satisfied in a model. These predicates will be mutually recursive. Here is the code for `satisfyDrs`:

```
satisfyDrs(drs(Dom,C),Model):-
   assignReferents(Dom),
   satisfyConditions(C,Model).

assignReferents([]).
assignReferents([Referent|Others]):-
   constant(Referent),
   assignReferents(Others).
```

Using the predicate `assignReferents/1`, `satisfyDrs/2` assigns elements of the model (which are specified as constants in our vocabulary) to the discourse referents of the DRS. The next step involves checking the conditions of this DRS, and this is done with the help of `satisfyConditions/2`.

So how do we define `satisfyConditions/2`? Obviously we will need a recursive clause that works us through the condition list, and this is straightforward to define:

```
satisfyConditions([],_).
satisfyConditions([Condition|Others],Model):-
   satisfyCondition(Condition,Model),
   satisfyConditions(Others,Model).
```

So it remains to specify what `satisfyConditions/2` does to individual conditions. Let's first consider negative and disjunctive conditions. In fact, nothing much needs to be said here; we can deal with them simply by using Prolog negation (negation as failure) and Prolog disjunction respectively:

```
satisfyCondition(~ Drs,Model):- \+ satisfyDrs(Drs,Model).

satisfyCondition(Drs1 v Drs2,Model):-
   (
    satisfyDrs(Drs1,Model)
   ;
    satisfyDrs(Drs2,Model)
   ).
```

The clause for $\Rightarrow$ is more interesting; here's how we'll do it:

```
satisfyCondition(Drs1 > Drs2,Model):-
   (
    satisfyDrs(Drs1,Model),
    \+ satisfyDrs(Drs2,Model),
    !,
    fail
   ;
    true
   ).
```

What's going on here? Well, from a computational perspective, the following: this clause tries to satisfy the antecedent DRS (that is, `Drs1`), and, at the same time, tries to prove that the consequent DRS (that is, `Drs2`) *cannot* be evaluated. If this is the case, the implication as a whole cannot be satisfied, and we use the cut-fail combination to inform Prolog of this. Now, using backtracking, Prolog will try out every possible assignment of constants of the domain to the discourse referents of the antecedent DRS; if it does not succeed in showing that the consequent DRS cannot be satisfied, then we land in the second disjunct of the clause, where the built-in Prolog predicate `true` tells us that the implication as a whole *can* be satisfied.

But there's a more abstract perspective on this code. As we asked the reader to show in Exercises 1.2.1 and 1.3.5, $\Rightarrow$ conditions can be eliminated without loss of expressive power. In fact, a configuration of the form

$$\begin{array}{|l|}\hline \\ \hline \begin{array}{|l|}\hline \mathrm{y}_1,\ldots,\mathrm{y}_n \\ \hline \mathrm{K}_1 \\ \vdots \\ \mathrm{K}_n \\ \hline \end{array} \Rightarrow \mathbf{B} \\ \\ \hline \end{array}$$

is equivalent to a configuration of the form

$$\begin{array}{|l|}\hline \\ \hline \neg \begin{array}{|l|}\hline \mathrm{y}_1,\ldots,\mathrm{y}_n \\ \hline \mathrm{K}_1 \\ \vdots \\ \mathrm{K}_n \\ \neg\mathbf{B} \\ \hline \end{array} \\ \\ \hline \end{array}$$

A moment's thought reveals that the code above exploits this equivalence.

We come at last to the basic conditions. Note that identity conditions are simply checked by trying to unify them—this only succeeds if `X` and `Y` are identified with the same constants out of the model's domain.

```
satisfyCondition(X=Y,_Model):-
   X=Y.

satisfyCondition(BasicCond,Model):-
   memberOfList(BasicCond,Model).
```

It only remains to wrap this all up in a driver. We'll do this in the way we did in Chapter **??**; we'll define a driver that, given a DRS, picks an example model and tries to satisfy it:

```
evaluate(Drs,Example):-
   example(Example,Model),
   satisfyDrs(Drs,Model).
```

And now for the $64,000 question: *what have we actually implemented*? Clearly we've implemented some kind of DRS model-checker—but is it an implementation of embedding semantics or of dynamic semantics?

It's impossible to decide—it's both. Read declaratively, the code seems a fairly direct implementation of embedding semantics. Of course, we have to make allowances for the fact that we're working with lists and not sets—nonetheless, the basic correspondence is clear. On the other hand, if you think of what the Prolog interpreter is actually doing with this code, one is drawn to dynamic semantics; that is, read procedurally, this code seems more like an implementation of dynamic semantics. In short, simply expressing the basic semantic ideas of DRT in Prolog blurs the distinction between the dynamic and embedding semantics, which emphasizes how easy it is to view DRS languages either statically or dynamically.

In our view, this is just the way it should be. DRT supports a rich and productive interplay of intuitions. Simple though it is, our little model checker underlines the fundamental unity of some of its key supporting ideas.

Exercise 1.4.2 Explain why, in the clause for checking identity of two discourse referents, there is no need to use information from the model, as supplied by the second argument of `satisfyCondition/2`.

Exercise 1.4.3 As we know from Chapter 1, using Prolog unification in combination with negation as failure can have nasty consequences. Test how the DRS model checker fares with respect the problems we noted for the naive first-order logic model checker developed in Chapter 1. Define a refined version of the DRS-evaluation predicate that deals with these problems, reusing as much Prolog code as possible.

Compiling DRSs into First-Order Logic

In this section we define a simple Prolog predicate called `drs2fol` which takes a DRS and produces an equivalent first-order formula; this predicate is a straightforward implementation of the translation function *fo* discussed in Section 1.3.

Let's first consider how to translate boxes. Recall that *fo* does this as follows:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n\\ \hline \gamma_1\\ \cdot\\ \cdot\\ \cdot\\ \gamma_m\\ \hline\end{array}\right)^{fo} = \exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo})$$

Given our list-based Prolog representation of DRSs, the following is a natural implementation:

```
drs2fol(drs([],[Cond]),Formula):- cond2fol(Cond,Formula).

drs2fol(drs([],[Cond1,Cond2|Conds]),Formula1 & Formula2):-
    cond2fol(Cond1,Formula1),
    drs2fol(drs([],[Cond2|Conds]),Formula2).

drs2fol(drs([X|Referents],Conds),exists(X,Formula)):-
    drs2fol(drs(Referents,Conds),Formula).
```

That is, we work through the list of discourse referents, recursively pumping out the existential quantifiers we need; the third clause does this. In addition, we work our way through the condition list, recursively conjoining their translations as we do so; the second clause does this. Of course, as DRSs and conditions are mutually inductive concepts, the second clauses need to call on a predicate `cond2fol` which tells us how to translate conditions to first-order formulas; we will define this predicate shortly. The first clause grounds this process with a call to `cond2fol`.

So our next task is is to specify what `cond2fol` does. The clauses for disjunctive and negative conditions are transparent implementations of what *fo* does in these cases:

```
cond2fol(~ Drs, ~ Formula):-
    drs2fol(Drs,Formula).

cond2fol(Drs1 v Drs2, Formula1 v Formula2):-
    drs2fol(Drs1,Formula1),
    drs2fol(Drs2,Formula2).
```

But now we must deal with $\Rightarrow$ conditions. Recall that *fo* handles these as follows:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1, \ldots, \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array} \Rightarrow B\right)^{fo} \quad = \quad \forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n(((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo}) \rightarrow (B)^{fo})$$

We implement this by working recursively through the discourse referent list, pumping out the universally quantified formulas we need, and then placing the implication in place:

```
cond2fol(drs([],Conds) > Drs2, Formula1 > Formula2):-
    drs2fol(drs([],Conds),Formula1),
    drs2fol(Drs2,Formula2).
```

```
cond2fol(drs([X|Referents],Conds) > Drs2, forall(X,Formula)):-
    cond2fol(drs(Referents,Conds) > Drs2,Formula).
```

Only one task remains: we need to say how basic conditions are translated. This is the clause on which the entire mutual induction rests, and it is defined as follows:

```
cond2fol(BasicCondition,AtomicFormula):-
    BasicCondition =.. [_Predicate|[FirstArg|OtherArgs]],
    simpleTerms([FirstArg|OtherArgs]),
    AtomicFormula=BasicCondition.
```

That is, when we reach the bottom level, we check that what we have found is really a basic condition; if it is, we have the atomic first-order formula we need.

And that's it. Very simple, and as we shall see, very useful. We now have a bridge between the world of DRT and the world of first-order inference methods, and we shall make heavy use of it in Chapter 6.

Exercise 1.4.4 Implement the translation *dr* discussed in Section 1.3 that maps first-order expressions to DRSs. Don't just implement the lean version: add clauses to handle $\vee$, $\Rightarrow$ and $\forall$ directly (see Exercise 1.3.4).

Exercise 1.4.5 Is `drs2fol/2` reversible? It clearly isn't. Check the predicate on all input modes, and report cases where it fails to give the right result.

Software Summary of Chapter 1

`modelDRT.pl` Contains the predicate that evaluate a DRS with respect to a first-order model. (page 173)

`drs2fol.pl` The translation from DRSs to first-order formulas. (page 175)

Notes

DRT was originally developed by Kamp (1984) and Heim (1982) (who called it "File Change Semantics"). DRT is blessed with an excellent textbook, namely Kamp and Reyle

1993. 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 standard construction algorithm is developed in detail and DRS are interpreted via the embedding semantics; the reader who wants to find out more about DRT is advised to start here. Another textbook level account well worth looking at is given in the second volume of Gamut (Gamut 1991). This account is more critical of DRT, and develops an alternative called Dynamic Predicate Logic (DPL); this interprets the standard syntax of first-order using a variant of the dynamic semantics discussed in the text. However the box-based syntax of DRT is now widely accepted as standard notation in linguistics and computational linguistics, and dynamic semantics seems to be increasingly viewed as an alternative interpretation of DRSs. The translation *fo* of DRSs into first-order logic with equality given in the text was taken from Kamp and Reyle (Kamp and Reyle 1993). It's probably the best known translation, but there are others; Muskens (Muskens 1996), for example, gives an interesting alternative. For a more advanced discussion of DRT, try Van Eijck and Kamp 1997.

Chapter 2

Building Discourse Representations

We now know what DRT is, and have developed some simple Prolog programs for working with DRS languages. But how are we to construct the DRSs that correspond to sentences, or indeed, entire discourses? This chapter explores the issue in detail, and the basic message that emerges is straightforward:

> *DRT is a highly flexible theory, compatible with a wide range of semantic construction strategies.*

We begin by exploring three alternative approaches to building discourse representations. First, we introduce a technique called *DRS-threading*. This technique, which can be viewed as a declarative approach to the standard algorithm, is elegant, efficient, and highly instructive. However, our initial implementation of threading is open to the same sort of difficulties we encountered in Experiment 2 in Chapter **??**. So we abandon threading and turn to a second technique, the lambda-based methods developed in Part I. As we shall see, the techniques and tools developed in our earlier work adapt straightforwardly to DRT; we will be able to reuse the rules in `englishGrammar.pl`, and the required interface is very simple.

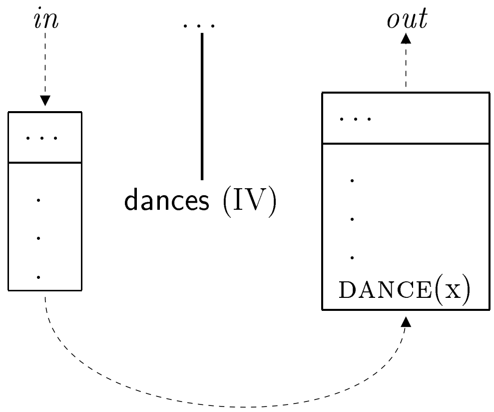
We then change gears. Basic semantic construction is all very well, but how are we to handle quantifier scope ambiguities? As we shall show, the hole semantics approach to underspecification (Chapter **??**) adapts easily DRS languages, and implementing it requires only routine adaptations of our earlier code.

2.1 DRS-Threading

Threading is an interesting technique for building DRSs. For a start, as it is based on the use of *difference structures*, threading is extremely efficient. Difference structure programming uses the difference between incomplete data structures to represent partial results of a computation; the classic Prolog example is the use of difference lists to combine lists efficiently. As we have chosen to represent DRSs in Prolog as a pair of lists, and as we want to combine the small DRSs associated with lexical entries into a big DRS that represents the entire discourse, it seems sensible to experiment with difference structures. But there is an even more compelling reason for thinking about DRS construction in terms of threading: it is enlightening and instructive to do so. Threading embodies a clear and remarkably simple idea—an idea that will enable to peel away the inessential top-down paraphernalia of the standard DRS construction algorithm and reveal the elegant declarative core that lies at it heart. Moreover it gives us (yet another) computational perspective on the idea of context change potential.

When we introduced the standard construction algorithm in Section 1.1, we did so with the help of a series of diagrams showing a parse tree, a DRS, and an arrows linking them. We explained how we moved around the tree in a top-down left-to-right direction, gathering the semantic contributions of the various constituents and packing them into a DRS. Let's change this picture slightly. Think of the parse tree as a rigid structure, perhaps made of wire. Think of the DRS as a little plastic bubble which we can freely slide round this rigid tree (rather like the beads on an abacus, or a children's game). At the start of a 'calculation' or 'game' we place the DRS representing the prior discourse at the top of this tree and then slide it smoothly from node to node, taking care to slide it over every single node in the tree. When the DRS slides over a node (or to put it another way, when a node is *threaded through* the DRS) the node places its semantic contribution inside the DRS.

Here's an example. What happens when we move the DRS over a node labeled by an intransitive verb, say 'dances'? Now, the contribution of an intransitive verb to the overall DRS is just a basic condition; it doesn't introduce a discourse referent. So when the DRS threads its way around a node labeled by 'dances', the following occurs:



Note that the domain of the outgoing DRS is equal to the domain of the ingoing DRS, and that the condition set of the outgoing DRS consists of the conditions of the ingoing DRS *plus* the basic condition introduced by the verb. It is clear that the node has simply made the expected semantic contribution as it was threaded through the DRS.

As far as the basic idea of threading is concerned, there really is not a lot more to say than is suggested by this diagram. All the work in this section is directly based on the intuition of a DRS sliding its way round a parse tree, collecting the semantic contributions as it goes, and we shall simply work out the consequences of this idea for the various syntactic categories. But then, why is this new picture important? After all, sliding a DRS around a tree sounds every bit as procedural as the original presentation of the standard construction algorithm!

However we really have taken an important step forward, for threading focuses attention on what is truly important: at every node there is an *ingoing* DRS, an *outgoing* DRS, and the difference between the ingoing and outgoing DRS is exactly the information that is contributed by the syntactic category. To put it another way: threading is really about thinking in terms of *pairs* of DRSs, for these give us 'before-and-after' pictures of the various stages of semantic construction.

And now for the payoff. If we think in terms of ingoing and outgoing DRSs, the procedural aspects of the standard construction algorithm simply melt away. We are left with its declarative core, and this can be expressed using simple Prolog constraints. To see how, let's return to our 'dances' example.

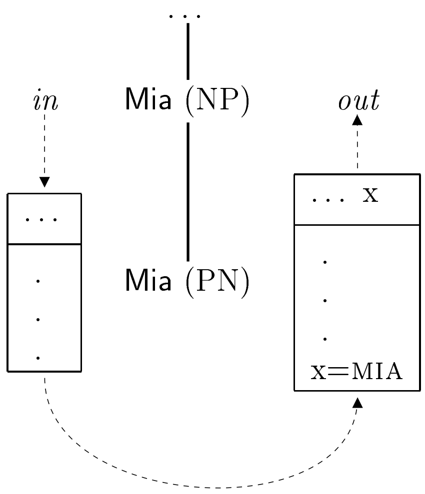
We shall associate each category of the syntactic analysis with a complex Prolog term `DrsIn-DrsOut`, representing the ingoing and outgoing DRSs. So, to capture the effect of the previous diagram, the lexical entry for the intransitive verb 'dances' must be:

```
iv(X,DrsIn-DrsOut)-->
   {
    DrsIn=drs(Dom,Conds),
    DrsOut=drs(Dom,[dance(X)|Conds])
   },
   [dances].
```

The most important point to observe is two unification equations: these constraints constitute the declarative core of the previous diagram. However the reader should also note that we are keeping track of the discourse referent `X` used as the argument of `dance`. This is clearly sensible, but note that we have done so by adding it as an extra argument to the grammar rule and letting the DCG take care of it. Recall that this is essentially the strategy we adopted in Experiment 2 of Chapter **??**, and given the outcome of that experiment, our first hint that trouble is brewing; but let's not worry about it for now. Finally, note that the code can be simplified by replacing `DrsIn` and `DrsOut` by the terms they have to unify with:

```
iv(X,drs(Dom,Conds)-drs(Dom,[dance(X)|Conds]))-->
   [dances].
```

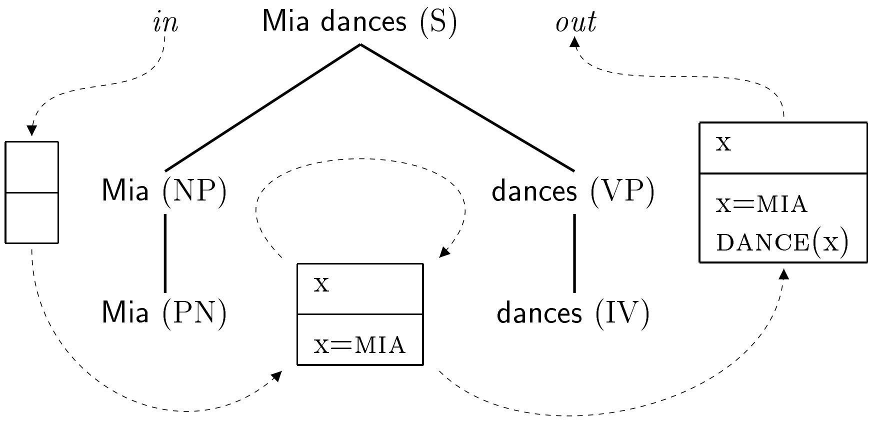
Lets now give a threading analysis of the sentence 'Mia dances'. First, we need to add a lexical entry for the proper name 'Mia'. Proper names introduce both a discourse referent and a condition. The following diagram shows what happens when a node labeled by 'Mia' is threaded through a DRS:



The following Prolog code turns this picture into a simple constraint (once again, note that we've also added an extra DCG argument, Experiment 2 style, to keep track of the discourse referent):

```
pn(X,drs(Dom,Conds)-drs([X|Dom],[X=mia|Conds]))-->
   [mia].
```

That's the situation in the lexicon. So let's now see what happens as the DRS representing the previous discourse slides its way around the parse tree for 'Mia dances' (in the following diagram we assume that this initial DRS is empty):



There are two unary branching rules in this example (NP $\rightarrow$ PN and VP $\rightarrow$ IV), and they are easy to handle in Prolog—they simply pass on the pair of DRSs, since no new information is added. That is, the ingoing DRS of the daughter category is the ingoing DRS of the mother, and the outgoing DRS of the mother category is the outgoing DRS of its daughter:

```
np(X,DrsIn-DrsOut)-->
   pn(X,DrsIn-DrsOut).

vp(X,DrsIn-DrsOut)-->
   iv(X,DrsIn-DrsOut).
```

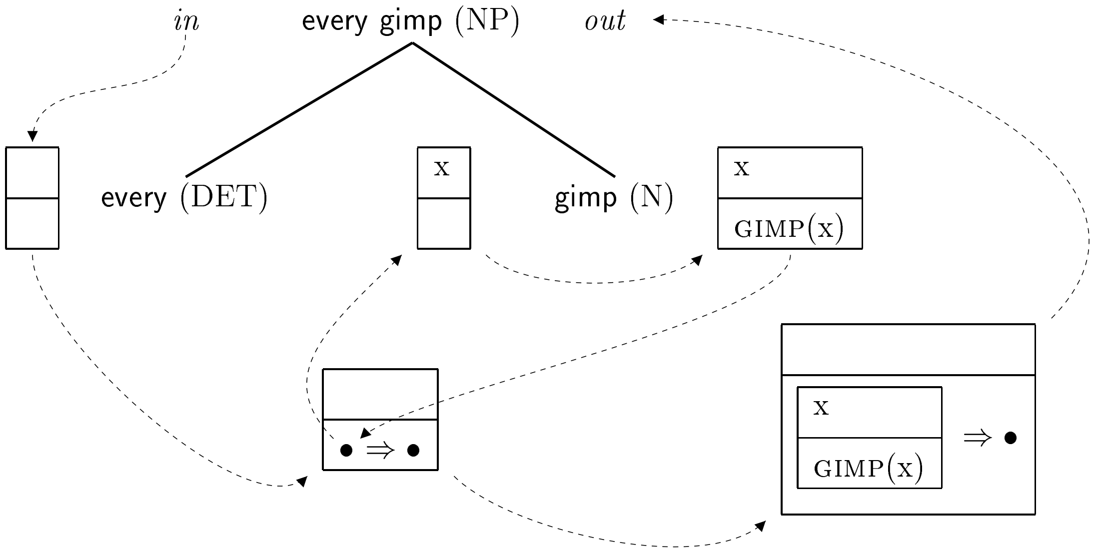
For the binary branching rule S $\rightarrow$ NP VP, the following serves our needs:

```
s(DrsIn-DrsOut)-->
   np(X,DrsIn-Drs),
   vp(X,Drs-DrsOut).
```

Here two points need to be made here. First, note that the outgoing DRS of the NP node (that is, `Drs`) unifies with the ingoing DRS of the VP node, the ingoing DRS of the sentence is the ingoing node of the NP, and the outgoing DRS of the sentence is the outgoing DRS of the VP. These constraints ensure that the information is packed into the DRS correctly—in effect, they perform the same function as the processing insisted on in the standard construction algorithm, but abstract away from its (unnecessarily restrictive) procedural stipulations. Second, note that the extra arguments percolated upwards by the noun phrase and verb phrase are unified; this ensures the correct bindings of the discourse referent with its conditions, very much in the spirit of Experiment 2 of Chapter **??**.

And that completes our threading analysis of 'Mia dances'. But this is a very simple sentence; how does DRS-threading extend to sentences containing determiners such as 'every'. Let's give a threading analysis of 'Every gimp runs'.

The semantic information associated with the the determiner 'every' is a complex structure: an implicational condition consisting of two DRSs. This is shown in the analysis for the noun phrase 'every gimp' (we assume that the ingoing DRS is empty):



The outgoing DRS for ‘every’ is the ingoing one to which implicational condition (that is, an arrow linking two sub-DRSs) has been added. We don’t yet know what the contents of these sub-DRSs are, so we represent each of them with a black hole •. Actually, we *do* know a little more: this implication is a universal quantification, hence the antecedent black hole must contain a discourse referent x; this is indicated by the third threading arrow. How is the condition set of the antecedent to be filled? By the contribution of the noun (‘gimp’ in this example), as fourth threading arrow indicates. In short, once we’ve threaded every node in the noun phrase through the initial DRS, we will have successfully filled in the antecedent black hole and now are ready to pass on the resulting incomplete DRS (incomplete because the consequent is still just a black hole) to the rest of the sentence for further threading. But before going any further, let’s express this noun phrase analysis in Prolog. First, the lexical entry for ‘every’:

```
det(X,DrsIn-DrsOut,RestrIn-RestrOut,ScopeIn-ScopeOut)-->
   [every],
   {
    DrsIn = drs(Dom,Conds),
    DrsOut = drs(Dom,[RestrOut > ScopeOut|Conds]),
    RestrIn = drs([X],[]),
    ScopeIn = drs([],[])
   }.
```

Note that we have added two DRS-threading pairs: `RestrIn-RestrOut`, which builds the DRS for the restriction (the first •), and `ScopeIn-ScopeOut` which builds the DRS for the nuclear scope (the second •). The outgoing DRS equals the ingoing DRS plus a new implicational condition. A new box `RestrIn` is opened in which information about the restriction can be placed, and a new discourse referent is placed in it. Similarly, a new box ScopeIn is provided for the nuclear scope, but no information can be placed in this box by 'every'.

Now for the DCG rule that combines a determiner and a noun to form an NP:

```
np(X,DrsIn-DrsOut,ScopeIn-ScopeOut)-->
   det(X,DrsIn-DrsOut,RestrIn-RestrOut,ScopeIn-ScopeOut),
   noun(X,RestrIn-RestrOut).
```

Quite simply, the noun fills in the restriction information, just as we saw in the previous diagram. Note that that this rule can be simplified without affecting the way it works:

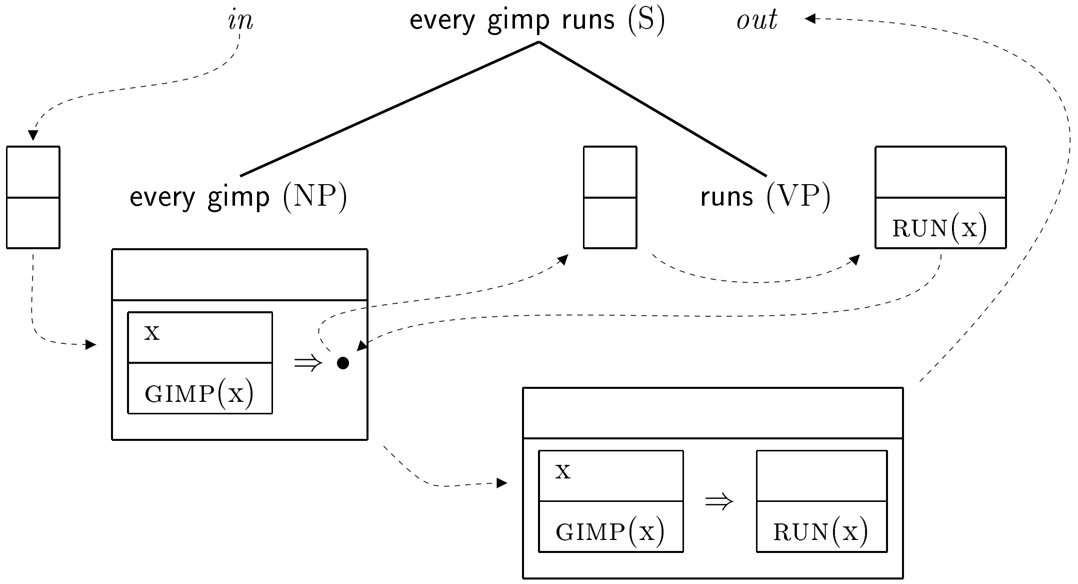
```
np(X,Drs,Scope)-->
   det(X,Drs,Restr,Scope),
   noun(X,Restr).
```

Here Drs stands for the DRS-pair corresponding to the the overall DRS, Restr for the DRS-pair that fills the first hole (the restriction of the quantifier), and Scope for the DRS-pair that fills the second hole, the nuclear scope of the quantifier. Now this is nice—but observe that trouble is on its way. In particular, note that whereas in this rule np has *three* arguments, in the earlier proper name rule used to analyze 'Mia dances' it only had *two* arguments. In short, we are starting to run into the same sort of unpleasantness we encountered in Experiment 2: the need to adapt all our rules to a common format. We will do this shortly (for the small collection of rules we're working with here, it's not difficult) but clearly our DCG-based handling of missing information is raising problems that will require further attention.

Exercise 2.1.1 [Easy] Simplify the DCG rules that deal with determiners and nouns by introducing the discourse referent in the rule for nouns, instead of the rule for determiners.

Exercise 2.1.2 [Intermediate] Draw a picture that illustrates how to give a threading analysis the noun phrase 'a gimp'. Using this picture as a guide, give a DCG rule for the determiner 'a'.

Lets return to our analysis of 'Every gimp runs'. We now have an analysis for 'Every gimp', but this contains a black hole marking the missing nuclear scope information. How do we fill it? The verb phrase takes care of this, as the following analysis shows:



That is, we start threading the VP with an empty DRS and substitute the result for the black hole in the consequent of the implicational condition introduced by 'every'. (Incidentally, note that the rule for 'every' already states that threading for the scope commences with the empty DRS, for it contains the constraint `ScopeIn = drs([],[])`.)

Now the basic idea is clear, but once again Experiment 2 strikes, and to get it working properly we have to revise the DCG rule for sentences (given in our earlier analysis of 'Mia dances') as follows:

```
s(DrsIn-DrsOut)-->
   np(X,DrsIn-DrsOut,ScopeIn-ScopeOut),
   vp(X,ScopeIn-ScopeOut).
```

Putting this in a more readable format we get:

```
s(Drs)-->
   np(X,Drs,Scope),
   vp(X,Scope).
```

This completes our threading analysis of 'Every gimp runs', and all important the ingredients of DRS-threading have now been discussed. To finish off, we'll extend our DCG with some new grammar rules (including the required reformulation of the rule for proper names), see how to make use of the lexicon defined in Chapter ??, and define a simple driver. First, the rules:

```
d(DrsIn-DrsOut)-->
   s(DrsIn-Drs),
   d(Drs-DrsOut).

d(Drs-Drs)-->
   [].

vp(X,DrsIn-DrsOut)-->
   tv(X,Y,Scope),
   np(Y,DrsIn-DrsOut,Scope).

np(X,DrsIn-DrsOut,Drs-DrsOut)-->
   pn(X,DrsIn-Drs).
```

The first two rules allows us to make small discourses by sequencing sentences (we have introduced a new category D for 'discourse'). We also give a DCG rule for combining a transitive verb and a noun phrase to form a verb phrase. Finally, as promised, we have given a revised version of the DCG rule for NP $\rightarrow$ PN so that we have one uniform treatment of noun phrases within the grammar. This new rule may be a little puzzling, and the reader is asked to study it in the following exercise.

Exercise 2.1.3 [Intermediate] Using either pen and paper or a tracer, compare the sequence of variable instantiations this program performs when building representations for 'Mia dances' and 'A gimp dances'. Compare the new rule given for proper names with the corresponding rule of Experiment 2 in Chapter **??**. Make sure you understand how the use of variable doubling in the earlier chapter relates to the code given above.

It is straightforward to make use of the the English lexicon defined in Chapter **??** with the threading DCG, For example, the entry for a transitive verb is simply:

```
tv(X,Y,drs(Dom,Conds)-drs(Dom,[Cond|Conds]))-->
   {
    lexicon(tv,Sym,Phrase),
    compose(Cond,Sym,[Y,X])
   },
   Phrase.
```

It remains to design a driver predicate that feeds the DCG with an ingoing DRS and returns the outgoing DRS. We shall simply give it the empty DRS, but we could just as easily give it the DRS produced by previously analyzed discourse.

```
parse:-
   readLine(Discourse),
   d(drs([],[])-Drs,Discourse,[]),
   printRepresentation(Drs).
```

For instance, suppose we want to build the DRS for the discourse that contains of the sentence 'Vincent snorts' followed by 'A woman collapses'.

```
?- parse.

> Vincent snorts. A woman collapses.

drs([A,B],[collapse(A),woman(A),snort(B),B=vincent])
```

Note that the lists are built from right to left; that is, the most recent information comes first in the list.

And that's DRS threading. What can we say about it at a more general level?

Threading clearly has a lot going for it. For a start, the way it arises as declarative abstraction from the standard construction algorithm is instructive. Moreover, it is suggestive in other ways: it mirrors the idea that the meaning of an utterance lies in its context change potential in way that is reminiscent of dynamic semantics. Recall that dynamic semantics interprets DRSs with respect to *pairs* of embeddings, an input embedding and an output embedding—but whereas dynamic semantics exploit the use of input/output pairs to define an alternative semantics, threading uses the same idea to place constraints of semantic construction. The reader will find it instructive to re-examine our presentation of dynamic semantics in the light of the work of the present section.

But now for the bad side. Our approach to DRS-threading faces two problems. Actually, the first of these, which concerns our treatment of proper names, is more interesting than problematic. As we defined DRS-threading, proper names add a discourse referent and condition to the ingoing DRS. But the ingoing DRS need *not* be the main DRS: it could easily be a sub-DRS (for example, the consequent DRS of an implicational DRS) and this gives rise to an an unwanted result. The discourse referent of a proper name and its condition should float to the top of the DRS, but in our presentation of threading they may stay down at a lower level. This, however, is not a serious problem.

Exercise 2.1.4 [Intermediate] Change the program in such a way that proper names are floated to the top DRS. Hint: use a second DRS pair.

But there is a problem with our work that is potentially far more damaging. Quite simply, all the problems associated with the Experiment 2—the problems which motivated our introduction of the lambda calculus—are back in force and as annoying as ever. Although we solved the problems for the present (very simple) grammar, the solution is clearly Prolog specific and rather ad-hoc. Thus an obvious question faces us. Is it possible to work with variables in DRT in more disciplined, less Prolog oriented way?

2.2 Building DRSs with Lambdas

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 the lambda calculus, but will use λ to mark information missing from DRSs rather than first-order formulas. We often call this combination as *λ-DRT*, and we shall see, it enables us to reuse the tools and techniques developed in Part I.

In order to make this approach work, we are going to have to enrich the box languages with a new construct: the *merge* operation $\oplus$. This combines two DRSs by taking the union of the two universes and the two lists of conditions. For example,

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{boxer}(\mathrm{x}) \\ \textsc{lose}(\mathrm{x}) \\ \hline\end{array} \oplus \begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{die}(\mathrm{y}) \\ \mathrm{y{=}x} \\ \hline\end{array} = \begin{array}{|l|}\hline \mathrm{x\ y} \\ \hline \textsc{boxer}(\mathrm{x}) \\ \textsc{lose}(\mathrm{x}) \\ \textsc{die}(\mathrm{y}) \\ \mathrm{y{=}x} \\ \hline\end{array}$$

There are two reasons for adding the merge. The first is simply that it makes *explicit* something we have been doing *implicitly* all along. In particular, note that whenever we use the standard construction algorithm (or indeed, the threading implementation of the previous section) we take care to add the semantic contribution of the latest sentence to the DRS associated with the discourse so far—after all, that's one of the key ideas of DRT. The merge gives us some natural terminology and notation for talking about this idea. Indeed, it lets us break it down this process into slightly smaller steps, for we can now say that the semantic representation for a discourse is obtained by building a (completely new) DRS for the incoming sentence and then merging this new DRS with the DRS associated with the discourse so far. The second reason is deeper: when used in combination with λ, the merge is precisely the operation on DRS we need to state lexical entries clearly and concisely. We shall shortly see why.

What sort of λ-DRSs (as we usually call them) should be assigned to lexical items? Let's consider some examples. Nouns are associated with the following kind of representation:

'boxer': $\lambda y.\begin{array}{|c|}\hline \\ \hline \textsc{boxer}(y) \\ \hline\end{array}$

'woman': $\lambda y.\begin{array}{|c|}\hline \\ \hline \textsc{woman}(y) \\ \hline\end{array}$

That is, just as with our earlier work with first-order representations, we are using lambdas as a tool to explicitly mark missing information—a tool that has nothing to do with the peculiarities of Prolog, and DCGs, or indeed an other language. In the examples just given, the bound occurrences of y indicate that nouns are associated with DRS—but DRSs that need to be supplied with a term, in the indicated position, to become a full DRS.

The lexical entries for the determiners 'a' and 'every' are more interesting. In particular, they require us to use both lambdas and the merge:

'a': $\lambda P.\lambda Q.\begin{array}{|c|}\hline z \\ \hline \\ \hline\end{array} \oplus P@z \oplus Q@z$

'every': $\lambda P.\lambda Q.\begin{array}{|c|}\hline \\ \hline \begin{array}{|c|}\hline z \\ \hline \\ \hline\end{array} \oplus P@z \Rightarrow Q@z \\ \hline\end{array}$

These are clearly analogous to the lexical entries given for these determiner given in Chapter **??**. For example, we there assigned the following representation to 'every':

'every': $\lambda P.\lambda Q.\forall z(P@z \rightarrow Q@z)$

In both representations, the abstracted variable P marks the missing restriction information, while Q marks the missing nuclear scope. Moreover, the quantificational force of the λ-DRS representation is obtained by using a combination of $\begin{array}{|c|}\hline z \\ \hline \\ \hline\end{array}$ and $\Rightarrow$, in a fashion that is clearly analogous to the use of $\forall z$ and $\rightarrow$ in the other. Note that $\oplus$ is useful because it lets us state which DRS need to be glued together in the course of semantic construction.

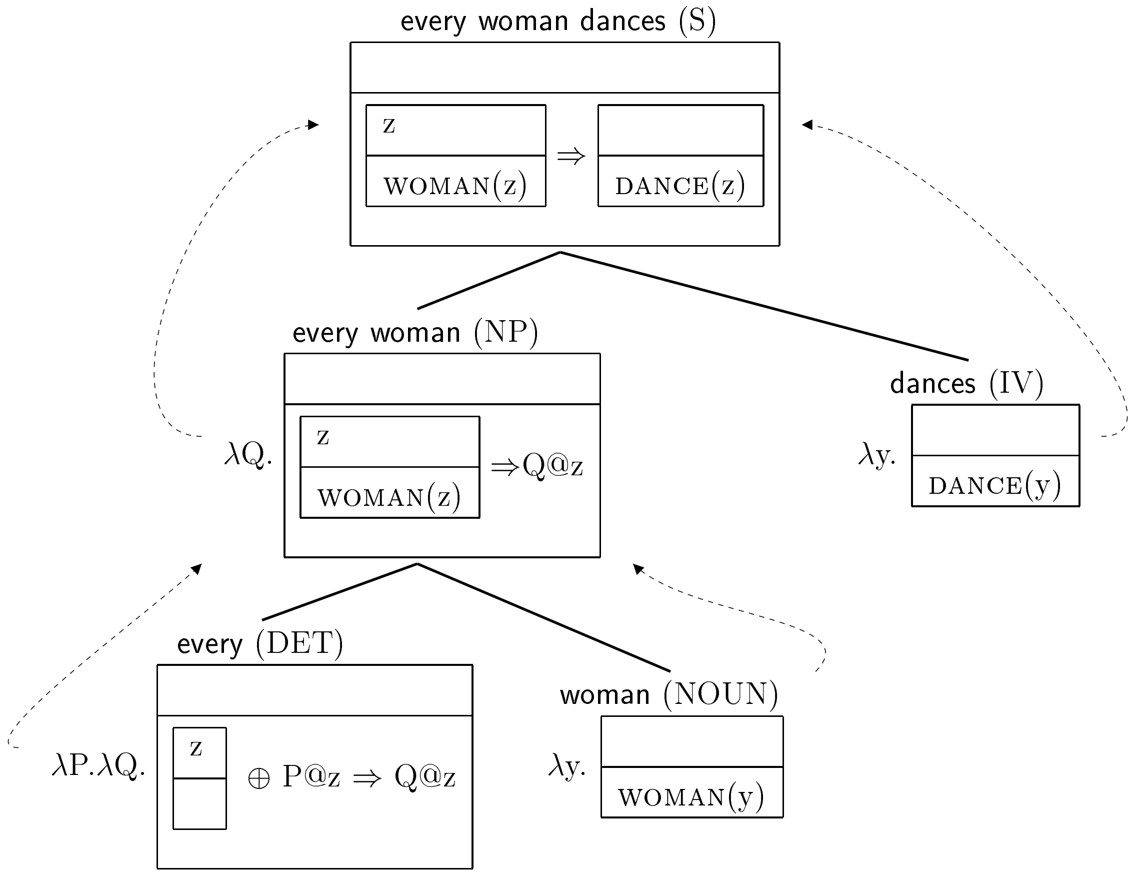
Finally, let's consider the lexical entries for intransitive and transitive verbs:

'dances': $\lambda y.\begin{array}{|c|}\hline \\ \hline \textsc{dance}(y) \\ \hline\end{array}$

‘loves’: $\lambda X.\lambda y.X@\lambda x.\begin{array}{|l|}\hline \\ \hline \text{LOVE}(y,x) \\ \hline\end{array}$

Again, the reader should compare these entries with their analogs in Chapter **??**.

It’s time to put these entries to work. As a sample, here’s how we build a DRS for ‘Every woman dances’:



Note that the fundamental mechanism that drives this construction is the one we are familiar with from Chapter **??**: the lambda calculus plugs things together in the expected way. Note, however, that merge plays an important supporting role. For example, the λ-DRS associated with ‘every woman’ 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 2.2.1 [Easy] Write out the steps involved in obtaining the λ-DRS associated with ‘every woman’ in the above tree.

Exercise 2.2.2 [Easy] Build the DRS representing ‘A woman snorts’ using the lexical representations given above.

Exercise 2.2.3 [Intermediate] State the needed lexical representations, and then build the DRS representation of the donkey sentence 'If a boxer loses, he dies'.

Exercise 2.2.4 [Easy] Suppose that we tried to avoid using the merge by giving 'a' the following lexical entry instead of the one given above:

$$\text{'a': } \lambda P.\lambda Q. \begin{array}{|l|}\hline x \\ \hline P@x \\ Q@x \\ \hline \end{array}$$

This won't work. Why?

In short, the main ideas underlying semantic λ-DRT are those we are familiar with from Part I. Moreover, as we shall now see, implementing λ-DRT in Prolog is rather easy—indeed the only part of the implementation that requires thought is how to handle the merge. Just about everything else is a matter of reusing the tools developed in Part I.

Let's start with the really good news: *the grammar requires no work whatsoever*. A moments thought shows why. Our grammar is a DCG which uses our predicates for functional application to manipulate the representations handed up by the lexical entries. It doesn't much matter to this DCG whether these representations are first-order formulas mixed with lambdas, or DRSs mixed with lambdas. In short, as we claimed in Chapter **??**, lambda-based construction methods make good sense from a grammar engineering perspective.

So what do we actually have to do? Just two things: define the macros for the lexical entries, and wrap everything up in a new driver. Let's start with thinking about the semantic macros. It follows from our earlier discussion that they will look like this:

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

ivSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

modSem(neg,lambda(P,lambda(X,drs([],[~(P@X)])))).
```

But now we have more work to do. Up till now we have not had to use the merge. But what are we to do with merges which cannot be conjured away like this, as is the case for the merges in the lexical entry for the determiner 'a'? One option would be to make use of `append/3` in the semantic macro for determiners, but this doesn't work. Suppose that we use the Prolog variables `P` and `Q` for the λ-DRSs for the 'N' and the 'VP' part, and introduce `X` as the new discourse referent 'a' generates. Then we need to merge the DRSs that result from, say `P@X` and `Q@X`. But from Prolog's point of view, these are not DRSs (for Prolog DRSs are of the form `drs/2`).

So we need to think of something else. 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. That is, we are going to adopt the same strategy we used to implement the substitution based approach to β conversion.

Suppose `B1` and `B2` are DRSs. We can represent the merge of these DRSs as `merge(B1,B2)`. Then the revised macro for indefinite determiners looks like:

```
detSem(indef,lambda(P,lambda(Q,merge(merge(drs([X],[]),P@X),Q@X)))).
```

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(S,[a,boxer,eats,a,big,kahuna,burger],[]), betaConvert(S,B).

B = merge(drs([X],[boxer(X)]),
          merge(drs([Y],[big_kahuna_burger(Y)]),drs([],[eat(X,Y)])))
```

That is, we obtain the merge of a DRS with a DRS that is itself the result of another merge. Now this is correct, but not very readable, so let's write a predicate `mergeDrs/2` that will actually carry out the stipulated merges. Note that as DRSs can be used to build complex DRSs, merges can also appear in complex conditions of DRSs, thus we need the following recursive predicate:

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

With the help of this predicate, we can pursue the following strategy: we first parse a sentence or discourse. This gives us a DRS with explicit merge statements. We then use `mergeDRS` to arrive at simpler DRSs containing no merge statements.

We can now define a driver that parses the discourse and returns the associated DRS for it. By incorporating a call to `mergeDrs`, we ensure that our DRSs are presented to us nicely simplified:

```
parse:-
   readLine(Discourse),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,ReducedDrs),
   mergeDrs(ReducedDrs,Drs),
   printRepresentation(Drs).
```

For example:

```
?- parse.

> Every boxer growls

drs([],[drs([A],[boxer(A)])>drs([],[growl(A)])])
```

And that's all there is to it. The exercises at the end of the section extend this work in various directions.

What can we say of a more general nature concerning λ-DRT? Clearly the approach is natural. For a start, it imitates the Montagovian approach to semantic construction, and extends it to discourse in a very straightforward way; the merge plays a key role here, giving us an operation which can be explicitly invoked to define the required DRSs. Moreover, the fact that we can reuse so much of our earlier work is a powerful testimony both to the robustness of Montague's original methods, and to the naturalness of the ideas underlying DRT.

Exercise 2.2.5 [Intermediate] Extend program `lambdaDRT.pl` with a treatment of the determiner 'no'. (Try to handle sentences like 'No boxer retires'.)

Exercise 2.2.6 [Intermediate] Extend the program to handle adjectives. (Try to handle sentences like 'A rich boxer retires'.)

Exercise 2.2.7 [Intermediate] Extend the program with ditransitive verbs (for example, 'give').

Exercise 2.2.8 [Project] Rework the programs given in this section so that they are based on the use of difference lists.

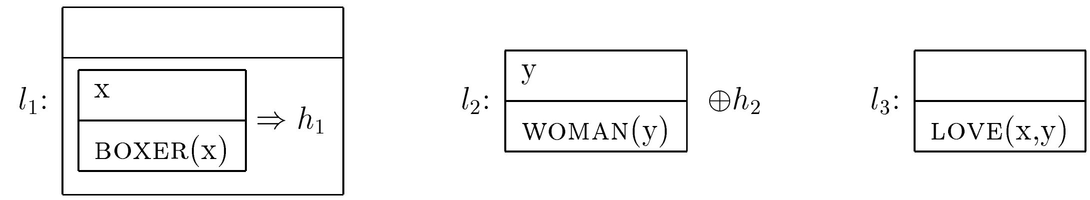
2.3 Underspecified DRSs

Until now we have been concerned with how to use syntactic structure to guide DRS construction. But as we know from Chapter 3, this is only the start of the story, for we also need a way to cope with scope ambiguities. What are we to do here?

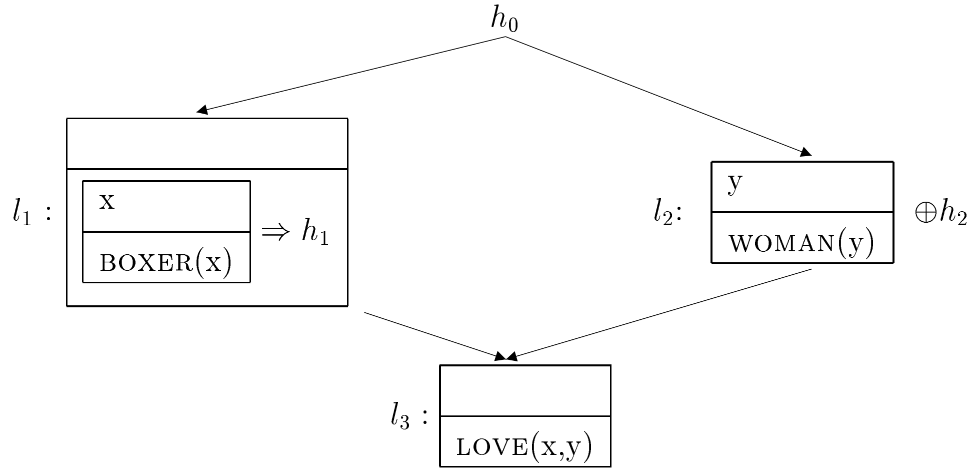
In fact both the storage and underspecification methods developed in Chapter 3 extend straightforwardly to λ-DRT. As far as storage methods are concerned, this is probably clear: from a computational perspective storage methods, essentially required us to work with a rather simple list-based extension of our normal semantic representation. Making these ideas work for DRS-based representations involves no really new ideas, and we ask the reader to explore this option in Exercise 2.3.2. But what about underspecified representations based on hole semantics? In fact, as we shall now show, most of the work has already been done: the underlying ideas transfer straightforwardly, we can reuse the plugging algorithm introduced in Chapter 3, and we can reuse the lexicon and rules for our fragment of English introduced in Chapter 2.

Combining hole semantics with DRS involves two actions: we first need to *unplug* our DRS-language, by permitting DRSs to be built which contain *holes*. Then, we make use of the simple constraint language defined in Chapter 3 that governs the way how the DRS-chunks can be *plugged together*. Let's work though a simple example, 'every boxer loves a woman', to show what's involved.

Unplugging the DRS representation for this sentence yields the following chunks, labeled l_1, l_2, and l_3:



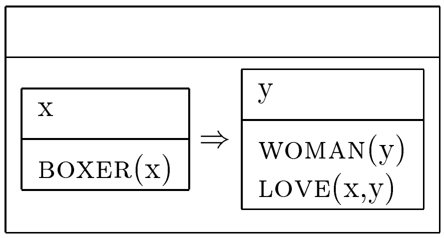
Adding the constraints $l_3 \leq h_1$, $l_3 \leq h_2$ ensures that the main verb is out-scoped by the DRSs introduced by the NPs, and $l_1 \leq h_0$, $l_2 \leq h_0$ enclose all the chunks within the scope domain h_0. Note that we arrive at the essentially the same underspecified representations as in Chapter **??**, but instead of dealing with first-order formula's, we now deal with DRSs.



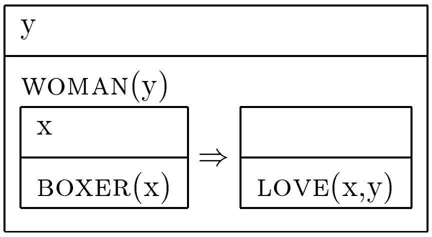
And resolving these underspecified DRS-representations is not different from resolving underspecified first-order representations either. Again, the holes underspecify scope, and in order to give us non-ambiguous interpretations, each hole should be *plugged* with a DRS in such a way that all the constraints are satisfied. In other words, a plugging for a USR is admissible if the instantiations of the holes with labels result in a DRS in which satisfy the constraints. To return to our example, here we have two pluggings:

	h_0	h_1	h_2
P_1	l_1	l_2	l_3
P_2	l_2	l_3	l_1

Plugging P_1 interprets the USR as giving the universally quantifying NP wide scope, out-scoping the indefinite NP. The corresponding DRS is:



Plugging P_2 yields the other reading, where the indefinite NP has wide scope. Here is the corresponding DRS:



That's the idea—so how do we implement it? As in Chapter **??**, a USR is represented in Prolog as `usr(D,L,C)`, where `D` is a list of holes and labels of the USR, `L` a list of the labeled DRSs, and `C` a list of constraints. Here is an example:

```
usr([A,C,E,G,I,J,D,H],
    [A:drs([],[merge(drs([B],[]),C)>D]),
     C:drs([],[boxer(B)]),
     E:merge(merge(drs([F],[]),G),H),
     G:drs([],[woman(F)]),
     I:drs([],[love(B,F)])],
    [leq(A,J),leq(I,D),leq(E,J),leq(I,H),leq(I,J)])
```

As remarked earlier, we can reuse the plugging algorithm introduced in Chapter 3. Moreover, we can reuse the lexicon and rules for our fragment of English introduced in Chapter 2. In fact, all that's left to do is defining the lexical macros and the basic semantic operations.

Let's go through a few examples of macro definitions. They should be essentially familiar to the reader. Here, for example, are the definitions for nouns, proper names, and verbs.

```
nounSem(Sym,lambda(X,lambda(_,lambda(L,usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

pnSem(Sym,_,lambda(P,lambda(H,lambda(M,
      merge(usr([],[M:merge(drs([X],[X=Sym]),L)],[leq(M,H)]),P@X@H@L))))).

ivSem(Sym,lambda(X,lambda(_,lambda(L,usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

tvSem(Sym,lambda(K,lambda(Y,K@lambda(X,lambda(_,lambda(L,
               usr([],[L:drs([],[Cond])],[]))))))):-
   compose(Cond,Sym,[Y,X]).
```

Nouns and proper names introduce one labeled DRS, and impose no constraints. Verbs, on the other hand, introduce a labeled DRS together with a $\leq$-constraint that sets down the scope domain by `leq(L,H)`: both `H` and `L` are lambda arguments, eventually used by other scope introducing elements of the sentence.

Determiners function add two constraints to the USR, one that states that the main verb (labeled L) should be out-scoped by its nuclear scope, and one that ensures that its box stays within the actual scope domain, imposed by H:

```
detSem(uni,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([F,R,S],
                       [F:drs([],[merge(drs([X],[]),R)>S])],
                       [leq(F,H),leq(L,S)]),P@X@H@R),Q@X@H@L)))))).
```

And here is the macro for verb negation. From the kind of constraints it introduces, it is very similar to the definitions of determiners.

```
modSem(neg,lambda(P,lambda(X,lambda(H,lambda(L,
       merge(usr([N,S],
                 [N:drs([],[~S])],
                 [leq(N,H),leq(L,S)]),
             P@X@H@L)))))).
```

By putting together a driver predicate, we combine our underspecified DRSs with the plugging algorithm. Here it is:

```
parse:-
   readLine(Sentence),
   d(Sem,Sentence,[]),
   betaConvert(merge(usr([Top,Main],[],[]),Sem@Top@Main),Reduced),
   mergeUSR(Reduced,usr(D,L,C)),
   printRepresentation(usr(D,L,C)),
   findall(Drs,(plugHole(Top,L-[],C,[]),mergeDrs(Top,Drs)),DRSs),
   printReadings(DRSs).
```

When calling this predicate, Prolog asks the user to type in a sentence, and then gives this sentence to the grammar, which returns a USR. This USR then is β-converted, the stipulated merges are carried out, and all possible pluggings are computed. Here is an example session:

```
?- parse.

> Every boxer or criminal knows a woman.

usr([A:drs([],[merge(drs([B],[]),C)>D]),
     C:drs([],[E v F]),
     E:drs([],[boxer(B)]),
     F:drs([],[criminal(B)]),
     G:merge(merge(drs([H],[]),I),J),
```

```
     I:drs([],[woman(H)]),
     K:drs([],[know(B,H)])],
    [leq(A,L),leq(K,D),leq(C,L),leq(G,L),leq(K,J),leq(K,L)])

Readings:
1 drs([],[drs([A],[drs([],[boxer(A)])v drs([],[criminal(A)])])>drs([B],[woman(B),know(A,B)])])
2 drs([A],[woman(A),drs([B],[drs([],[boxer(B)])v drs([],[criminal(B)])])>drs([],[know(B,A)])])
```

Exercise 2.3.1 [hard] Running the program on coordinated NPs sometimes brings problems. Find out which cases do this, explain why, and propose a solution.

Exercise 2.3.2 [easy] Implement both Cooper storage and Keller storage for λ-DRT. Reuse as much code from Chapter 3 as possible.

Exercise 2.3.3 Discuss the possibilities to integrate an account for scope ambiguities (Storage or Hole Semantics) in the threading approach to DRS-construction.

2.4 Merging into Darkness?

This section will be devoted solely to the problems involved by merging DRSs (and indeed, underspecified DRSs). So far we took for granted that we can carry out any merge-instruction without problems. But we didn't pay attention to special occasions. What would we do, for instance, if we're ought to merge two DRSs that both contain an occurrence of the same discoure referent? Like for example:

$$(1)\quad \begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{walk}(\mathrm{x}) \\ \hline \end{array} \oplus \begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{talk}(\mathrm{x}) \\ \hline \end{array}$$

If we take our initial definition of Section 8.2, which says that we have to take the unions of the discourse referents and the conditions, we'll get the following DRS:

$$(2)\quad \begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \textsc{walk}(\mathrm{x}) \\ \textsc{talk}(\mathrm{x}) \\ \hline \end{array}$$

The two discourse referents, because we gave them the same name, collapsed into one! Before thinking whether this would be a correct way of dealing with such cases, we should answer the following question: Is it possible that we, in the course of DRS construction, arrive at such expressions? If we can answer this question negatively, we don't need to bother about it (because we're basically interested in the computational aspects of DRS-construction, and less in its theoretical considerations).

Interestingly enough, such cases appear. A classic example is coordination: '`A woman walks and a woman talks`' gives exactly the initial representation (1) above. But is the reduced DRS in (2), that describes that there is a woman that walks and talks, the final representation we would like to have? Clearly not! So there is something dramatically wrong with our merge-operation, that needs to be fixed.

Exercise 2.4.1 Think of other examples in natural language (non-coordination) where this problem appears.

More to be added...

Software Summary of Chapter 2

`threadingDRT.pl` Implementation of the Johnson & Klein's threading technique for constructing discourse representation structures. (page 177)

`mainLambdaDRT.pl` Building DRSs with lambdas. (page 180)

`semMacrosLambdaDRT.pl` The semantic macros for lambda-DRT. (page 181)

`mergeDRT.pl` The predicates for the DRS-merge. (page 182)

`mainDRTU.pl` Main program for DRT Unplugged. (page 183)

`semMacrosDRTU.pl` Semantic macros for DRTU. (page 184)

Notes

The idea of DRS threading was first introduced by Johnson & Klein (1986). This elegant paper remains one of the best discussions the semantic construction in DRT and we strongly recommend it to our readers. Threading is combined with lambdas in Johnson and Kay (Johnson and Kay 1990); this paper addresses a number of important grammar engineerinng issues for computational semantics—required reading.

There are a number of "Box + λ" systems. The earliest we know of is due to Frey, who proposes an implementation of DRS construction based on the f-structures of Lexical Functional Grammar (Frey 1985; Wada and Asher 1986). This bottom-up approach specifies partial DRSs for lexical entries in much the same spirit as we do with lambdas, and includes a modest treatment of quantifier scope ambiguities. More recent approaches include λ-DRT (Bos et al. 1994; Kohlhase et al. 1996) and the system of Asher (1993); moreover, Dynamic Montague Grammar (Groenendijk and Stokhof 1990) has much in common with these systems. Most Box + λ systems make use of some kind of merge operator; an explicit merge operation seems to have been first introduced into DRT by Zeevat (1989). The theoretical aspects of Box + λ systems have been increasingly explored in recent years; Muskens (Muskens 1996) is probably the best starting point for readers interested in this topic. This (very readable) paper shows how to construct Box + λ system in standard type theory.

Combining boxes with lambdas 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. 1994 for an account of this application).

Chapter 3

Pronoun Resolution

Pronoun resolution is an extremely lively and fascinating area and it has drawn strong interests from artificial intelligence, computational, and theoretical linguistics. In this chapter we will investigate and combine different approaches to pronoun resolution, by extending our Prolog programs for working with DRSs from Chapter 8. We will look at both linguistic and AI-oriented procedures for pronoun resolution.

3.1 The Nature of Pronouns

A lot of factors come together in processing pronouns. In Chapter 7 we already demonstrated how the geometrical structure of DRSs constrains the resolution of pronouns. We will certainly make use of this observation in our implementation of pronoun resolution, but it is overwhelmingly wrong to think that only the structure of discourse, as we canned them in DRSs, will determine the correct antecedent for a pronoun. Discourse structure undoubtly plays an important part in pronoun resolution, but goes far beyond the simple structures we assigned to discourses so far.

In fact, there is a lot more to pronoun resolution than the structure of discourse. Obviously, syntactic and semantic agreement play a role, depending on the natural language that we want to analyze (even closely related language as Dutch, German and English show significant differences in behavior). But also the way human beings communicate affects resolution: normally people talk about one thing at a time. World knowledge is important too, which requires, eventually, to integrate some kind of inference component in our algorithm for pronoun resolution. (We will do this in Chapter 11.)

Pronouns are, by their nature, context-sensitive expressions, and the way they appear syntactically (in some language they are simply left out—this phenomena is known as pro-drop or zero anaphora) and operate semantically varies in a number of ways. We won't treat all of this kinds, this goes far behind our purposes—instead we will concentrate on, admittingly, the most simplest cases of pronouns, the third person singular personal pronouns 'he/him', 'she/her', and 'it', and their reflexive forms 'himself', 'herself', and 'itself'.

We will only consider their *anaphoric* use (pronouns that relate back to other objects in the discourse so far processed), as opposed to their *deictic* use (relating to objects in a situation, for instance by pointing at something), or *cataphoric* use (when pronouns refer to items not yet mentioned in the text, as in 'After he won the match, Butch left town.'). Neither will we discuss nor implement pleonastic use of pronouns as in 'It's about nine o'clock in the morning'.

As we have already mentioned in the introductory chapter to DRT, a discourse referent can play the role of antecedent for an anaphor only if it is accessible. Recall that DRT predicts that in the following examples anaphoric links are allowed.

(1) A woman snorts. She collapses.

(2) Mia ordered a five dollar shake. Vincent tasted it.

However, by changing the examples slightly, trying to establish anaphoric links results in weird or even unacceptable discourses.

(3) Every woman snorts. She collapses.

(4) Mia didn't order a five dollar shake. Vincent tasted it.

Universally quantified noun phrase, such as 'every woman', and indefinite noun phrases in the scope of negation, introduce their discourse referent in a subordinated DRS, and are hence not accessible as antecedents for pronouns in following sentences.

But the structure of DRSs is, although a very important one, by no reason the *only* constraint on pronoun resolution. There is a number of other constraints. An obvious one is grammatical agreement: in English, pronouns come with a gender and number feature and can only refer to antecedents that carry the same features values. Consider

(5) Mia ordered a five dollar shake. It tasted delicious.

where the pronoun it can only refer to the five dollar shake, although there is in principal another discourse referent available: the one introduced by the proper name Mia. But since this is a name for a female entity, one would normally uses the correct pronoun she to refer her. This sound obvious, but languages behave quite differently with respect to gender agreement.

Pronouns notoriously introduce ambiguity. Often there is more than one interpretation possible.

(6) Butch threw a TV at the window. It broke.

Our program will eventually generate loads of readings (too many to deal with in an efficient natural language processing system). Some of these are more likely than others, so one way of dealing with it is adding preferences to the readings.

World knowledge might help us in some cases to select an antecedent. So, in (7) the pronoun it is likely to refer to a vase introduced in the first sentence (and not to the linguistically equally well the wall).

(7) Butch threw a vase at the wall. It broke.

Selectional constraints play a role here. The pronoun in it broke can only refer to breakable things. Here is another example that illustrates this.

(8) Butch walks into his modest kitchen. He opens the refrigerator.
He takes out a milk and drinks it.

This might sound intuitively correct, matters are however, far more complicated. Consider for example 'drinking a cup of coffee'. Here, the cup is the argument of drinking, but selectional constraints would erroneously rule out this, as strictly spoken, only *beverages* can be drinken.

It should be clear that incorporating world knowledge is far more difficult than integrating pure linguistic constraints in the resolution component. We will therefore concentrate on purely linguistic constraints first.

Exercise 3.1.1 If you're not a native speaker of English, find out how pronouns are used in your mother tongue, and compare it with the use of pronouns in English.

3.2 Implementing Pronoun Resolution in DRT

This section discusses a basic implementation of pronoun resolution in DRT. We will focus on the linguistic sources to constrain resolution. As noted above, we take into account the geometrical structure of DRSs. Further, we will use *gender* as sortal information to rule out some anaphoric links. Normally, in English, 'he' refers to male, 'she' to female, and 'it' to non-human entities (there are clear exceptions to this rule, but we won't deal with these in our implementation). Recall the grammar rule for handling pronouns we have:

```
pro(Pro)-->
  {
   lexicon(pro,Gender,Phrase,Type),
   proSem(Gender,Type,Pro)
  },
  Phrase.
```

This rule declares `Phrase` a pronoun (i.e., being a member of the syntactic category 'Pro') if it is listed in the lexicon as such, and associates the gender information (male, female, or non-human) and type information (reflexive or non-reflexive) with it. Further, the semantic macro `proSem/3` generates the semantics according to this gender and type information. (We'll define this predicate shortly below.)

The lexicon entries are divided into non-reflexive and reflexive pronouns:

```
lexicon(pro,male,[he],nonrefl).
lexicon(pro,female,[she],nonrefl).
lexicon(pro,nonhuman,[it],nonrefl).
lexicon(pro,male,[him],nonrefl).
lexicon(pro,female,[her],nonrefl).

lexicon(pro,male,[himself],refl).
lexicon(pro,female,[herself],refl).
lexicon(pro,nonhuman,[itself],refl).
```

This is how pronouns are integrated in our fragment of English grammar. But we face a practical problem when resolving pronouns. As they are introduced in the lexicon, we don't have access to the context and cannot determine their antecedents. How are we to cope with this?

Our strategy will be the following: we will 'mark' the discourse referents that are introduced by pronouns, and postpone unifying them with candidate antecedents (for example, when we have parsed the entire discourse). So let's extend our Prolog DRS language with the notation

```
alfa(X,Type,Gender,Drs)
```

where `X` is the (still unknown) value of an antecedent discourse referent, `Type` either `refl` (reflexive) or `nonrefl` (non-reflexive), and `Gender`, either `male(X)`, `female(X)`, or `nonhuman(X)`. The `Drs` in this notation, is the DRS in which the pronoun is used.

We'll call such a term an α-DRS. For example, she snorts is represented by the following α-DRS:

```
alfa(Y,nonrefl,female(Y),drs([],[snorts(Y)]))
```

or in our familiar boxed notation:

$$\alpha_{\mathrm{y}} \begin{array}{|l|}\hline \\ \hline \textsc{snort}(\mathrm{y}) \\ \hline \end{array}$$

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 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 `alfa` (for 'anaphoric') binder rather than with `lambda`. The semantic macro that generates the α-DRS is:

```
proSem(Gender,Type,lambda(P,alfa(X,Type,Cond,P@X))):-
   compose(Cond,Gender,[X]).
```

Before we start thinking about how to implement anaphora resolution, let's try out the parser on the discourse 'A woman snorts. She collapses.' by posing the following Prolog query:

```
?- d(B,[a,woman,snorts,she,collapses],[]).

B = merge(
          merge(
                merge(
                      drs([A],[]),
                      drs([],[woman(A),female(A)])
                     ),
                drs([],[snort(A)])
               ),
          alfa(B,nonrefl,female(B),drs([],[collapse(B)]))
         )
```

In box-notation this Prolog-expression is translated as:

$$((\begin{array}{|c|}\hline \mathrm{a} \\ \hline \\ \hline\end{array} \oplus \begin{array}{|l|}\hline \\ \hline \textsc{woman}(\mathrm{a}) \\ \textsc{female}(\mathrm{a}) \\ \hline\end{array}) \oplus \begin{array}{|l|}\hline \\ \hline \textsc{snort}(\mathrm{a}) \\ \hline\end{array}) \oplus \alpha_{\mathrm{b}} \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{b}) \\ \hline\end{array}$$

There are some DRSs in the output that still have to be merged, and there is also an α-DRS introduced by the pronoun she. To arrive at a pure DRS (a DRS that contains no merges or α-expressions) we need to carry out procedures that resolve these two kinds of expressions. 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 (it is not specified for α-expressions). However, it turns out that it is most efficient to combine the two resolution procedures into one predicate, because for both of them we have to traverse the entire structure of the DRS; combining pronoun resolution with merge reduction kills two birds with one stone.

So let's define a predicate `resolveDrs/1` that will do this for us. Recall that a discourse referent of a DRS B_1 is accessible from DRS B_2 when B_1 subordinates B_2, or when B_1 equals B_2. (For the definition of subordination, see Chapter 7.) The predicate `resolveDrs/1` encodes this concept of accessibility in a fairly transparent way.

The predicate `resolveDrs/1` works with difference lists of stacks of DRSs. The input list is a stack of DRSs, of which the first item is the DRS currently under examination, and the other all subordinating it. (That means, if we work on a DRS which is the consequent of a conditional, the main DRS and the DRS of the antecedent are on this list.) The output list is a copy of the input list, but with the merges and instructions to resolve pronouns carried out. So, eventually (if `resolveDrs/1` succeeds), the output list will contain ordinary DRSs. Here are the three and only clauses that define it:

```
resolveDrs([merge(B1,B2)|A1]-[drs(D,C)|A3]):-
        resolveDrs([B1|A1]-A2),
        resolveDrs([B2|A2]-[drs(D2,C2),drs(D1,C1)|A3]),
        append(D1,D2,D),
        append(C1,C2,C).

resolveDrs([alfa(Referent,Type,Gender,B1)|A1]-A2):-
        potentialAntecedent(A1,Referent,Gender),
        resolveDrs([B1|A1]-A2).

resolveDrs([drs(D1,C1)|A1]-A2):-
        resolveConds(C1,[drs(D1,[])|A1]-A2).
```

The first clause handles the merge by resolving its argument DRSs (after all, they can contain merges or α-DRSs), and then concatening the lists of discourse referents and conditions. Note that `B1` is put on the stack before resolving `B2`, so the discourse referents of `B1` will become automatically accessible for `B2`.

The second clause carries out pronoun resolution. It takes a DRS from the accessible DRSs hold in list A1, and tries to find a unification of the α-bound variable and an accessible discourse referent with the help of the following predicates:

```
potentialAntecedent(A,X,Gender):-
   member(drs(Dom,Conds),A),
   member(X,Dom),
   compose(Gender,Symbol1,_),
   \+ (
         member(Cond,Conds),
         compose(Cond,Symbol2,[Y]),
         Y==X,
         \+ consistent(Symbol1,Symbol2)
      ).
```

The predicate potentialAntecedent/3 succeeds only if there is an accessible discourse referent (here this is X, a discourse referent of one of the DRSs on the stack A), such that the gender information is not inconsistent with the conditions imposed on discourse referent X. By backtracking on the member/2 goals, it could find more potential candidates. Note that we make use of consistent/2, a predicate that we defined in Chapter 6.

The third and last clause of resolveDrs/1 deals with ordinary DRSs, and uses another predicate resolveConds/2 that works through the DRS-conditions:

```
resolveConds([~B1|Conds],A1-A3):-
        resolveDrs([B1|A1]-[B2,drs(D,C)|A2]),
        resolveConds(Conds,[drs(D,[~B2|C])|A2]-A3).

resolveConds([B1 > B2|Conds],A1-A4):-
        resolveDrs([B1|A1]-A2),
        resolveDrs([B2|A2]-[B4,B3,drs(D,C)|A3]),
        resolveConds(Conds,[drs(D,[B3 > B4|C])|A3]-A4).

resolveConds([B1 v B2|Conds],A1-A4):-
        resolveDrs([B1|A1]-[B3|A2]),
        resolveDrs([B2|A2]-[B4,drs(D,C)|A3]),
        resolveConds(Conds,[drs(D,[B3 v B4|C])|A3]-A4).

resolveConds([Cond|Conds],[drs(D,C)|A1]-A2):-
        compose(Cond,_Symbol,Arguments),
        simpleTerms(Arguments),
        resolveConds(Conds,[drs(D,[Cond|C])|A1]-A2).
```

```
resolveConds([],A-A).
```

The clauses of `resolveConds/2` work recursively through the DRS-conditions, by placing sub-DRSs on the stack of DRSs and calling back to `resolveDrs/1`.

So, what left to do is defining a driver predicate that ties together the parser, lambda-DRT, and our pronoun resolution module:

```
parse:-
   readLine(Discourse),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,Drs),
   resolveDrs([Drs]-[ResolvedDrs]),
   printRepresentation(ResolvedDrs).
```

After consulting the complete program (`mainPronounsDRT.pl`) we test it as follows:

```
?- parse.

> A woman snorts. She collapses.

drs([A],[collapse(A),snort(A),female(A),woman(A)])
```

In short, we now have a simple implementation of pronoun resolution in DRT. We will shortly see that our implementation has a number of problems. Before coming to that, the reader is asked to do the following exercises.

Exercise 3.2.1 [easy] Imagine this program processing 'If a man eats a big kahuna burger, he enjoys it.'. How many readings do you think it will generate? Why? And how many readings will the program give for 'Vincent knows Butch. He likes him.'?

Exercise 3.2.2 [hard] The current implementation of `resolveDrs/1` makes use of `append/3`. Explain why this is not efficient and make the program more efficient by using difference lists to concatenate lists.

A Problem

Pronouns are resolved on the basis of its gender information. A discourse referent is considered a proper candidate antecedent for a pronoun if its sortal information (as presented in the ontology of nouns, Chapter 6) is not inconsistent with the gender information specified for the pronoun. To perform this test, we use the `consistent/2` predicate developed in Chapter 6.

This works well for male and female pronouns. But it doesn't work for nonhuman pronouns for a simple reason: the sort NONHUMAN is not part of our ontology. So, in the current setting, there are no sortal constraint on the pronoun 'it'. Can't we just add NONHUMAN to our ontology of nouns? This is difficult? What would it be? We know what it is not (human), but it can be a plant, an animal, an artifact, an abstraction, and so on.

Actually, the only thing we want to say about the sort NONHUMAN is that is disjoint from HUMAN. We can't state this in the lexicon directly (at least not in the way we designed the lexicon). The trick to incorporate disjointness information is to directly code in the predicate `consistent/2`:

```
consistent(X,Y):-
   generateIsa(I),
   generateDisjoint(I-Isa,Disjoint),
   \+ inconsistent(X,Y,Isa,[disjoint(human,nonhuman)|Disjoint]).
```

This works. Now the neuter pronouns only match with nonhuman objects. There is another—related—problem. We'll solve this in Chapter 10, but the reader is asked to get aware of this problem by trying the following exercise.

Exercise 3.2.3 [easy] Try the program (`mainPronounsDRT.pl`) on the discourse 'A boxer growls. She collapses. He dies.', and explain what is wrong with the analysis our program gives for it. Revise the semantic macro `proSem/3` to do away with this problem.

3.3 Adding Reflexive Pronouns

Another unavoidable requirement in pronoun resolution is to cope with the difference in anaphoric behavior between reflexive and non-reflexive pronouns. Reflexive pronouns appearing in object NP position of a verb phrase can only be anaphorically linked to the subject NP of this verb phrase:

(9) Vincent goes to the toilet, and Jules enjoys himself.

Non-reflexive pronouns show a complementary behavior to reflexive pronouns: they *cannot* refer to their subject! The following examples make this fact clear:

(10) Vincent enters the restaurant, and Jules watches him.

In sum, pronouns obey rules of binding, and we will incorporate these constraints in our resolution algorithm. We will make use of the fact that a non-reflexive pronoun and its antecedent may not occur in the same simplex sentence, and that a reflexive pronoun and its antecedent should do so. We appeal to this linguistic rule of thumb purely on a "semantic representational" basis: non-reflexive pronouns with discourse referent x cannot be identified with a discourse referent y if both appear in a condition P(x,y) introduced by the transitive verb. In contrast, reflexive pronouns *require* such a condition for a suitable antecedent.

We will refine our Prolog program with a treatment of reflexive pronouns, by appealing to an extra test in our resolution algorithm. This test is implemented by the predicate `properBinding/3`). Here is the revised clause for the predicate `resolveDrs` that integrates this new predicate:

```
resolveDrs([alfa(Referent,Type,Gender,B1)|A1]-A2):-
   potentialAntecedent(A1,Referent,Gender),
   properBinding(Type,Referent,B1),
   resolveDrs([B1|A1]-A2).
```

What does `properBinding/3` do? If the pronoun is of reflexive type, it succeeds if the binding constraints for reflexives are not violated. (This is carried out by `reflexiveBinding/2`).

```
properBinding(Type,X,Drs):-
        Type=refl,
        reflexiveBinding(X,Drs).
```

If, on the other hand, a pronoun is non-reflexive, we try to proof if the constraints for reflexive pronouns are not violated (by using the same predicate `reflexiveBinding/2`), and if this is the case, use the `!, fail` Prolog combination to let `properBinding` fail. Otherwise, it will succeed.

```
properBinding(Type,X,Drs):-
        \+ Type=refl,
        (
            reflexiveBinding(X,Drs),
            !, fail
        ;
            true
        ).
```

In the case of a reflexive binding constraints, we would like to find a basic condition in which the two discourse referents corresponding to the subject and object (the reflexive pronoun) of the transitive verb appear. If we find such a condition we stop, otherwise we continue by recursing through the list of other conditions of the DRS.

```
reflexiveBinding(X,[Basic|Conds]):- !,
        (
            compose(Basic,_Sym,[Subj,Obj]),
            Subj==Obj,
            X==Obj, !
        ;
            reflexiveBinding(X,Conds)
        ).
```

If we don't find such a condition, then the test fails (so we we're not allowed to use a reflexive pronoun here). There are two ways in which it could fail: when finishing checking all the DRS-conditions, or when the argument DRS of the relexive pronoun is another α-DRS, a merge, or a negation. (In the second case the reflexive pronoun is surely not in object position, so we fail here too). Here is the code:

```
reflexiveBinding(_,[]):- fail.
reflexiveBinding(_,alfa(_,_,_,_)):- fail.
reflexiveBinding(_,merge(_,_)):- fail.
reflexiveBinding(_,~_):- !, fail.
reflexiveBinding(X,drs(_,Conds)):-
   reflexiveBinding(X,Conds).
```

This completes our way of dealing with reflexive pronouns. In principle, it works for most of the examples you will encounter in texts. But it also has it shortcomings. We only discussed the interaction of reflexive pronouns in transitive verbs, but we didn't deal with the ditransitive verb. Extending our analysis of reflexive pronouns to ditransitive verbs is not too hard, and is left as an exercise to the reader.

Exercise 3.3.1 [easy] Extend the program to handle reflexive pronouns appearing as arguments of ditransitive verbs, as in 'Vincent walks to the bar and pours himself a drink' and 'Vincent has given himself a little pep talk'.

Nor did we consider cases where reflexive pronouns appear with nouns. Reflexive pronouns are fine to use in combination with nouns that express *information*, such as 'story', 'episode', 'movie', 'picture', 'image', and so on. Here is an example that shows this behavior:

(11) Mia hates every episode about herself.

Our analysis of reflexive pronouns incorrectly rejects this sentence. Notice that for these cases, a use of a non-reflexive pronoun seems to be permitted too.

Exercise 3.3.2 [hard] Examine the example above and similar ones, explain why our program deals with them incorrectly, and sketch a solution that overcomes this problem.

3.4 The Focusing Algorithm

So far we have implemented a pronoun resolution algorithm that does its work under the supervision of syntactic and semantic constraints. This won't be enough in serious discourse analysis, though. Each new sentence in a discourse introduced new entities in the discourse that can be picked up by pronouns used later in the text. For this simple reason, the constraints that our resolution algorithm appeals to cannot avoid the problem of producing an explosion of potential readings. This leads us to one obvious choice to deal with this: adding preferences to some of the solutions we produce.

How are we going to do this? Various proposals are available (see also the Notes at the end of this chapter for a brief overview). We choose to extend our pronoun resolution program with the focusing algorithm (due to Candy Sidner). The task of the focusing algorithm is to predict an antecedent for a pronoun. The syntactic or semantic constraints we already have confirm or reject this choice.

What is focusing and focus? Given a discourse or a dialogue, the speaker or speakers talk about something, one thing at a time. The element or object of discourse on which the attention of the speakers is centered is called the *focus*. The focus is one of the very important threads that makes a series of sentences a coherent piece of text, i.e., a discourse. *Focusing* is the process of keeping track (and updating) the focus of the discourse.

What has focusing to do with pronoun resolution? Well, speakers who talk in several sentences about one focus, do not re-introduce the focus in its full syntactic form in each sentence. Instead they use pronouns (or other anaphoric expressions) under the assumption that the focus is part of the shared knowledge of the participants in the conversation.

We will distinguish two kinds of foci in a discourse: the discourse focus, and the actor focus. Here is an example that shows the use of discourse focus:

(12) The Captain pulls a gold wrist watch out of his pocket. This watch was first purchased by Butch's great-granddaddy. It was bought during the First World War in a little general store in Knoxville, Tennessee.

The three sentences of this discourse give information about the focused element: the gold watch. It is introduced with an indefinite noun phrase in the first sentence, and referred to by 'this watch' and 'it' in the succeeding sentences. Note that there also noun phrases introduced in the text that were not referred to in succeeding sentences (the Captain, his pocket, Butch's great-granddaddy), and further notice that each new sentence makes a reference to the discourse focus.

In addition to the *discourse focus*, focusing must take into account the actors of the discourse. We will call this the *actor focus*. Pronouns in agent position tend to refer to the actor focus, whereas pronouns in other position mostly co-refer with the discourse focus. Sometimes the actor focus can become the discourse focus when no other candidate is available, but under normal circumstances it plays a subordinated role in focusing.

The actor focus is needed besides the discourse focus because the focus of the discourse is often distinguished from the actor, and both can be referred to with pronouns. Here is an example that illustrates this:

(13) Jules grabs the burger. He takes a bite of it.

Here the discourse focus is assigned to the burger, the actor focus is Jules. The occurrence of 'he' refers to Jules, the pronoun 'it' to the burger.

The final thing to mention on focusing is that focus can shift from one object to another in the course of a text or dialogue. The following example shows how the discourse focus changes from the hammer to the chainsaw (the actor focus remains assigned to Butch):

(14) Butch picks up a big destructive-looking hammer, then discards it: not destructive enough. He picks up a chainsaw, thinks about it for a moment, then puts it back.

The last two occurrences of 'it' both refer to the chainsaw, not to the (yet out of focus) hammer.

Both the discourse and actor focus are ordinary discourse referents, but we have to distinguish them from the other referents to signify their special status. We'll do this by adding a new structure to the DRSs we have worked with so far. Ordinary DRSs consists of a set of discourse referents `D` and a set of conditions `C`, and are represented in Prolog as `drs(D,C)`. From now on we will also consider Focus-DRSs, that consists of the familiar set of discourse referents, the actor focus `AF` (a stack of discourse referents), the discourse focus `DF` (a stack of discourse referents), and the familiar set of DRS-conditions. In Prolog we will code Focus-DRSs as `drs(D,AF,DF,C)`, where both the actor and discourse focus are lists of discourse referents.

We will transfer this newly structured DRSs also to our familiar box-notation. So the Focus-DRS for 'Butch picks up a hammer. He discards it.' is

x y
x
y
x=BUTCH HAMMER(y) PICK-UP(x,y) DISCARD(x,y)

where x is the actor focus (Butch), and y the discourse focus (the hammer picked up by Butch). Of course, in a progressing discourse, there can be more of these labels, as the focus of attention can shift. Continuing this discourse with 'He picks up a chainsaw' triggers a shift of the discourse focus.

x y z
x
z y
x=BUTCH HAMMER(y) PICK-UP(x,y) DISCARD(x,y) PICK-UP(x,z) CHAINSAW(z)

The discourse focus now has two elements on its stack. But note that z is the first element of the stack, representing the *current* discourse focus.

Note that Focus-DRSs can play a role on the subordinated level as well. For example, the conditional 'If a customer eats a big Kahuna burger, he likes it.' leads to the construction of the following Focus-DRS:

x y		
x		
y	⇒	
CUSTOMER(x) BIG-KAHUNA-BURGER(y) EAT(x,y)		LIKE(x,y)

This is the basic idea of focusing—we will get a more detailed understanding while integrating the focusing algorithm directly into our DRT-implementation for pronoun resolution. This algorithm consists of three procedural steps:

1. choose foci based on the first sentence of the discourse
2. use foci to resolve pronouns
3. update the foci on the basis of the current DRS

Step 1 guesses the foci on the basis of syntactic and semantic structure of the first sentence. Step 2 involves an extension to the resolution algorithm we have so far—we will give preference to potential antecedents that are in focus. Step 3 retains or resets the focus, using the information how pronouns got resolved. The focus will be moved to another discourse referent if the discourse referent previously in focus is not referred to in the current sentence.

The choose the foci, we will make an initial prediction based on simple heuristics. These, we call it the expected foci, will either be confirmed or rejected by step 3 of the algorithm. We might be wrong in choosing an expected discourse focus (there might be more than one choice), but still the prediction helps us to rank the readings on preference.

Step 1: Choosing Foci

Let's first have a closer look at step 1. How do we identify foci? For the actor focus, this turns out to be rather simple: it is the agent of the sentence if there is one (when the agent of the next sentence is a pronoun, the actor focus is usually the antecedent).

This idea is implemented straightforwardly as follows (we will discuss the predicate `role/3` below):

```
expectedActorFocus(drs(Dom,Conds),Focus):-
   member(Focus,Dom),
   role(agent,Focus,Conds).
```

Determining the discourse focus is somehow more problematic. It boils down to characterizing what is talked about in a discourse. Language seems to have different phenomena for marking focus, deviating strongly from language to language. In English, there-insertion, clefts, and the prosodic structure of the sentence indicate what the sentence is about. Further, usage of grammatical roles give a clue.

The modest fragment of English that we cover does not include clefts or there-insertion constructions (actually, clefts are rarely used in the beginning of a discourse). So we will assign the focus solely on the basis of semantic categories of the verb. Often, the discourse focus is the thematic role of the verb, and the actor focus the agent. Here is an example:

(15) A hand lays an envelope full of money on the table in front of Butch. Butch picks it up.

The first sentence introduces four discourse referents that are candidates for discourse and actor focus. The hand is proposed as actor focus, but not retained, as there is no reference to it in the follow-up sentence. The discourse focus is the envelope with money, because that's what Butch picks up. (Note that world knowledge would not exclude that the pronoun 'it' refers to the table. But this is not the preferred antecedent, as predicted by the focusing theory.) Copular sentences form an exception to this rule, they take their subject as discourse focus.

Translated into Prolog this gives us:

```
expectedDiscourseFocus(drs(Dom,Conds),Focus):-
   member(Focus,Dom),
   role(theme,Focus,Conds).
```

The thematic roles are determined by traversing through the DRS looking for a condition that matches the properties for a discourse referent for being in agent or theme position. Here are the clauses of `role/3` that deal with Focus-DRSs and DRSs:

```
role(Role,X,drs(_,_,_,Conds)):-
   role(Role,X,Conds).

role(Role,X,drs(_,Conds)):-
   role(Role,X,Conds).

role(Role,X,merge(B1,B2)):-
```

```
      role(Role,X,B1);
      role(Role,X,B2).

   role(Role,X,alfa(_,_,_,B)):-
      role(Role,X,B).
```

And here are the conditions that deal with the DRS-conditions:

```
   role(Role,X,[~ B|_]):-
      role(Role,X,B).

   role(Role,X,[B1 > B2|_]):-
      role(Role,X,B1);
      role(Role,X,B2).

   role(Role,X,[B1 v B2|_]):-
      role(Role,X,B1);
      role(Role,X,B2).

   role(Role,X,[_|Conds]):-
      role(Role,X,Conds).

   role(agent,X,[C|_]):-
      agent(X,C).

   role(theme,X,[C|_]):-
      theme(X,C).
```

We use a simple rule of thumb for determining the agent role of a discourse referent: if it is the subject of a transitive verb. Therefore, in our Prolog code that implements this rule, we appeal to our lexicon:

```
   agent(Focus,Cond):-
      compose(Cond,Sym,[Subject,_]),
      Focus==Subject,
      lexicon(tv,Sym,_,inf).
```

Similarly, we implement thematic positions, as either being the subject of an intransitive verb, the object of a transitive verb, or the subject of a copular sentence. This is coded as follows:

```
theme(Focus,Cond):-
   compose(Cond,Sym,[Subject]),
   Focus==Subject,
   lexicon(iv,Sym,_,inf).

theme(Focus,Cond):-
   compose(Cond,Sym,[_,Object]),
   Focus==Object,
   lexicon(tv,Sym,_,inf).

theme(Focus,Cond):-
   compose(Cond,'=',[Subject,Object]),
   var(Object),
   Focus==Subject.
```

Exercise 3.4.1 Extend `theme/2` by adding clauses for ditransitive verbs.

Step 2: Pronoun Resolution using Focus

The focus information is used to resolve pronouns as formulated by the following two rules:

If the pronoun appears in a sentence thematic relation other than agent, choose the discourse focus as antecedent.

If the pronoun appears in agent position, choose the actor focus as antecedent.

These rules apply only if other constraints do not rule out these choices. Here is the implementation of the first rule (pronoun appearing in agent-position):

```
resolveDrs([alfa(X,Type,Gender,B1)|A1]-A2,R1-[Type:X|R2]):-
   role(agent,X,B1), !,
   (
    currentActorFocus(A1,AF),
    potentialAntecedent(A1,X,Gender),
    X == AF,
    properBinding(Type,X,B1)
   ;
    currentDiscourseFocus(A1,DF),
    potentialAntecedent(A1,X,Gender),
    X == DF,
```

```
     properBinding(Type,X,B1)
    ;
     potentialAntecedent(A1,X,Gender),
     properBinding(Type,X,B1)
    ),
    resolveDrs([B1|A1]-A2,R1-R2).
```

The order of the three disjuncts mirror the preferences in a clear way: first try to actor focus, then the discourse focus, and then any other discourse referent.

The implementation of the second rule (the pronoun is not in agent position) is set up in a similar way, but with the preference order changed:

```
resolveDrs([alfa(X,Type,Gender,B1)|A1]-A2,R1-[Type:X|R2]):-
   (
    currentDiscourseFocus(A1,DF),
    potentialAntecedent(A1,X,Gender),
    X == DF,
    properBinding(Type,X,B1)
   ;
    currentActorFocus(A1,AF),
    potentialAntecedent(A1,X,Gender),
    X == AF,
    properBinding(Type,X,B1)
   ;
    potentialAntecedent(A1,X,Gender),
    properBinding(Type,X,B1)
   ),
   resolveDrs([B1|A1]-A2,R1-R2).
```

The predicates that define the current actor and discourse focus are specified as follows:

```
currentDiscourseFocus(DRSs,Focus):-
   member(drs(_,_,[Focus|_],_),DRSs), !.

currentActorFocus(DRSs,Focus):-
   member(drs(_,[Focus|_],_,_),DRSs), !.
```

What these predicates do is find a DRS on the stack that have a non-empty focus. The cut prevents backtracking after succeeding in finding a focus.

Step 3: Updating the Focus

After each analysis of a sentence, the focus information needs to be updated. This applies also to sentence-internally, for example in conditional sentences. For updating the focus we will use the predicates `expectedActorFocus` and `expectedDiscourseFocus` for an initial sentence of a discourse; otherwise we use the information of how the pronouns in the current sentence are resolved. Recall from Step 2 that we represent the resolution information as a list with terms `Alfa-Type:Antecedent`.

The body of the focusing update function can be designed as follows. First update the actor focus, and then the discourse focus.

```
updateFocus(Anaphora,DRSs1,Updated):-
   updateActorFocus(DRSs1,DRSs2),
   updateDiscourseFocus(Anaphora,DRSs2,Updated).
```

The actor focus is updated by taking the expected actor focus (if there is one).

```
updateActorFocus([drs(D,AF,DF,C)|A],Updated):-
   expectedActorFocus(drs(D,C),Focus), !,
   addActorFocus(Focus,[drs(D,AF,DF,C)|A],Updated).

updateActorFocus(DRSs,DRSs).

addActorFocus(Focus,DRSs,Updated):-
   (
       currentActorFocus(DRSs,Current),
       Current==Focus, !,
       Updated=DRSs
   ;
       DRSs=[drs(D,AF,DF,C)|A],
       Updated=[drs(D,[Focus|AF],DF,C)|A]
   ).
```

This is how the discourse focus gets updated. If there is a previous discourse focus and there is an anaphor binding it, then the focus stays like it was. If there is no previous discourse focus, then the current focus is the expected discourse focus. In all other cases, we leave it as it is.

```
updateDiscourseFocus(Anaphora,DRSs,Updated):-
   currentDiscourseFocus(DRSs,Current),
   member(_:A,Anaphora), A==Current, !,
```

```
   Updated=DRSs.

updateDiscourseFocus(_Anaphora,[drs(D,AF,DF,C)|A],Updated):-
   expectedDiscourseFocus(drs(D,C),Focus), !,
   addDiscourseFocus(Focus,[drs(D,AF,DF,C)|A],Updated).

updateDiscourseFocus(_,DRSs,DRSs).

addDiscourseFocus(Focus,DRSs,Updated):-
   (
       currentDiscourseFocus(DRSs,Current),
       Current==Focus, !,
       Updated=DRSs
   ;
       DRSs=[drs(D,AF,DF,C)|A],
       Updated=[drs(D,AF,[Focus|DF],C)|A]
   ).
```

Here is how the focus updating predicates are integrated into our resolution algorithm (for a merge of DRSs, for Focus-DRSs, and ordinary DRSs):

```
resolveDrs([merge(B1,B2)|A1]-[drs(D,AF,DF,C)|A5],R1-R3):-
   resolveDrs([B1|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3),
   resolveDrs([B2|A3]-A4,R2-R3),
   updateFocus(R3,A4,[drs(D1,AF1,DF1,C1),drs(D2,AF2,DF2,C2)|A5]),
   append(D1,D2,D),
   append(AF1,AF2,AF),
   append(DF1,DF2,DF),
   append(C1,C2,C).

resolveDrs([drs(D,AF,DF,C)|A1]-A3,R1-R2):-
   resolveConds(C,[drs(D,AF,DF,[])|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3).

resolveDrs([drs(D,C)|A1]-A3,R1-R2):-
   resolveConds(C,[drs(D,[],[],[])|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3).
```

Resolving DRS-conditions is exactly like that in the basic pronoun resolution algorithm, except for conditionals. Here we also need to introduce a focus update. Because in a ‘if

S1 then S2' sentence, pronouns appearing in the second sentence (S2) will be resolved also depending on the information of sentence (S1).

```
resolveConds([B1 > B2|Conds],A1-A5,R1-R4):-
   resolveDrs([B1|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3),
   resolveDrs([B2|A3]-[B4,B3,drs(D,AF,DF,C)|A4],R2-R3),
   resolveConds(Conds,[drs(D,AF,DF,[B3 > B4|C])|A4]-A5,R3-R4).
```

Finally, we provide a driver predicate that makes it easy to experiment with our focusing algorithm. After each sentence it analyses, it prints the DRS, after which the user can extend the discourse with a new sentence.

```
parse:-
   parse(drs([],[])).

parse(Old):-
   readLine(Discourse),
   d(SDrs,Discourse,[]),
   betaConvert(SDrs,Drs),
   resolveDrs([merge(Old,Drs)]-A,[]-Anaphora),
   updateFocus(Anaphora,A,[New]),
   printRepresentation(New), !,
   parse(New).
```

Exercise 3.4.2 [easy] Try the program with the following example: 'Butch picks up a hammer. He discards it. He picks up a chainsaw, and he likes it.' Explain the focus shifts.

Exercise 3.4.3 [hard] Write a driver that generates all readings, ordered on preference. Assign scores to the readings, according to the focus rules.

Software Summary of Chapter 3

`mainPronounsDRT.pl` The basic implementation of pronoun resoluition for DRSs. (page 186)

`bindingDRT.pl` Predicates that check binding of pronouns in DRT. (page 188)

`mainFocusDRT.pl` The focus algorithm. (page 190)

Notes

The binding behavior of pronouns (coreference within a sentence) is discussed in the linguistic literature in great detail, starting with Chomsky's Binding Theory (Chomsky 1988). A rich source of information on this topic is provided in Reinhart's work (Reinhart 1983). For more inspiration on (especially reflexive) pronouns the reader is encouraged to check out the first two chapters of Fauconnier's book on Mental Spaces (Fauconnier 1985).

Pronoun resolution procedures that go beyond the sentence level are widely available in the field of Artificial Intelligence. Pioneers in defining heuristics for pronoun resolution are Terry 'the blocks world' Winograd (Winograd 1972) and Eugene Charniak. Charniak's dissertation (Charniak 1972) includes a proposal for pronoun resolution included syntactic (gender, number) as well as semantic properties. He was the first that made aware the need of common-sense knowledge reasoning to determine antecedents of pronouns. See also (Charniak and Wilks 1976; Charniak and McDermott 1985).

Hobbs presents a syntactic and semantic approach to pronoun resolution (Hobbs 1986). The syntactic approach offers what he called the 'naive' algorithm, which traverse the parse tree to determine potential antecedents, thereby using constraints known from transformational grammar to deal with reflexive pronouns and cataphora. Within the semantic approach he proposes different strategies using inference to determine antecedents. This paper also give a modest but useful overview of the problems that appear with pronoun resolution.

Wada and Asher included in their DRS construction implementation a module that takes care of pronoun resolution (Wada and Asher 1986). Our basic implementation borrows a lot from this work as it: (1) marks pronouns in the DRS by a special condition; (2) associates discourse referents with gender and number information stored in a tree structure that parallels the subordination relation of a DRS; (3) resolves this tree by finding the closest antecedent that matches gender and number; and (4) uses additional constraints (e.g., the reflexive/non-reflexive distinction).

The focusing algorithm as presented here is originally due to Candy Sidner (Sidner 1986). Some of the ideas in this paper—that appeared in the late seventies—are strikingly similar to the principles underlying DRT: using an intermediate level of representation, where what Sidner calls *specifiers* play a similar role as the discourse referents in DRSs. Sidner motivates the rules in her focusing algorithm by a number of examples, and also discusses the role of plural pronouns in focusing, and that of the demonstrative pronouns 'this' and 'that'. The implementation that we gave here is a slight variant of Sidner's model. We deal with sentence-internal discourse structure (which Sidner doesn't), but, on the other hand, simplified the set of update rules.

The focus model is a precurser of what is called the *centering* approach to pronoun resolution. Centering elaborates on the basic idea of focusing. This approach distinguishes *forward looking centers* (i.e., in the DRT-terminology, a set of available discourse referents, ordered on grammatical relations or other criteria) and the *backward looking center* (the highest ranked forward looking center of the previous utterance, to which a pronoun or other anaphoric expressions in the current utterance refers to). Useful pointers are (Grosz, Joshi, and Weinstein 1995) and (Kehler 1997). Beaver (unpublished) proposes a model that integrates centering within a formal update semantics framework.

Focus information, as we plainly added it to the evolving DRS of a text, adds another dimension to the structure of the discourse. A more elaborate model of discourse structure can be found in the classical paper of Grosz and Sidner (Grosz and Sidner 1986), or a formalized 'boxed' version, in Asher's Segmented DRT (Asher 1993).

Pronouns are a motley crew of context-sensitive expressions. We've only dealt with third person singular pronouns, and didn't consider other uses of anaphoric expressions, such as impersonal use of pronouns, plural anaphora, or pronouns that refer to abstract entities such as eventualities or propositions. A recommended DRT-based introduction to plural anaphora is Chapter 4 of (Kamp and Reyle 1993). The undoubtedly best overview of propositional and event-type anaphora, as well as the related phenomenon known as verb phrase ellipsis, is (Asher 1993), also presented in DRT.

Chapter 4

Presupposition Resolution

In this chapter we are going to deal with a stubborn obstacle to the computational analysis of discourse: presuppositions. Roughly speaking, presuppositions are pieces of information that are taken for granted in a context. In this chapter we will learn what the typical problems associated with presuppositions are, and how to deal with them in Discourse Representation Theory.

Presupposition is a huge topic. It has been discussed in detail by both the philosophical and linguistic communities, and we won't be able to cover the wide variety of analyses such discussions have inspired. In fact, we shall concentrate exclusively on the DRT based approach of Rob van der Sandt. There are a number of reasons for this choice. Empirically, it is one of the most successful approaches to the problems posed by presuppositions (for example, the *projection problem* introduced below). Conceptually the theory is also extremely natural: it is simple, allows the important notion of *accommodation* to be modeled very directly, and links presupposition and anaphora in a suggestive way. All in all, Van der Sandt's account both draws on and illuminates the intuitions about discourse processing that DRT embodies. Finally, the approach is interesting from a computational perspective: it demands an extension of our earlier implementation of pronoun resolution, and access to inference methods.

4.1 Introducing Presuppositions

In the previous chapter we worked with elements of natural language that were obviously context dependent, namely pronouns. Pronoun interpretation can only be determined with help of the previous discourse, and we gave a DRT-based account of anaphora resolution. But there is another natural language phenomenon, possibly related to pronoun use, that behaves in a similar way. Consider the following example sentences, uttered out of the blue:

(1) The couple that won the dance contest was pleased.

(2) Jody loves her husband.

(3) Vincent regrets that Mia is married.

There is something interesting about these examples: they force us to take something for granted if we want to accept them as natural contributions to a discourse or conversation. For (1), we need to suppose that there actually is a couple that won the dance contest, for (2) that Jody has a husband, and for (3) that Mia is married.

Moreover, given a context containing contrary information, these sentences do not make sense at all. For example, the following sequences are unacceptable:

(4) Jody has no husband. ? Jody loves her husband.

(5) Mia is not married. ? Vincent regrets that Mia is married.

Furthermore, whatever we are dealing with here, it behaves completely differently from ordinary entailment. Not only does 'Jody loves her husband' take it for granted that Jody has a husband, but so does its negation 'Jody does not love her husband'; note that

(6) Jody has no husband. ? Jody does not love her husband.

is just as bad as (4). Whatever this phenomenon is, it seems to follow its own set of laws.

In fact we are dealing with *presupposition*, and this is a very common natural language phenomenon that any analysis of discourse has to cope with. The sentence 'Jody loves her husband' *presupposes* that Jody is married, as does 'Jody does not love her husband'. Any context in which Jody is not married is inappropriate for accepting either the sentence 'Jody loves her husband' or its negation.

Is there a systematic way to compute the presuppositions of a sentence? To answer this question we need to investigate how presuppositions come to life. It turns out that, in most cases, they are *lexically* triggered. (At least this is the case for English, certain prosodic or syntactic structures can also introduce presuppositions, though we won't deal with such triggers in this book.) In (1), the definite article 'the' is the source of the presupposition (the reader might verify that by changing 'the' into 'a'), in (2) it is the possessive 'her', and in (3) the verb 'regret'. Such lexical items are called *presupposition triggers*: once they are used, they introduce a presupposition, thereby putting extra constraints on the context.

At first glance this problem seems easy to deal with. Surely it is just a matter of going through our lexicon, marking all presupposition triggers and the presupposition they induce, and then when analyzing a sentence, checking if all presuppositions triggered by the complete sentence are fulfilled by the discourse. And in fact, this *is* a good way of thinking of it, and we will handle presupposition triggers along these lines in this chapter. However, matters will not always be so simple. There are three issues we need to pay attention to: the *binding* problem, the *projection problem*, and *accommodation*. All three issues are well known in the (vast) literature on presuppositions.

Let's first turn to the *binding* problem. An example like

> A boxer nearly escaped from his apartment.

should clarify what is at stake here. The trigger 'his' induces the presupposition that someone has an apartment. But it does not presuppose that just anyone has an apartment, nor that some boxer or other owns an apartment. No, it is *the boxer we are actually talking about* who has an apartment. That is, the existentially quantified NP ties together two types of information—ordinary factual information, and presuppositional information. As these two types of information obey different laws (recall the way presuppositions survive negation) it is no trivial matter to tie them together, and many otherwise interesting accounts of presupposition have been shipwrecked on precisely this rock. However, as we shall soon see, Van der Sandt's DRT-based account handles it rather elegantly. Because DRT was designed right from the start to deal with *integrating* information, the binding problem simply won't be an issue.

The *projection problem* has to do with complex sentences. We have already seen that

> Mia's husband is out of town.

is a sentence presupposing that Mia has a husband. But if we use this sentence to build the more complex

> If Mia has a husband, then her husband is out of town.

then the result does *not* presuppose that she has a husband. Neither does

> If Mia is married, then her husband is out of town.

although

> If Mia dates Vincent, then her husband is out of town.

clearly does. The moral is clear. In complex sentences we need to be careful when dealing with presupposition triggers, as sometimes subparts of complex sentence carry presuppositions which are "neutralized" in the main sentence. A natural solution to this problem would seem to be to take linguistic context into account, and this is also what we are going to do, again within the framework of DRT.

Finally a few words on *accommodation*. This can be thought of as a way of obtaining a robust and realistic treatment of presupposition. Analyzing

Vincent informed his boss.

yields the elementary presupposition that Vincent has a boss. If the discourse built up so far is incompatible with Vincent having a boss (maybe he is a free-lancer), then we could just say: "Hey, what are you telling me, Vincent has no boss", and not accept this contribution to the discourse. But what if we don't have a clue whether Vincent has a boss or not? There are two obvious ways to proceed here: the simplistic way is simply to refuse to accept the utterance because its accompanying presuppositions are not supported by the context. This might seem adequate, but it isn't what we would typically do in such a situation—instead we would try to add the presupposition to the context. The addition process is called *accommodation*, and it is best viewed as a repair strategy. We are trying to build up a picture of a particular situation, but it turns out that (somehow or other) we seem have missed some piece of information. OK—let's try and incorporate this information the best we can and carry on. This is an inherently robust and realistic way of dealing with presuppositions and it is the option we shall explore in this book.

4.2 Dealing with Presupposition in DRT

We are now going to show how to deal with these three problems in DRT using a method due to Rob van der Sandt. In his view, presuppositions are essentially extremely *rich* pronouns. Like ordinary pronouns, they make use of the notion of accessibility in the way we are familiar with. But presuppositions are rich in a way that pronouns aren't: they have descriptive content. Thus presuppositions can't simply introduce discourse referents, so what do they do? Here's Van der Sandt's answer: *presuppositions introduce new DRSs*.

This is a strikingly simple and promising answer. DRSs are pictures of the evolving discourse; what better way to incorporate presuppositional information than by adding a new picture? The key word here is *incorporate*. Simply by placing the DRS yielded by a presupposition into a larger DRS we locate it in a web of discourse referents regulated by accessibility constraints. That is, by proceeding in this way we automatically *contextualize* presuppositional information. It seems plausible that this is a good way of coping with the binding, projection, and accommodation problems, and this in fact is the way things turn out. Let's turn to the details.

To begin with, we have to carry out two tasks:

1. select presupposition triggers in the lexicon; and
2. indicate what they presuppose.

Task 1 appeals to our empirical knowledge of presupposition. For our fragment of English, we classify the definite article, possessive constructions, and proper names as presupposition triggers. For task 2 we extend the core DRS language with an α operator. The α operator takes two normal DRSs as arguments forming a new special kind of DRS; it is simply a marker (or if you prefer, a *semantic feature*) that indicates that one DRS is presupposed information for the other. So, we will write $\mathrm{B}_1\ \alpha_i\ \mathrm{B}_2$ to say that B_1 is presupposed information for B_2, or put differently, B_2 presupposed B_1. We use the index i to express that it is the principal discourse referent that is anaphoric, as there could be more discourse referents in the presupposed DRS (this is the discourse referent for which we will try to find an antecedent). We will call B_1 the α-DRS (this is the "anaphoric" DRS for which we have to resolve), B_2 is a DRS that remains at its place after resolution.

Exercise 4.2.1 Find natural language examples where there are more discourse referents in the α-DRS (the DRS that contains presupposed information).

Once we've dealt with the lexicon, what's the next step? Well, using our familiar DRS construction technology, we build a DRS for the sentence. If the sentence contains presupposition triggers, the DRS we output will generally contain α-DRSs. Here's the DRS we obtain for 'The woman collapses':

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \hline\end{array}\ \alpha_x\ \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{x}) \\ \hline\end{array}$$

The best way to view this DRS is as an ordinary DRS, but an ordinary DRS marked as being unresolved with respect to presuppositions. (Another way to look at this is to say that it is a kind of underspecified DRS; we'll see why later.) What are we going to do with this DRS?

First we merge this new DRS with the DRS that represents the discourse so far; this merging process takes place while the presuppositions are still unresolved. For example, suppose that the previous utterance was 'A woman snorts'. Then after merging we obtain:

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{woman}(\mathrm{y}) \\ \textsc{snort}(\mathrm{y}) \\ \hline\end{array} \oplus \left(\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \hline\end{array}\ \alpha_x\ \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{x}) \\ \hline\end{array}\right)$$

Only after merging of the previous DRS with the new DRS we attempt to resolve the presuppositions. We recursively travel through this new merged DRS and, for each α-marked DRS we encounter, we try to find a suitable 'anchor' to resolve to. That is, *we try to match the context of the α-DRS with that of superordinated DRSs.* Intuitively this is a natural thing to do; after all, presupposed information is supposed to be available in the previous context.

Let's see how this works. In our example, we only have one presupposition trigger with α-DRS:

$$\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \hline\end{array}$$

To resolve α-DRS, there is only one superordinated DRS (with the candidate antecedent discourse referent y) for resolution.

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{woman}(\mathrm{y}) \\ \textsc{snort}(\mathrm{y}) \\ \hline\end{array}$$

First we identify the discourse referents x and y by adding the condition x=y to the α-DRS:

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{woman}(\mathrm{y}) \\ \textsc{snort}(\mathrm{y}) \\ \hline\end{array} \oplus \left(\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \mathrm{x=y} \\ \hline\end{array} \;\alpha_x\; \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{x}) \\ \hline\end{array} \right)$$

Then we move the information of the α-DRSs to the superordinated DRS:

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \textsc{woman}(\mathrm{y}) \\ \textsc{snort}(\mathrm{y}) \\ \\ \hline\end{array} \oplus \left(\begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \mathrm{x=y} \\ \hline\end{array} \;\alpha_x\; \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{x}) \\ \hline\end{array} \right)$$

Finally we replace the operator α_x by a merge-instruction and arrive at:

$$\begin{array}{|l|}\hline \mathrm{y\ x} \\ \hline \textsc{woman}(\mathrm{y}) \\ \textsc{snort}(\mathrm{y}) \\ \textsc{woman}(\mathrm{x}) \\ \mathrm{x=y} \\ \hline\end{array} \oplus \left(\begin{array}{|l|}\hline \\ \hline \\ \hline\end{array} \oplus \begin{array}{|l|}\hline \\ \hline \textsc{collapse}(\mathrm{x}) \\ \hline\end{array} \right)$$

This is a DRS that contains two instructions for merging. We can reduced it to:

$$\begin{array}{|l|}\hline \text{y x} \\ \hline \textsc{woman}(\text{y}) \\ \textsc{snort}(\text{y}) \\ \textsc{collapse}(\text{y}) \\ \textsc{woman}(\text{x}) \\ \text{x=y} \\ \hline \end{array}$$

But note that we had have could carry out the identification of x and y explicitly (by unification) and compute the following, equivalent DRS:

$$\begin{array}{|l|}\hline \text{y} \\ \hline \textsc{woman}(\text{y}) \\ \textsc{snort}(\text{y}) \\ \textsc{collapse}(\text{y}) \\ \hline \end{array}$$

In the other examples we will discuss, and also in our Prolog-implementation of presupposition resolution, we will perform this unification operation instead of adding an equality condition to the DRS, as this will lead to much simpler DRSs.

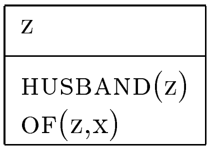
In short, we have successfully dealt with the presupposition induced by the definite article 'the' by identifying the discourse referent x it introduced with the woman-denoting discourse referent in the preceding context.

That's the basic idea, but things don't always go this smoothly. Sometimes we can't find the presupposed information in the preceding context. (Maybe, we missed a bit of a conversation; and anyway, people typically make different assumptions about what the assumed context actually is.) To deal with such cases we make use of *accommodation*: if we can't link our elementary presuppositions to a suitable element in the context, we don't give up. Instead we simply add the required background information.

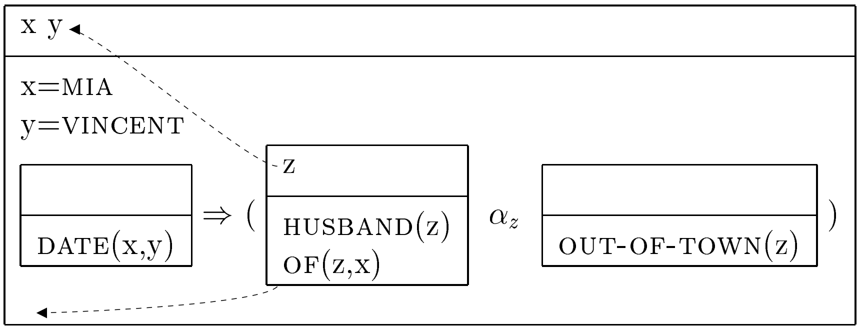
Here's an example. Consider the sentence 'If Mia dates Vincent, then her husband is out of town'. Concentrating only on the trigger 'her husband', we get:

$$\begin{array}{|l|}\hline \text{x y} \\ \hline \text{x=}\textsc{mia} \\ \text{y=}\textsc{vincent} \\ \begin{array}{|l|}\hline \\ \hline \textsc{date}(\text{x,y}) \\ \hline \end{array} \Rightarrow \left(\begin{array}{|l|}\hline \text{z} \\ \hline \textsc{husband}(\text{z}) \\ \textsc{of}(\text{z,x}) \\ \hline \end{array} \;\alpha_z\; \begin{array}{|l|}\hline \\ \hline \textsc{out-of-town}(\text{z}) \\ \hline \end{array} \right) \\ \\ \hline \end{array}$$

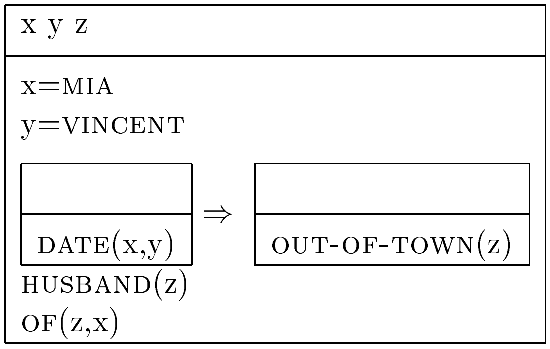
Assuming this is the first DRS we have to process (that is, that the DRS built up so far is still empty), there is no candidate DRS for matching the presupposed information that Mia has a husband, which is coded by the following DRS:



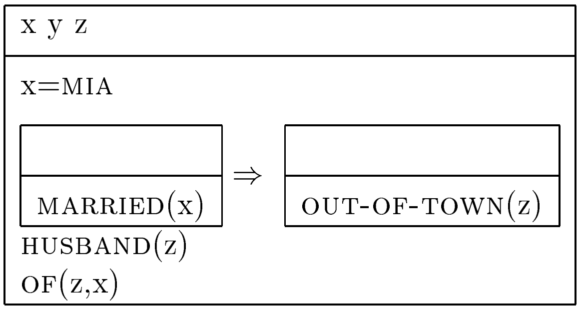
In such cases our robust way of dealing with discourse comes into action: we *accommodate* the presupposed information. Accommodation is similar to resolution. The big difference is that we *don't* identify discourse referents of the α-DRS with a superordinated DRS. For the rest, accommodation is exactly the same as resolution, and we move the α-DRS to a superordinated DRS:



As we moved to the presupposed information to the outermost DRS, we will call this *global* accommodation. After replacing the α-operator by $\oplus$ and carrying out the merges we get the following, final, DRS:



So far so good: our "try to match the presupposed information, and if that fails, accommodate it at the global DRS" strategy has survived its first test. But we were lucky with our example; consider instead 'If Mia is married, then her husband is out of town'. Does this presuppose that Mia has a husband? No, it doesn't—but applying our global accommodation strategy yields:



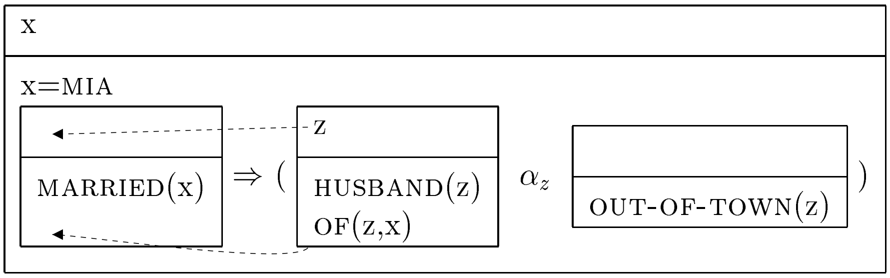
The content of this DRS is paraphrased as

(7) Mia has a husband. If she is married, then he is out of town.

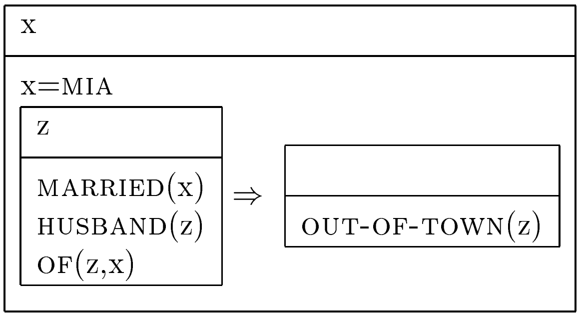
which is a spectacularly wrong result, as it asserts that Mia is married.

But wait on—who said we needed to perform accommodation in the outermost DRS? For sure, all discourse referents in the universe of outermost DRS are accessible—but one of the most pleasant aspects of DRT is that it shows how pictures can be nested one inside the other, and tells us which discourse referents are available at arbitrary levels of nesting. Why not pursue a more flexible attitude towards accommodation and allow accommodation at other DRS levels as well? There is certainly no technical barrier to doing this, and arguably it is the natural option to explore: bluntly insisting that all information has to be added globally doesn't seem to reflect the myriad possibilities offered by natural language.

Let's see how this more flexible strategy can help us with our previous example. In fact it licenses accommodation on a non-global, intermediate, level:

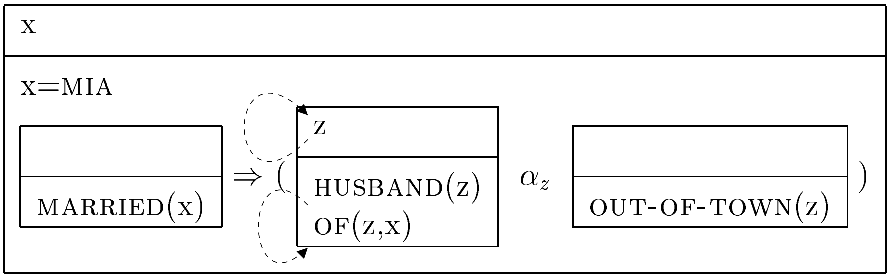


This gives us (again, after replacing the α-operator by a merge) the following reading:

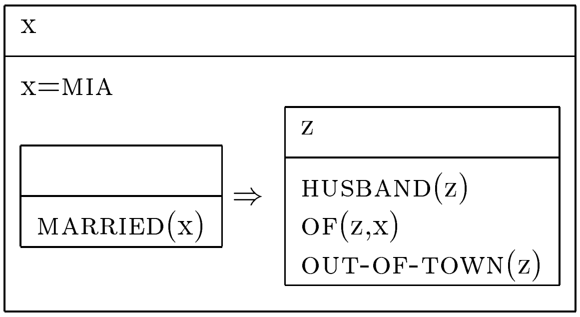


Here the presupposed information is accommodated in the left-hand-side DRS of the conditional. This is the reading we would like to get: it doesn't demand the existence of a husband for Mia.

Indeed, in this example, there is another possibility for accommodation. It is called local accommodation, and sketched as follows:



Local accommodation is actually not performing any move operation. The presupposed information just stays where it originated. After replacing α_z by a merge we get:



That's the basic idea—but there are two further points that need to be mentioned. First, it's clear that we can't perform accommodation arbitrarily; we've already seen what happens when we try to inappropriately accommodate at the global level. Rather, just like ordinary pronoun resolution, accommodation is subject to a variety of acceptability constraints. We will briefly discuss the required constraints shortly; Chapter 11 is largely devoted to implementing them. Second, accommodation is a non-deterministic process.

That is, we are free to accommodate where we like, so long as we don't violate an acceptability constraint. Note that this has linguistic consequences: we are predicting that presupposition is an additional source of ambiguity. Incidentally, this is why we claimed earlier that α-DRSs could be viewed as underspecified representations: essentially α offers a compact encoding of accommodation possibilities.

Summing up, both here and in Chapter 11, we are working with the following non-deterministic algorithm:

1. Generate a DRS for the input sentence with the tools of Chapter 2, with all elementary presuppositions given as an α-DRS.
2. Merge this DRS with the DRS of the discourse so far processed.
3. Traverse the DRS, and on encountering an α-DRS try to
   (a) link the presupposed information to an accessible antecedent (partial match);
   (b) if that fails, accommodate the information to a superordinated level of discourse.
4. Remove those DRS from the set of potential readings that violate the acceptability constraints.

What sort of constraints do we appeal to at Step 4? Just like pronoun resolution, accommodation is subject to two kinds of constraints. The first is essentially syntactic, or formal (that is, like the accessibility constraint on pronoun resolution it simply imposes a well-formedness condition on the resulting DRSs—only simple syntactic checking is required to check whether it is fulfilled). This constraint is called the *free variable check*. The following example shows what this involves; it also shows the kinds of ambiguities predicted by the Van der Sandtian approach to presupposition.

Consider the sentence 'Every woman likes her husband'. There are three candidate accommodation strategies: local, intermediate, and global. Here are the DRS these options yield:

local

$$\begin{array}{|c|}\hline \quad \\ \hline \begin{array}{|l|}\hline x \\ \hline \text{WOMAN}(x) \\ \hline\end{array} \Rightarrow \begin{array}{|l|}\hline z \\ \hline \text{HUSBAND}(z) \\ \text{OF}(z,x) \\ \text{LIKE}(x,z) \\ \hline\end{array} \\ \hline\end{array}$$

intermediate

$$\begin{array}{|c|}\hline \quad \\ \hline \begin{array}{|l|}\hline x\ z \\ \hline \text{WOMAN}(x) \\ \text{HUSBAND}(z) \\ \text{OF}(z,x) \\ \hline\end{array} \Rightarrow \begin{array}{|l|}\hline \quad \\ \hline \text{LIKE}(x,z) \\ \hline\end{array} \\ \hline\end{array}$$

global

$$\begin{array}{|l|}\hline z \\ \hline \text{HUSBAND}(z) \\ \text{OF}(z,x) \\ \begin{array}{|l|}\hline x \\ \hline \text{WOMAN}(x) \\ \hline\end{array} \Rightarrow \begin{array}{|l|}\hline z \\ \hline \text{LIKE}(x,z) \\ \hline\end{array} \\ \hline\end{array}$$

In fact, both the local and the intermediate are acceptable: neither violates an acceptability constraint.

Exercise 4.2.2 Spell out the readings given by local and intermediate accommodation.

But now consider the DRS obtained by global accommodation. Clearly it is rather strange; one of the conditions (namely, OF(z,x)) ascribes a property to a discourse referent (namely, x) that does not occur in the universe. So to speak, x is a free discourse referent, and in fact if we used the translation function *fo* to compile this DRS into a first-order formula, we would obtain not a sentence but a formula in which the variable x occurred free. The free variable constraint simply rules out such DRSs.

Exercise 4.2.3 Although we can appeal to *fo* to define what we mean by a 'free discourse referent', we certainly don't need to do this. Give a direct definition of this concept.

But what are the other acceptability constraints? Here matters get a lot more interesting (and a lot more difficult) for these constraints require us to perform *inference*, and a great deal of inference at that. Let's briefly note the two most obvious constraints we should observe.

First, we need to obey the *consistency* constraint; obviously it would be foolish to accommodate information in such a way that the resulting DRS was inconsistent. Note, however, that we can't test for inconsistency simply by inspecting the DRS; we have to do some real work. Second, we should obey the *informativity* constraint. It would be redundant to accommodate information in such a way that the resulting DRS actually followed from the DRS we started with. That is, we should only carry out accommodation if it results in something genuinely new. Once again, however, it requires inference to determine whether or not this constraint is obeyed. Actually, Van der Sandt requires *local* versions of consistency and informativity as well, and like the constraints just mentioned, these require inference to determine whether or not they are fulfilled. We shall discuss these constraints further in Chapter 11.

Here's our plan. In the remainder of this chapter we implement the basic architecture described above. That is, we shall learn how to generate candidate DRSs (as we shall see, this part of the process is closely related to the basic pronoun resolution method implemented in the previous chapter) but we *won't* perform the filtering demanded by the acceptability check. In Chapter 11 we shall add the required checks. We will do so by using `drs2fol`, our DRS to first-order logic compiler, and then exploiting standard first-order inference techniques.

4.3 Presupposition Resolution in Prolog

Implementing accommodation has two main steps. First we must take care of the lexicon: we need to say exactly what sort of presuppositions the various entries trigger. Secondly we need to think about how the actual accommodation process is to be performed.

Extending Lexical Macros

All presupposition triggers in our fragment of English introduce an α-condition. We shall use a prolog term

```
alfa(X,Type,ADrs,Drs),
```

where `ADrs` is the α-DRSs, and `Drs` its scope. (So, we altered the representations for pronouns of the previous chapter into one where all we did is replacing the gender information by the α-DRS. In fact, we will use this notation for pronouns as well.) Here is the semantic macro for the determiner 'the':

```
detSem(def,lambda(P,lambda(Q,alfa(X,def,merge(drs([X],[]),P@X),Q@X)))).
```

That is, the scope of the presupposition operator α is restricted to the 'noun part' of the definite article. Compare this with the rule for the determiner 'a':

```
detSem(indef,lambda(P,lambda(Q,merge(merge(drs([X],[]),P@X),Q@X)))).
```

Proper names are also viewed as presupposition triggers. This give us an important practical advantage: we don't need an additional device that moves the discourse referents to the outermost DRS, thereby allowing accessibility for later pronouns: global accommodation deals with this automatically. So here is the revised macro for proper names:

```
pnSem(Sym,Gender,lambda(P,alfa(X,name,drs([X],[X=Sym,Cond]),P@X))):-
   compose(Cond,Gender,[X]).
```

Since linking is part of our presupposition resolution mechanism, pronouns are dealt with in the same spirit. So pronouns also add an α, with a DRS that only has a discourse referent and no conditions.

```
proSem(Gender,Type,lambda(P,alfa(X,Type,drs([X],[Cond]),P@X))):-
   compose(Cond,Gender,[X]).
```

Possessives like 'her X' and 'his X' are analyzed as 'the X of her' and 'the X of him'. Syntactically, they are determiners. Here is the semantic macro; note that this is a rather complex entry, for we have nested α's.

```
detSem(poss(Gender),lambda(P,lambda(Q,
       alfa(X,nonrefl,drs([X],[Cond]),
            alfa(Y,nonrefl,merge(drs([Y],[of(Y,X)]),P@Y),Q@X))))):-
   compose(Cond,Gender,[X]).
```

With one exception, these are the presupposition triggers we deal with in our grammar fragment of English. The remaining one is left as as an exercise for the reader.

Exercise 4.3.1 Fill in the the interface definition for the determiner '`another`'. (It is already in the lexicon). What kind of presupposition does it trigger?

Exercise 4.3.2 Adjectives like '`old`' and '`new`' are under one reading presupposition triggers. Spell out the presuppositions for these kind of triggers and specify their lexical entries.

Implementing Linking and Accommodation

The initial output of our parser is a DRS with α-DRSs just like our implementation for pronoun resolution. This will be the input to the predicate `projectDrs/1`, that performs the linking and accommodation operations and finally returns a normal DRS.

Notably, `projectDrs/1` works along the same DRT-principles as `resolveDrs/1` from the previous chapter. (The reader who doesn't know about this predicate, is advised to learn about from the previous chapter, before proceeding). As its closely related `resolveDrs/1`, `projectDrs/1` works with stacks of DRSs. The three clauses of `projectDrs/1` resemble those of `resolveDrs/1`:

```
projectDrs([merge(B1,B2)|A1]-[drs(D3,C3)|A3]):-
        projectDrs([B1|A1]-A2),
        projectDrs([B2|A2]-[drs(D1,C1),drs(D2,C2)|A3]),
        append(D1,D2,D3),
        append(C1,C2,C3).

projectDrs([alfa(X,Type,B1,B3)|A1]-A5):-
        projectDrs([B1|A1]-[B2|A2]),
        resolveAlfa(X,B2,A2-A3),
        projectDrs([B3|A3]-A4),
        addBindingInfo(X,Type,A4-A5).

projectDrs([drs(D1,C1)|A1]-A2):-
        projectConds(C1,[drs(D1,[])|A1]-A2).
```

But there is more to it. We want to modify the stack more drastically, for we want to allow accommodation. This comes to play in the clauses that deal with the DRS-conditions, for which we introduce the predicate `projectConds/2` (indeed, as you might have expected, this predicate mirrors `resolveConds/2` of the previous chapter).

Accommodation is allowed at subordinated levels of discourse, and at the global level. But not inside nested merges. So we have to be careful when to apply accommodation, and avoid spurious accommodation. To manage accommodation, we introduce a special object on the stack, coded as `pre(Pres)`, that is a place holder for an accommodation site. Here, `Pres` is a list of presuppositions (naturally coded as DRSs) that are accommodated at this site (at the beginning, this is the empty list). As we will see later, `resolveAlfa` will fill this slot.

The clauses of `projectConds/2` make clear where these accommodation sites are introduced. The clause for negation, for instance, puts an empty site `pre([])` on the stack, which is filled with locally accommodated presuppositions `pre(Pres)` after processing the negated box, which are then, finally, accommodated to the negated box itself by use of `accommodate/2`. (We will come to this predicate shortly.)

```
projectConds([~B1|Conds],A1-A3):-
   projectDrs([B1,pre([])|A1]-[B2,pre(Pres),drs(D,C)|A2]),
   accommodate(Pres,B2-B3),
   projectConds(Conds,[drs(D,[~B3|C])|A2]-A3).

projectConds([B1 > B2|Conds],A1-A4):-
   projectDrs([B1,pre([])|A1]-A2),
   projectDrs([B2,pre([])|A2]-[B4,pre(P2),B3,pre(P1),drs(D,C)|A3]),
   accommodate(P1,B3-B5),
   accommodate(P2,B4-B6),
   projectConds(Conds,[drs(D,[B5 > B6|C])|A3]-A4).

projectConds([B1 v B2|Conds],A1-A4):-
   projectDrs([B1,pre([])|A1]-[B3,pre(P1)|A2]),
   projectDrs([B2,pre([])|A2]-[B4,pre(P2),drs(D,C)|A3]),
   accommodate(P1,B3-B5),
   accommodate(P2,B4-B6),
   projectConds(Conds,[drs(D,[B5 v B6|C])|A3]-A4).

projectConds([Basic|Conds],[drs(D,C)|A1]-A2):-
   compose(Basic,_Symbol,Arguments),
   simpleTerms(Arguments),
   projectConds(Conds,[drs(D,[Basic|C])|A1]-A2).

projectConds([],A-A).
```

Gathering all accommodated presuppositions on accommodation sites is very useful indeed. It is a means to compute the presuppositions for each sub-DRS. After the gathering is done, the presuppositions are added to the actual DRS they belong to with the help of `accommodate/2`. There is nothing special to this predicate: it recursively cycles through the list of accommodated DRSs and adds them to the actual DRS, by straightforward use of `append/3`.

```
accommodate([],B-B).

accommodate([drs(D1,C1)|Presups],drs(D2,C2)-B):-
        append(D1,D2,D3),
        append(C1,C2,C3),
        accommodate(Presups,drs(D3,C3)-B).
```

This is how, basically, the implementation for presupposition projection is arranged. But we haven't explained how the actual resolving and accommodation takes place. It is the predicate `resolveAlfa/4` that carries out this task. (It is part of the clause of `projectDrs` that handles α-DRSs).

The first two clause of `resolveAlfa` deal with linking (partially matching) the α-DRS with one of the DRS on the stack. (Here we use `matchDrs/5`, which we explain in a minute.) By the recursive nature of this predicate, partial matching is allowed with any of the DRSs on the stack by use of backtracking.

```
resolveAlfa(X,AlfaDrs,[drs(D,C)|Others]-[New|Others]):-
       matchDrs(X,AlfaDrs,drs(D,C),New).

resolveAlfa(X,AlfaDrs,[AnteDrs|Others]-[AnteDrs|NewOthers]):-
       resolveAlfa(X,AlfaDrs,Others-NewOthers).
```

But not only linking is possible. Accommodation is allowed on the sites on the stack marked by the term `pre/1`. This is straightforward too, as the argument of `pre/1` is a list of DRSs, and we can use the list operator and simply add the α-DRS to the accommodation site. Again, with the recursive set-up of this predicate, all possible accommodation sites are given a chance under backtracking.

```
resolveAlfa(_,AlfaDrs,[pre(A)|Others]-[pre([AlfaDrs|A])|Others]).

resolveAlfa(X,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others]):-
       select(Drs,A1,A2),
       matchDrs(X,Alfa,Drs,New).
```

Finally, we have to code the partial match of DRSs. The predicate `matchDrs/5` matches two DRSs yielding a new one, by unifying the anaphoric referent `X` with one of the discourse referents of the antecedent DRS (so, it will fail if the domain of this DRS is empty), and then take the union of the discourse referents and DRS-conditions to form the new resolved DRS.

```
matchDrs(X,drs(D1,C1),drs(D2,C2),drs(D3,C3)):-
   member(X,D2),
   mergeLists(D1,D2,D3),
   mergeLists(C1,C2,C3),
   consistentConditions(X,Conds).
```

This is done with the help of mergeLists/3. It works very much like append/3, but it prevents that multiple occurences of items will turn up in the resulting list. This is how it is coded:

```
mergeLists([],L,L).

mergeLists([X|R],L1,L2):-
        member(Y,L1),
        X==Y, !,
        mergeLists(R,L1,L2).

mergeLists([X|R],L1,[X|L2]):-
        mergeLists(R,L1,L2).
```

Matching of two DRSs only succeeds if the resulting DRS is internally consistent. We'll use the consistent predicate from Chapter 6 for a first check (for complex cases, we have to rely on deep automated reasoning—see Chapter 11). A DRS is consistent with respect to a discourse referent x, if it is not the case that we can find two conditions on x that are not consistent. Here is the implementation in Prolog:

```
consistentConditions(X,Conds):-
   \+ (
          member(Cond1,Conds),
          member(Cond2,Conds), \+ Cond1=Cond2,
          compose(Cond1,Symbol1,[Y]), Y==X,
          compose(Cond2,Symbol2,[Z]), Z==X,
          \+ consistent(Symbol1,Symbol2)
      ).
```

And this finishes our basic implementation of Van der Sandt's algorithm. Of course we follow our computational tradition and design a driver predicate that allows easy input and provides us with readable output. The following driver computes all readings that our implementation generates (using the built-in Prolog predicate findall/3):

```
parse:-
```

```
    readLine(Discourse),
    d(Drs1,Discourse,[]),
    betaConvert(Drs1,Drs2),
    findall(Drs4,(
                  projectDrs([Drs2,pre([])]-[Drs3,pre(P)]),
                  accommodate(P,Drs3-Drs4),
                  checkBinding(Drs4)
                 ),
            Readings),
    printReadings(Readings).
```

Let's see how our implementation works for simple discourses. Here is an example of a simple definite. Two readings are generated, one caused by accommodation, and one caused by linking.

```
?- parse.

> A woman smokes. The woman collapses.

Readings:
1 drs([A],[collapse(A),smoke(A),woman(A)])
2 drs([A,B],[woman(A),collapse(A),smoke(B),woman(B)])
```

Experimenting with this basic implementation will probably surprise you. Even for simple examples, the program will produce a high number of readings. Here is an example that has only two anaphoric expressions (the proper name 'Vincent' and the pronoun 'it') but produces four readings—our implementation seems to over-generate readings:

```
?- parse.

> Vincent eats a big kahuna burger. Every criminal likes it.

Readings:
1 drs([A,B],[male(A),A=vincent,drs([C],[criminal(C)])>drs([D],[nonhuman(D),like(C,D)]),eat(A,B),bkburger(B)])
2 drs([A,B],[male(A),A=vincent,drs([C,D],[nonhuman(C),criminal(D)])>drs([],[like(D,C)]),eat(A,B),bkburger(B)])
3 drs([A,B],[male(A),A=vincent,drs([C],[criminal(C)])>drs([],[like(C,B)]),nonhuman(B),eat(A,B),bkburger(B)])
4 drs([A,B,C],[male(A),A=vincent,nonhuman(B),drs([D],[criminal(D)])>drs([],[like(D,B)]),eat(A,C),bkburger(C)])
```

Exercise 4.3.3 Inspect the readings above and explain why they are generated by our implementation of Van der Sandt's algorithm.

Note that the problem is not purely one of producing too many readings, but one of producing readings which are unequally acceptable, although not impossible to rule out at all. We will attack this problem in the next session, by using a series of linguistic heuristics to judge the readings.

Exercise 4.3.4 Change the program in such a way that it doesn't support intermediate accommodation (for example, accommodation in the antecedent DRS of an implicational condition).

Free Variable Trapping

Nested presuppositions can have free variables in their α-DRS, which can potentially escape from their orgininal binder when accommodated on a more global level. One of the notorious examples of this branch of problems is the solutions we generate for 'Every boxer enjoys his big kahuna burger'. The presupposition trigger 'his big kahuna burger' gives rise to local, intermediate, and global accommodation. Under global accommodation, a free variable will appear in the main DRS. In fact, this reading is not available at all.

This is serious problem, and we have to deal with it in a satisfactory way. But note that this problem is not entirely new, we discussed it in the description of Van der Sandt's algorithm, and proposed the free variable check as a solution.

Exercise 4.3.5 Add a free variable check on the generated solutions and test the modified programs on examples like 'every boxer likes his wife'.

4.4 Optimizing the Algorithm

The current implementation of Van der Sandt's algorithm faces the problem that it generates a high number of readings, of which some are more likely to appear than others, but this is not stated in the output.

This comes from being too tolerant with regard to linking and accommodation. Given a presupposition or anaphor, we are equally happy with accommodation or linking. Similarly, we have equal opnion whether it is resolved locally or globally. We just don't care. But this is not the intuitive way of dealing with it. Linking, for instance, is preferred to accommodation. And global accommodation is preferable to local accommdation. Moreover, these heuristics depend on the kind of trigger we deal with: proper names behave quite differently from pronouns or definite descriptions in this respect.

Let's have a look how the different anaphoric constructs behave, and what kind of heuristic we could attach to it.

Proper Names

Proper names are—just like any other presuppostion trigger—allowed to accommodate anywhere. But this is far too loosely formulated. Local or intermediate accommodation is not something one would expect for proper names (unlike other definites descriptions), and neither is linking on a non-global accommodation (although it is not to be totally excluded). A simple way of dealing with this is to give global accommodation and linking a much higher preference to local operations.

Exercise 4.4.1 Try to think of examples in English where proper names accommodate or link at a non-local level of discourse structure.

Related to this problem, is the fact that, at least for proper names, accommodate should only be an option if linking is impossible. Otherwise, for a discourse 'Vincent knows Butch. Butch is a boxer.', there will be a reading where there are two Butches, each having a different discourse referent. Again, this is not impossible, but it certainly is the less preferred option.

Pronouns

Pronouns have less accommodation power than proper names. On subordinate levels accommodation is perhaps totally impossible (well, try to think of an example), and on the global level it is possible (you might have missed part of the text or conversation) but it is certainly less preferred than linking.

Exercise 4.4.2 Discuss in what cases one would perhaps allow global or local accommodation of pronouns.

Definite Descriptions

These easily tend to accommodate, also on non-global levels. But of course, linking, if possible, is to be preferred to accommodation.

Adding Scores

What we are going to do is to attach to each reading a judgement. This judgement is a score between 0 and 1, the higher the score, the more preferred this reading is.

Intuitively, the score says something about the processing costs involved for a reading. The ideal score of 1 means that there no difficulties whatsoever to process the sentence under this reading. The extreme low score of 0, on the contrary, means that is is impossible to process the sentence under this reading.

So what we do is simple and straightforward: we extend the implementation of presupposition projection with one extra parameter: the score. At the beginning it gets assigned the value 1, and as we proceed our analysis, it will get updated by the different operations we have to perform. Accommodation will change the score in a negative way, but linking, since it is preferred to accommodation, in a less negative way.

We will use the notation `ScoreIn-ScoreOut` in the predicates, where `ScoreIn` is the score before the predicate is proved, and `ScoreOut` is the new value of the score if the proof was succesful. All the clauses for `projectDrs` and `projectConds` get this extra argument. For instance:

```
projectDrs([alfa(X,Type,B1,B3)|A1]-A5,S1-S4):-
        projectDrs([B1|A1]-[B2|A2],S1-S2),
        resolveAlfa(X,Type,B2,A2-A3,S2-S3),
        projectDrs([B3|A3]-A4,S3-S4),
        addBindingInfo(X,Type,A4-A5).
```

The value of the score is changed when we resolve the α-DRS, because that's exactly the point what kind of anaphoric construct we deal with, where it is resolved, and how it is resolved. All these factors play a role for determining the new score.

Here is how the scores are changed. Basically, it is done by multiplying the old score with a factor (ranging from 0 to 1), and assigning the result to the new score. (Note that this ensures that the resulting value will never outrange the scope 0-1.)

```
resolveAlfa(X,Type,AlfaDrs,[drs(D,C)|Others]-[New|Others],S1-S2):-
        global(Others),
        matchDrs(X,AlfaDrs,drs(D,C),New),
        (Type=refl, S2 = S1;
         Type=nonrefl, S2 = S1;
         Type=def, S2 = S1;
         Type=name, S2 = S1).

resolveAlfa(X,Type,AlfaDrs,[drs(D,C)|Others]-[New|Others],S1-S2):-
        nonglobal(Others),
        matchDrs(X,AlfaDrs,drs(D,C),New),
        (Type=refl, S2 is S1 * 1;
         Type=nonrefl, S2 is S1 * 1;
         Type=def, S2 is S1 * 0.5;
         Type=name, S2 is S1 * 0.2).

resolveAlfa(_,Type,Alfa,[pre(A)]-[pre([Alfa|A])],S1-S2):-
        global([pre(A)]),
        (Type=nonrefl, S2 is S1 * 0.5;
         Type=def, S2 is S1 * 0.9;
         Type=name, S2 is S1 * 0.9).

resolveAlfa(_,Type,Alfa,[pre(A)|Others]-[pre([Alfa|A])|Others],S1-S2):-
        nonglobal([pre(A)|Others]),
        (Type=nonrefl, S2 is S1 * 0.1;
         Type=def, S2 is S1 * 0.7;
         Type=name, S2 is S1 * 0.2).
```

```
resolveAlfa(X,Type,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others],S1-S2):-
        global([pre(A1)|Others]),
        deleteFromList(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New),
        (Type=refl, S2 = S1;
         Type=nonrefl, S2 = S1;
         Type=def, S2 = S1;
         Type=name, S2 = S1).

resolveAlfa(X,Type,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others],S1-S2):-
        nonglobal([pre(A1)|Others]),
        deleteFromList(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New),
        (Type=refl, S2 is S1 * 1;
         Type=nonrefl, S2 is S1 * 1;
         Type=def, S2 is S1 * 0.5;
         Type=name, S2 is S1 * 0.2).
```

Our revised predicate `resolveAlfa` is now turned into a *tuner* for presupposition projection. If you don't like the value, just take a screwdriver and change them what you think is more accurate (maybe you want to add more anaphoric types).

To distinguish between global and local accommodation, `resolveAlfa` has more clauses than its predecessor. It uses two auxiliary predicates to determine the level of DRS given the current situation of the stack. The global level of DRS is a situation where the stack contains exactly one `pre/1` term (and this is the last item on the stack). This is coded by `global/1`:

```
global([pre(_)]).
global([drs(_,_)|Stack]):- global(Stack).
```

The stack determines a non-global level of discourse if there at least two occurrences of `pre/1` on it. This is coded by `nonglobal/1`:

```
nonglobal([pre(_),drs(_,_)|_]).
nonglobal([drs(_,_)|Stack]):- nonglobal(Stack).
```

The driver has to be adapted too. All we do is add the calculated score to the generated readings.

```
parse:-
```

```
    readLine(Discourse),
    d(Drs1,Discourse,[]),
    betaConvert(Drs1,Drs2),
    findall((Score:Drs4),(
                  projectDrs([Drs2,pre([])]-[Drs3,pre(P)],1-Score),
                  accommodate(P,Drs3-Drs4),
                  checkBinding(Drs4)
                 ),
            Readings),
    printReadings(Readings).
```

Here is the same example as we run with our first implementation, but now the scores are made visible:

```
?- parse.

> A woman smokes. The woman collapses.

Readings:
1 1:drs([A],[collapse(A),smoke(A),woman(A)])
2 0.9:drs([A,B],[woman(A),collapse(A),smoke(B),woman(B)])
```

Our implementation with scores gives a perfect score of 1 to the reading where 'the woman' is linked to the discourse referent introduced by 'a woman', and a score of 0.9 to the reading where 'the woman' accommodates a discourse referent. How nice!

Now for the other example:

```
?- parse.

> Vincent eats a big kahuna burger. Every criminal enjoys it.

Readings:
1 0.9:drs([A,B],[male(A),A=vincent,drs([C],[criminal(C)])>drs([],[enjoy(C,B)]),nonhuman(B),eat(A,B),bkburger(B)])
2 0.45:drs([A,B,C],[male(A),A=vincent,nonhuman(B),drs([D],[criminal(D)])>drs([],[enjoy(D,B)]),eat(A,C),bkburger(C)])
```

Exercise 4.4.3 Compare the readings for 'Vincent eats a big kahuna burger. Every criminal enjoys it.' with the ones that we got with our first version of presupposition resolution. Which readings are suppressed and why

Exercise 4.4.4 [easy] Change the program such that it allows local accommodation of proper names.

Software Summary of Chapter 4

`mainPresupDRT.pl` Implementation of Van der Sandt's presupposition resolution algorithm for DRSs. (page 196)

`mainPresupScoreDRT.pl` Implementation of Van der Sandt's presupposition resolution algorithm for DRSs, with score computation for each generated reading. (page 200)

`resolvePresup.pl` Implementation of Van der Sandt's presupposition resolution algorithm for DRSs. (page 197)

`resolvePresupScore.pl` Implementation of Van der Sandt's presupposition resolution algorithm for DRSs, with score computation for each generated reading. (page 201)

`matchDRT.pl` Code for partial matching of DRSs. (page 199)

`semMacrosPresupDRT.pl` Definitions of the semantic macros for the presupposition projection programs. (page 204)

Notes

By far the most important reference for the work of this chapter is Van der Sandt's classic paper: (Van der Sandt 1992). This contains a wealth of examples, and locates the approach, both intellectually and historically, in the complex terrain of presupposition theory. For a recent overview of this terrain from a rather different perspective, see David Beaver's survey article: (Beaver 1997).

Van der Sandt attributes the term accommodation for presupposition to Lewis (Lewis 1979). Accommodation is a hot topic in presupposition theory. Although most tend to agree with global and local forms of accommodation, *intermediate* accommodation is heavily disputed.

Chapter 5

Acceptability Constraints

This chapter is devoted to implementing the acceptability constraints imposed by Van der Sandt on presupposition resolution and accommodation. We begin by explaining how to implement the consistency and informativity constraints. We then describe and implement the local versions of these constraints that Van der Sandt also imposes.

In essence this chapter revolves around a single theme: integrating semantic construction and inference. As we shall show, it is possible to develop an interesting interactive discourse system by hooking together a theorem prover (here we use the widely available Otter system) with the semantic construction tools we have been developing in this book.

5.1 Maxims of Conversation

Humans communicate successfully on the basis of a simple conversational principle: cooperation between speaker and hearer. Normally, speakers do not give less or more information than is appropriate, they don't say something they believe is false or already known. Two very general maxims which we plan to implement in our DRS-toolkit are:

1. each contribution to the discourse should be consistent with it, and
2. each contribution should introduce new information.

The first maxim, 'be consistent!', is an obvious constraint on discourse contributions. One should not contradict oneself. Obeying this rule, the following discourse are unacceptable:

> Mia is a man. Mia is a woman.
> Jody is married. Jody does not have a husband.

Incidentally, note that the inconsistency of these discourses is not a fact of pure logic; it depends on additional information, namely that men are not women, that women are not men, that Jody is a woman, and that married women have husbands.

The second maxim, 'be informative!', is a constraint on the newness of information. Every contribution to the discourse should introduce new information. This too is quite natural. Discourses that violate this principle are:

> Jody is a boxer. Jody is a boxer.
> Mia is a married. She has a husband.

Again, note that while the first inference is purely logical, the second inference hinges on our knowledge that Mia is a woman, and that married women have husbands.

It is important to realize that these maxims are not always obeyed by humans. (Obviously, liars do not.) They are, however, the default setting in which humans analyze utterances. Violation of consistency appears when somebody *corrects* himself/herself, as for instance in:

> Jody is a boxer. No, Jody is *not* a boxer.

The informativity constraint (which, perhaps, has less priority than consistency) can be violated as well, and assertions may well be part of the established conversation. Utterances may be repeated for emphasis:

> Jody is *not* a boxer, she is *not* a boxer.

Consistency and informativity are two of the acceptability constraints Van der Sandt imposes on accommodation, and we are going to build these constraints into a small interactive discourse system. The system is set up as follows. The user can build up a discourse by inputting a sentence to the system. Using the DCG-parser and the DRS-construction program, we build a DRS for this sentence, and integrate it with the DRS from the previous contribution. Presuppositions will be handled as in Chapter 10, but now we're going to use a theorem prover to check which accommodation possibilities obey the maxims.

5.2 Choosing a Theorem Prover

Which theorem prover are we are going to use? For a start, standard theorem provers are not able to process DRSs directly, so we need to translate DRSs to first-order logic. But we know from Chapter 7 that this is no big deal: we have a compiler, `drs2fol` which handles this efficiently.

Can we use our theorem prover from Chapter 5? Mostly, yes—but not always. DRS languages translate to first-order logic *with equality*. Our theorem prover cannot deal with equality, so we have to look for something different.

Exercise 5.2.1 [hard] Actually, for the grammar fragment under consideration, most equality conditions in DRSs can be compiled away in the translation to first-order logic. Write a predicate that does this.

Exercise 5.2.2 [easy] Think of examples that invoke equality conditions that cannot be compiled out.

Fortunately, there are a number of freely available provers that handle equality. One of the most widely available is Otter, and we will use Otter here for our theorem proving work.

We'll hook up Otter to our Prolog-programs by defining an interface predicate called `callTheoremProver/3`. (This is done on the basis of a Unix platform, so you might want to redefine this interface if you're using another operating system.) The interface works as follows: with `fol2otter/2` the axioms and formulas are written to the file `temp.in` in Otter-syntax. (We won't discuss this little translator. It simply translates our first-order notation to the notation Otter expects and is a straightforward piece of code.) Then, using the Sicstus in-built predicate `shell/2`, we start Otter, supplying it with the input file, and let it output `temp.out`. The second argument of `shell/2` returns the state of the process. For some mysterious reasons, this state variable returns the value 26386 if Otter found a proof, and 26624 when it didn't succeed in finding a proof, and that's why we'll implement the interface like this:

```
callTheoremProver(Axioms,Formula,Proof):-
   fol2otter(Axioms,Formula),
   shell('./otter < temp.in > temp.out',X),
   (X=26368,Proof=yes;X=26624,Proof=no).
```

Checking for consistency boils down to translating the DRS to first-order logic—call this formula Φ—and then checking that Φ is consistent. For the theorem proving case this means we need to find out whether $\neg\Phi$ is valid: if it is, then Φ is inconsistent, and if it is not, then Φ is consistent. For the model building case this involves finding out whether Φ is satisfiable: if it is not, then Φ is inconsistent. This is summarized in the following table:

Now, Otter is a refutation based theorem prover, which means that we have to give it the *negation* of what we are trying to prove, thus we have to give it $\neg\neg\Phi$, or equivalently Φ, as input.

How do we test for informativity? Well, if the new DRS follows from the DRS of the previous discourse, then the new DRS does not encode new information. Given B_{new}

Table 5.1: Consistency checking for theorem provers and model builders

$\Phi \setminus \neg\Phi$	**valid**	?	**invalid**
satisfiable	–	consistent	consistent
?	inconsistent	?	consistent
not satisfiable	inconsistent	inconsistent	–

encoding the new information, and B_{old} the information contributed so far, we translate a DRS with solely the condition $B_{old} \Rightarrow B_{new}$ to first-order logic, say to a formula Φ. If Φ is a theorem or not satisfiable, we have no new information. Or for a complete picture of this setting, consult Table 5.2:

Table 5.2: Informativity checking for theorem provers and model builders

$\neg\Phi \setminus \Phi$	**valid**	?	**invalid**
satisfiable	–	informative	informative
?	not informative	?	informative
not satisfiable	not informative	not informative	–

One other thing needs to be done. We are *not* interested merely in what can be proved without supporting assumptions. Quite the contrary: we are interested in what can be proved given background knowledge (for example, that married woman have husbands, that Mia is a woman, and so on). Using `backgroundKnowledge(Drs)`, an interface to certain facts of the world coded in first-order formulas the following code for our consistency check takes such knowledge into account.

```
consistent(NewDrs):-
   backgroundKnowledge(Chi),
   drs2fol(NewDrs,Phi),
   callTheoremProver(Chi,Phi,Proof),
   (Proof=yes, !, fail; true).
```

There is a similar predicate for informativity. Here, too, we need to take background knowledge into account. So the actual coding for the check on informativity is as follows:

```
informative(drs([],[]),_):-!.
informative(OldDrs,NewDrs):-
   backgroundKnowledge(Chi),
   drs2fol(OldDrs,Phi),
   drs2fol(NewDrs,Psi),
   callTheoremProver(Chi,~(Phi > Psi),Proof),
   (Proof=yes, !, fail; true).
```

What about background knowledge? For a start, we use the information our ontology of noun objects gives us (see Chapter 6). When we discuss the local constraints, we'll extend the background knowledge with facts that are not covered by the ontology information (for instance, that married women have husbands).

```
backgroundKnowledge(Formulas):-
   generateOntology(Formulas).
```

Summing up, we now have predicates which take ordinary DRSs as input and tell us whether the acceptability constraints are violated or not. So now we're ready to combine our DRS-construction tools with our inference-tools, and integrate them into a small interactive system.

5.3 Chatting with Curt

The best way to show how the acceptability constraints work is to implement them in a small interactive program. We'll develop such a program that accepts utterances of the user and comments on them according to the maxims of conversation—we'll call this system Curt. While Curt is short for "Clever use of reasoning tools", it really owes its name to the brief (and indeed, rather rude) replies it gives.

Two (mutually recursive) predicates implement Curt: `curtInput` and `curtOutput`. The user's input is handled by `curtInput/1`:

```
curtInput(Readings):-
   readLine(Input),
   curtOutput(Input,Readings).
```

By `readLine/1` the user's input is read, and passed on (together with the current list of readings, represented as DRSs) to `curtOutput`. The predicate `curtOutput/2` checks the input and accordingly gives an appropriate response. If Curt finds a parse, it checks whether the readings it obtained are consistent (otherwise it objects), and informative (otherwise it will say so). The readings that remain are passed on to `curtInput/1`, and the user can enter a new sentence, which will be integrated into the DRS encoding the previous utterances.

```
curtOutput(Input,Readings):-
   d(MergeDrs,Input,[]),!,
   betaConvert(MergeDrs,ReducedDrs),
   getReadings(ReducedDrs,Readings,PotentialReadings),
   consistentReadings(PotentialReadings,ConsistentReadings),
   (
       ConsistentReadings=[],
       curtSays('No! I do not believe that!'),!,
       curtInput(Readings)
   ;
       informativeReadings(ConsistentReadings,InformativeReadings),
       (
           InformativeReadings=[],
           curtSays('Yes, I knew that!'),!,
           curtInput(Readings)
       ;
           curtSays('Ok.'),
           curtInput(InformativeReadings)
       )
   ).
```

The DRSs of the input sentence are computed with the help of the predicate `getReadings`. This predicate uses the module for presupposition projection (Chapter 10) to deal with pronouns and presuppositions.

```
getReadings(ProtoDrs,Readings,PotentialReadings):-
   findall((OldDrs,NewDrs),
           (
            member(OldDrs,Readings),
            projectDrs([merge(OldDrs,ProtoDrs),pre([])]-[Drs,pre(P)]),
            accommodate(P,Drs-NewDrs)
           ),
           PotentialReadings).
```

What if our parser fails to produce a result? The next clause implements Curt's behaviour for cases where it is not able to analyse the user's input.

```
curtOutput(_,Readings):-
```

```
        curtSays('What?'),
        curtInput(Readings).
```

In addition, we have some meta-commands to steer Curt. The input '`bye`' ends the conversation, '`new`' clears the DRSs and starts a new chat, and '`drs`' displays the current readings in DRS format.

```
    curtOutput([bye],_):- !,
        curtSays('Bye bye!').

    curtOutput([new],_):- !,
        curt.

    curtOutput([drs],Readings):- !,
        printReadings(Readings),
        curtInput(Readings).
```

Finally, there is a driver predicate that starts a chat-session with Curt:

```
    curt:-
        curtInput([drs([],[])]).
```

Her is an example session. (The input of the user is preceded by a `>`, the rest is output of the system.)

```
?- curt.

> Marsellus loves Butch and Mia.

Ok.

> Mia is a boxer.

Ok.

> Butch is a boxer.

Ok.

> Marsellus loves one boxer.

No! I do not believe that!

>
```

Is Otter a useful prover for this application? It's pretty good—indeed it's light years ahead of the simple theorem prover we first experimented with. In particular, when something is a theorem it's very good at finding a proof. However when something is not a theorem, it may run into problems. The reader is asked to do the following exercise to experience this.

Exercise 5.3.1 Consider the sequence 'Marsellus loves a boxer' followed by 'Marsellus loves one boxer'. Is this is a consistent and informative discourse? Try the program on the input sequence and find out which formula Otter tries to proof.

One line of experiment has been to simply substitute other theorem provers for Otter. This alone has lead to interesting results, but there is another line of development that deserves much more thorough exploration, namely the use of model builders to filter out non-theorems. It is true that, when all is said and done, first-order inference is an undecidable problem—nontheless, our experiments so far indicate that with a decent theorem prover to pin down the theorems, and a model builder to weed out the non-theorems, a striking amount can be done very speedily.

5.4 Adding Local Constraints

Van der Sandt's algorithm for presupposition projection actually uses the two maxims given above together with the the following extensions: superordinated DRSs should neither imply a subordinated DRS, nor a negated subordinated DRS. This constraint rules out the reading where the presupposition that Mia has a husband is globally accommodated:

If Mia is married then her husband is out of town.

This is because the fact that Mia has a husband follows from our background knowledge that she is a woman and is married. We will extend our discourse system to deal with this. To apply the local constraints to a certain DRS B, we have to calculate ordered pairs of DRSs, where each pairs contains super- and subordinated DRSs of B. But what exactly these pairs of super-sub DRSs, and how are they computed? The best way to show the idea is giving an example. Assume we have the following DRS with sub-DRSs A, B, and C, structured as follows:

$$\begin{array}{|l|}\hline \\ \hline \neg\ A \\ B \Rightarrow C \\ \hline\end{array}$$

Then the pairs of super-sub DRSs are:

1. $\left(\begin{array}{|c|}\hline \quad \\ \hline B \Rightarrow C \\ \hline \end{array} , A \right)$
2. $\left(\begin{array}{|c|}\hline \quad \\ \hline \neg\ A \\ \hline \end{array} , B \right)$
3. $\left(\begin{array}{|c|}\hline \quad \\ \hline \neg\ A \\ \hline \end{array} \oplus B, C \right)$

This is coded in Prolog using `superSubDrs/3`. The first argument is the DRS that is traversed, the second argument the superordinated, and the last argument the subordinated DRS. Note that the superordinated DRS is represented as a difference list—this is done while conditions that contain the sub-DRS are removed from the superordinated DRS.

```
superSubDrs(drs(D,[Sub > _|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[B > Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(merge(drs(D,C),B),Drs),Super).
superSubDrs(drs(D,[B1 > B2|C]),Drs-Super,Sub):-
   superSubDrs(B2,merge(Drs,merge(merge(drs(D,C),B1),B2))-Super,Sub).
superSubDrs(drs(D,[Sub v _|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[_ v Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[B v _|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[_ v B|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[~ Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[~ B|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[Cond|C]),Drs-Super,Sub):-
   superSubDrs(drs([],C),Drs-B,Sub),
   mergeDrs(merge(drs(D,[Cond]),B),Super).
```

We'll use this new predicate `superSubDrs` to traverse through the DRS and compute pairs of super- and subordinated DRSs in our definition of the predicates for the local constraints. Here is the predicate for local informativity:

```
localInformative(Drs):-
```

```
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalInformative(List).

allLocalInformative([]).
allLocalInformative([(Super,Sub)|Others]):-
   backgroundKnowledge(Chi),
   drs2fol(drs([],[Super>Sub]),Phi),
   callTheoremProver(Chi,~Phi,Proof),
   (Proof=yes, !, fail; allLocalInformative(Others)).
```

Similarly, we can code local consistency:

```
localConsistent(Drs):-
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalConsistent(List).

allLocalConsistent([]).
allLocalConsistent([(Super,Sub)|Others]):-
   backgroundKnowledge(Chi),
   drs2fol(drs([],[Super>drs([],[~Sub])]),Phi),
   callTheoremProver(Chi,~Phi,Proof),
   (Proof=yes, !, fail; allLocalConsistent(Others)).
```

How do we integrate our appeal to the local constraints in Curt? We'll uses these as a filter on the readings, and only apply them to readings that are consistent and informative. The local constraints are less important than their global versions. This is how we integrate it in the predicate `curtOutput/2`:

```
curtOutput(Input,Readings):-
   d(MergeDrs,Input,[]),!,
   betaConvert(MergeDrs,ReducedDrs),
   getReadings(ReducedDrs,Readings,PotentialReadings),
   consistentReadings(PotentialReadings,ConsistentReadings),
   (
    ConsistentReadings=[],
    curtSays('No! I do not believe that!'),!,
    curtInput(Readings)
   ;
    informativeReadings(ConsistentReadings,InformativeReadings),
    (
     InformativeReadings=[],
     curtSays('Yes, I knew that!'),!,
     curtInput(Readings)
    ;
```

```
          curtSays('Ok.'),
          localInformativeReadings(InformativeReadings,LocalInformative),
          localConsistentReadings(LocalInformative,LocalConsistent),
          (
           LocalConsistent=[], !,
           SelectedReadings=InformativeReadings
          ;
           SelectedReadings=LocalConsistent
          ),
          curtInput(SelectedReadings)
        )
       ).
```

Exercise 5.4.1 Change this predicate in the following way. Only accept readings that do not violate (global) consistency and (global) informativity, and from these, select those with the *least* number of violations of the local constraints. (This strategy for applying the local constraints is due to David Beaver).

Background knowledge is encoded in first-order formulas. Of course, we want to extend the knowledge derived from the lexicon (Chapter 6) with common-sense facts of the world, such as married persons (if they are female) have husbands, and so on. Here is the new encoding of background knowledge.

```
backgroundKnowledge(Formulas):-
   knowledge(Formulas1),
   generateOntology(Formulas2),
   append(Formulas1,Formulas2,Formulas).

knowledge([
           forall(X,forall(Y,have(X,Y)>of(Y,X))),
           forall(X,forall(Y,of(Y,X)>have(X,Y))),
           forall(X,female(X)&married(X)>exists(Y,husband(Y)&have(X,Y))),
           forall(X,forall(Y,husband(Y)&have(X,Y)>married(X)&female(X)))
          ]).
```

Exercise 5.4.2 Try Curt on examples 'If Mia is married then her husband is out of town' and 'Either Vincent does not have a car, or he cleans his car', and see how Curt handles the presupposition triggers 'her husband' and 'his car'.

Software Summary of Chapter 5

`curt.pl` The main predicates that build the framework of our interactive discourse system. (page 206)

`acceptabilityConstraints.pl` The implementation of consistency, informativity, and the local constraints. (page 209)

`callTheoremProver.pl` The Prolog-interface to the theorem prover Otter. (page 213)

`fol2otter.pl` Translates a formula in Otter syntax to standard output. (page 215)

Notes

The automatic deduction system Otter (McCune and Padmanabhan 1996) that we put together with our Prolog programs (we used version 3.0.5) is freely available on the World Wide Web, via:

`http://www-c.mcs.anl.gov/home/mccune/ar/otter/`

The DORIS system is an extended version of Curt (Blackburn, Bos, Kohlhase, and de Nivelle 1999). DORIS uses the MathWeb society of inference engines (currently, it uses the theorem provers Spass and Bliksem) for its reasoning tasks (Franke and Kohlhase 1999). Using the internet to communicate the inference problems and their answers, the theorem provers work on the problems in parallel running of different machines.

Chapter 6

Putting It All Together

6.1 Combining Scope and Presupposition

Exercise 6.1.1 Add the scores.

6.2 Adding Focus

Exercise 6.2.1 Use a theorem prover to sort the readings generated by the presupposition projection program.

Software Summary of Chapter 6

`mainPresupDRTU.pl` DRTU combined with presupposition resolution. (page 217)

`semMacrosPresupDRTU.pl` Semantic macros for DRTU and presupposition resolution. (page 218)

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!

Bibliography

Asher, N. (1993). *Reference to Abstract Objects in Discourse*. Dordrecht: Kluwer.

Beaver, D. I. (1997). Presupposition. In J. Van Benthem and A. Ter Meulen (Eds.), *Handbook of Logic and Language*, Chapter 17, pp. 939–1008. Elsevier, MIT.

Blackburn, P., J. Bos, M. Kohlhase, and H. de Nivelle (1999). Inference and Computational Semantics. In H. Bunt and E. Thijsse (Eds.), *Third International Workshop on Computational Semantics (IWCS-3)*, pp. 5–19. Computational Linguistics, Tilburg University.

Bos, J., E. Mastenbroek, S. McGlashan, S. Millies, and M. Pinkal (1994). A Compositional DRS-based Formalism for NLP Applications. In H. Bunt, R. Muskens, and G. Rentier (Eds.), *International Workshop on Computational Semantics*. University of Tilburg.

Bratko, I. (1990). *Prolog; Programming for Artificial Intelligence* (Second Edition ed.). Addison-Wesley.

Charniak, E. (1972). *Towards a Model of Children's Story Comprehension*. Ph. D. thesis, Massachusetts Institute of Technology.

Charniak, E. and D. McDermott (1985). *Introduction to Artificial Intelligence*. Addison-Wesley.

Charniak, E. and Y. Wilks (Eds.) (1976). *Computational Semantics. An Introduction to Artificial Intelligence and Natural Language Comprehension*, Volume 4 of *Fundamental Studies in Computer Science*. Amsterdam/New York/Oxford: North-Holland Publishing Company.

Chomsky, N. (1988). *Lectures on Government and Binding. The Pisa Lectures*. Studies in Generative Grammar. Foris Publications.

Clocksin, W. and C. Mellish (1987). *Programming in Prolog* (Third, Revised and Extended Edition ed.). Springer Verlag.

Cooper, R., I. Lewin, and A. W. Black (1993). Prolog and Natural Language Semantics. Notes for AI3/4 Computational Semantics, University of Edinburgh.

Deransart, P., A. Ed-Dbali, and L. Cervoni (1996). *Prolog: The Standard. Reference Manual*. Springer.

Fauconnier, G. (1985). *Mental Spaces. Aspects of Meaning Construction in Natural Language.* Bradford Books. The MIT Press.

Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving* (second ed.). Springer-Verlag.

Franke, A. and M. Kohlhase (1999). System description: Mathweb, an agent-based communication layer for distributed automated theorem proving. In *16th International Conference on Automated Deduction CADE-16*.

Frey, W. (1985). Syntax and Semantics of some Noun Phrases. In J. Laubsch (Ed.), *8th German Workshop on Artificial Intelligence (GWAI-84)*, Informatik Fachberichte 103, pp. 49–63.

Gamut, L. (1991). *Logic, Language, and Meaning. Volume II. Intensional Logic and Logical Grammar.* Chicago and London: The University of Chicago Press.

Groenendijk, J. and M. Stokhof (1990). Dynamic Montague Grammar. In L. Kálmán and L. Pólos (Eds.), *Papers from the Second Symposium on Logic and Language*, pp. 3–48.

Grosz, B. J., A. K. Joshi, and S. Weinstein (1995). Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics 21*(2), 203–225.

Grosz, B. J. and L. Sidner, Candace (1986). Attention, Intentions, and the Structure of Discourse. *Computational Linguistics 12*, 175–204.

Heim, I. (1982). *The Semantics of Definite and Indefinite Noun Phrases.* Ph. D. thesis, University of Massachusetts.

Hobbs, J. R. (1986). Resolving Pronoun References. In B. J. Grosz, K. Sparck Jones, and B. L. Webber (Eds.), *Readings in Natural Language Processing*, pp. 339–352. Morgan Kaufmann.

Johnson, M. and M. Kay (1990). Semantic Abstraction and Anaphora. In *COLING-90. Papers presented to the 13th International Conference on Computational Linguistics*, University of Helsinki, pp. 17–27.

Johnson, M. and E. Klein (1986). Discourse, anaphora and parsing. In *11th International Conference on Computational Linguistics. Proceedings of Coling '86*, University of Bonn, pp. 669–675.

Kamp, H. (1984). A Theory of Truth and Semantic Representation. In J. Groenendijk, T. M. Janssen, and M. Stokhof (Eds.), *Truth, Interpretation and Information; Selected Papers from the Third Amsterdam Colloquium*, pp. 1–41. Dordrecht - Holland/Cinnaminson - U.S.A.: Foris.

Kamp, H. and U. Reyle (1993). *From Discourse to Logic.* Dordrecht: Kluwer.

Kehler, A. (1997). Current theories of centering for pronoun interpretation. *Computational Linguistics 23*(3), 467–475.

Kohlhase, M., S. Kuschert, and M. Pinkal (1996). A Type-Theoretic Semantics for λ-DRT. In P. Dekker and M. Stokhof (Eds.), *Proceedings of the Tenth Amsterdam Colloquium*, ILLC/Department of Philosophy, University of Amsterdam, pp. 479–498.

Lewis, D. (1979). Scorekeeping in a Language Game. In R. Bäuerle, U. Egli, and A. von Stechow (Eds.), *Semantics from a different Point of View*, Volume 6 of *Springer Series in Language and Communication*, pp. 172–187. Springer-Verlag.

McCune, W. and R. Padmanabhan (1996). *Automated Deduction in Equational Logic and Cubic Curves.* Lecture Notes in Computer Science (AI subseries). Springer-Verlag.

Muskens, R. (1996). Combining montague semantics and discourse representation. *Linguistics and Philosophy 19*, 143–186.

Pereira, F. and S. Shieber (1987). *Prolog and Natural Language Analysis.* CSLI Lecture Notes 10. Stanford: Chicago University Press.

Reinhart, T. (1983). *Anaphora and Semantic Interpretation.* Chicago: The University of Chicago Press.

Sidner, C. L. (1986). Focusing in the Comprehension of Definite Anaphora. In B. J. Grosz, K. S. Jones, and B. L. Webber (Eds.), *Readings in Natural Language Processing*, pp. 363–394. Morgan Kaufman.

Smullyan, R. (1995). *First-Order Logic.* Dover Publications. This is a reprinted and corrected version of the 1968 Springer-Verlag edition.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog; Advanced Programming Techniques.* The MIT Press.

Van der Sandt, R. A. (1992). Presupposition Projection as Anaphora Resolution. *Journal of Semantics 9*, 333–377.

Van Eijck, J. and H. Kamp (1997). Representing discourse in context. In J. van Benthem and A. ter Meulen (Eds.), *Handbook of Logic and Language*, pp. 179–237. Elsevier.

Wada, H. and N. Asher (1986). BUILDRS: An Implementation of DR Theory and LFG. In *11th International Conference on Computational Linguistics. Proceedings of Coling '86*, University of Bonn, pp. 540–545.

Winograd, T. (1972). *Understanding Natural Language.* Academic Press.

Zeevat, H. (1989). A compositional approach to discourse representation theory. *Linguistics and Philosophy 12*, 95–131.

Appendix A

Propositional Languages

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

$$\text{CUSTOMER(y)} \rightarrow \text{LOVE(VINCENT,y)}$$

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

$$\forall y[\text{CUSTOMER(y)} \rightarrow \text{LOVE(VINCENT,y)}]$$

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 $\mathbf{M}$ (of appropriate vocabulary) and an assignment g in $\mathbf{M}$. 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(x). If we do this, then every quantifier-free formula is either true or false in $\mathbf{M}$ 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 LOVE(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 make use of the simpler propositional notation just defined.

Appendix B

Type Theory

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 generalize 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:

λx.(x@x)@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:

(λx.(x@x)@x)@(λx.(x@x)@x)

If we now apply β-conversion we get:

$((\lambda x.(x@x)@x)@(\lambda x.(x@x)@x))@(\lambda x.(x@x)@x)$

But note this is just the functional application followed by an additional occurrence of $\lambda x.(x@x)@x$! Obviously we can going on applying β-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?

In this appendix, 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 of Chapter 2.

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

The untyped lambda calculus adopts an 'anything goes' attitude to functional application. For example, $\lambda x.(x@x)@x$ 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 doesn'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 calculus, 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 characterizes 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 a 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 book 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 β-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.

Exercise B.0.1 Does our implementation of β-conversion allow self-application?

Exercise B.0.2 What happens when you functionally apply the formula λx.x@x to itself?

Appendix C

Theorem Provers and Model Builders

Automated reasoning has seen an enormous increase of performance of (especially first-order) inference engines and model builders. This appendix is a guide to a number of useful reasoning engines for computational semantics.

Theorem Provers

- BLIKSEM
  Resolution based theorem prover for first order logic with equality (Hans de Nivelle). Almost as fast as the speed of light.

  URL: `http://turing.wins.uva.nl/~mdr/ACLG/Provers/Bliksem/bliksem.html`

- SPASS
  First-order theorem prover (Christoph Weidenbach et al.)
  Winner at the CASC-15 in the divisions FOF and SAT.

  URL: `http://spass.mpi-sb.mpg.de/`.

- FDPLL:
  Or, if you prefer, the "First-Order Davis-Putnam-Loveland-Logeman" theorem prover (Peter Baumgartner). Good at satisfiable problems.

  URL: `http://www.uni-koblenz.de/~peter/FDPLL/`

- OTTER
  The 'classical' theorem prover by W. McCune.

  URL: `http://www-unix.mcs.anl.gov/AR/otter/`

Model Builders

- MACE
  Short for "Model and Counter-Examples". Mace is a Model builder for first-order logic with equality (McCune). Got the first price at CASC-16 in division SAT (working in tandem with OTTER).

  URL: `http://www-unix.mcs.anl.gov/AR/mace/`

- SATCHMO
  Satchmo is a model generator for first-order theories, implemented in Prolog.

  URL: `http://www.pms.informatik.uni-muenchen.de/software/`

- KIMBA
  Kimba is an Higher-Order Model Generator, a deduction system for abductive or circumscriptive reasoning within linguistic applications (Karsten Konrad).

  URL: `http://www.ags.uni-sb.de/~konrad/kimba.html`

Systems

- MATHWEB
  This system for automated theorem proving connects a wide-range of mathematical services by a common, mathematical software bus (Franke and Kohlhase 1999).

  URL: `http://www.ags.uni-sb.de/~omega/www/mathweb.html`

Appendix D

Prolog in a Nutshell

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.

The 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:

```
playsAirGuitar(mia):-
   listensToMusic(mia).
```

The `:-` should be read as "if", or "is implied by". 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 clauses 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

```
?- playsAirGuitar(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

```
listensToMusic(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:

```
playsAirGuitar(mia):-
   listensToMusic(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:

```
playsAirGuitar(mia):-
   listensToMusic(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:

```
playsAirGuitar(mia):-
   listensToMusic(mia).
playsAirGuitar(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.

Syntax

What exactly are the syntactic entities such as `woman(jody)` and `playsAirGuitar(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 `airGuitar` 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 `playsAirGuitar` predicate we would write it as `playsAirGuitar/1` to indicate that it takes one argument. We make use of this convention in the book.

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:

```
likeFootMassage(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

```
?- likeFootMassage(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.

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:

```
moreExpensive(X,Y):-
   costsaLittleMore(X,Y).

moreExpensive(X,Y):-
   costsaLittleMore(X,Z),
   moreExpensive(Z,Y).
```

The definition of the `moreExpensive/2` predicate is fairly typical of the way recursive predicates are defined in Prolog. Clearly, `moreExpensive/2` is (at least partially) defined in terms of itself, as the more `moreExpensive` 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 `costsaLittleMore` predicate, which occurs in both the first and second rules. (Significantly, the right hand side of the first rule makes no mention of `moreExpensive`.)

Let's see how Prolog makes use of such definitions. Suppose we had the following facts in our knowledge base:

```
costsaLittleMore(royaleWithCheese,bigKahunaBurger).

costsaLittleMore(fiveDollarShake,royaleWithCheese).
```

If we pose the query

```
?- moreExpensive(fiveDollarShake,bigKahunaBurger).
```

then Prolog goes to work as follows. First, it tries to make use of the first rule listed concerning `moreExpensive`. 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 `fiveDollarShake` and `Y` with `bigKahunaBurger` it obtains the following goal:

```
?- costsaLittleMore(fiveDollarShake,Z),
   moreExpensive(Z,bigKahunaBurger).
```

Prolog deduces `moreExpensive(fiveDollarShake,bigKahunaBurger)` if it can find a value for `Z` such that, firstly,

```
?- costsaLittleMore(fiveDollarShake,Z).
```

is deducible, and secondly,

```
?- moreExpensive(Z,bigKahunaBurger).
```

is deducible too. But there is such a value for `Z`: `royaleWithCheese`. It is immediate that

```
?- costsaLittleMore(fiveDollarShake,royaleWithCheese).
```

will succeed, for this fact is listed in the knowledge base. Deducing

```
?- moreExpensive(royaleWithCheese,bigKahunaBurger).
```

is almost as simple, for the first clause of `moreExpensive/2` reduces this goal to deducing

```
?- costsaLittleMore(royaleWithCheese,bigKahunBurger).
```

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!)

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.

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.

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
```

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.

```
?- likeFootMassage(Who)
Who = mia
yes

?- \+ \+ likeFootMassage(Who)
yes
```

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
```

There are two ways to influence Prolog's search strategy: using the *cut* to suppres backtracking, or changing the order of the clauses in the database.

The cut (written `!`, it is a built-in Prolog predicate with arity 0) 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.

The order of clauses play a role in Prolog's search strategy, since Prolog works on the database in a sequential way, and subgoals are proved from left to right in the search tree. (Sometimes the order even drastically determines the outcome of a proof.)

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.

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, note 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.

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 (Clocksin and Mellish 1987). For a leisurely, in-depth introduction to programming in Prolog, try Bratko (Bratko 1990). For a more theoretically oriented introduction, try Sterling and Shapiro (Sterling and Shapiro 1986). Finally, for an introduction specially geared towards computational linguistics, try Pereira and Shieber (Pereira and Shieber 1987).

Appendix E

Listing of Programs

This appendix includes the full program listings that are developed in this book. Most predicates are decorated with a short documentation, using the following notational conventions for its required argument instantiations:

`:Arg` Argument `Arg` should be instantiated to a term denoting a goal

`+Arg` Argument `Arg` should be instantiated

`-Arg` Argument `Arg` should not be instantiated

`?Arg` Argument `Arg` may or may not be instantiated

Since Prolog's birth in the beginning of the seventies, a number of Prolog dialects emerged, and not all agree on a syntax or the in-built predicates. The programs in this book follow the conventions of "Standard Prolog", the ISO international standard on Prolog (Deransart, Ed-Dbali, and Cervoni 1996), as close as possible. The following predicates are assumed to be built-in in your version of Prolog (such as for example Quintus or Sicstus Prolog):

All solutions

`bagof/3`

`findall/3`

Arithmetic comparison

`>` (arithmetic greater than)

`<` (arithmetic less than)

Arithmetic evaluation

`is/2` (evaluate expression)

Atomic term processing

`atom_chars/2` (conversion of atoms to character codes and vice versa)

Character input

get0/1

Clause creation and destruction

`asserta/1` (clause creation)

`retract/1` (clause destruction)

File consultation

[**File**|**Files**] (consult list of files)

List operation

`length/2`] (determine length of a list)

Logic and control

`,/2` (conjunction)

`;/2` (disjunction)

`!/0` (cut)

`fail/0`

`true/0`

\+ (not provable)

Operator definition

`op/3` (extending operator table)

Term comparison

`==/2` (term identical)

Term creation and decomposition

`arg/3`

`functor/3`

`=../2` (the "univ")

Term unification

`=` (unify)

Type testing

`atom/1`

`atomic/1`

`compound/1`

`nonvar/1`

`var/1`

Term output

`write/1`

`nl/0`

The following is a practical overview of the programs in this appendix. We start with the library files consulted by most of the other programs, and then give a chapter by chapter break-down description of the files.

All the files at one glance

```
/*************************************************************************

        name: comsemOperators.pl
     version: May 25, 1999
 description: Operator definitions
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(comsemOperators,[]).

/*========================================================================
   Operator Definitions
========================================================================*/

:- op(950,yfx,@).          % application
:- op(900,yfx,'<>').       % bin impl
:- op(900,yfx,>).          % implication
:- op(850,yfx,v).          % disjunction
:- op(800,yfx,&).          % conjunction
:- op(750, fy,~).          % negation
```

```
/*************************************************************************

         name: comsemPredicates.pl
      version: November 8, 1997
  description: Set of Prolog predicates
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(comsemPredicates,
          [member/2,
           select/3,
           append/3,
           simpleTerms/1,
           compose/3,
           unify/2,
           removeFirst/3,
           substitute/4,
           variablesInTerm/2,
           newFunctionCounter/1,
           printReadings/1,
           printRepresentation/1]).

/*========================================================================

   List Manipulation
   -----------------

member(?Element,?List)
   Element occurs in List.

select(?X,+OldList,?NewList).
   X is removed from OldList, resultin in NewList.

append(?List1,?List2,?List3)
   List3 is the concatenation of List1 and List2.

allExactMember(?Elements,?List)
   Elements all occur in List (no unification, exact match).

removeAll(?Item,?List,?Newlist)
   Newlist is the result of removing all occurrences of Item from List.

removeFirst(?Item,?List,?Newlist)
   Newlist is the result of removing the first occurrence of Item from
   List. Fails when Item is not member of List.

========================================================================*/

member(X,[X|_]).
```

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

select(X,[X|L],L).
select(X,[Y|L1],[Y|L2]):-
   select(X,L1,L2).

append([],List,List).
append([X|Tail1],List,[X|Tail2]):-
   append(Tail1,List,Tail2).

allExactMember([],_).
allExactMember([X|R],L):-
   memberOfList(Y,L),
   X==Y,
   allExactMember(R,L).

removeAll(_,[],[]).
removeAll(X,[X|Tail],Newtail):-
   removeAll(X,Tail,Newtail).
removeAll(X,[Head|Tail],[Head|Newtail]):-
   X \== Head,
   removeAll(X,Tail,Newtail).

removeFirst(X,[X|Tail],Tail) :- !.
removeFirst(X,[Head|Tail],[Head|NewTail]):-
   removeFirst(X,Tail,NewTail).

/*========================================================================

   Term Manipulation
   -----------------

simpleTerms(?List)
   List is a list of elements that are currently uninstantiated or
   instantiated to an atom or number. Uses built-in Quintus/Sicstus
   predicate simple/1.

compose(?Term,+Symbol,+ArgList)
compose(+Term,?Symbol,?ArgList)
   Composes a complex Term with functor Symbol and arguments ArgList.
   Uses the Prolog built-in =.. predicate.

variablesInTerm(+Term,?InList-?OutList)
   Adds all occurrences of variables in Term (arbitrarily deeply
   nested to the difference list InList-OutList.

========================================================================*/

simpleTerms([]).
```

```
simpleTerms([X|Rest]):-
    simple(X), simpleTerms(Rest).

compose(Term,Symbol,ArgList):-
    Term =.. [Symbol|ArgList].

variablesInTerm(Term,Var1-Var2):-
   compose(Term,_,Args),
   countVar(Args,Var1-Var2).

countVar([],Var-Var).
countVar([X|Rest],Var1-Var2):-
   var(X),!,
   countVar(Rest,[X|Var1]-Var2).
countVar([X|Rest],Var1-Var3):-
   variablesInTerm(X,Var1-Var2),
   countVar(Rest,Var2-Var3).


/*========================================================================

   Unification Predicates
   ----------------------

unify(Term1,Term2)
   Unify Term1 with Term2 including occurs check. Adapted from
   "The Art of Prolog" by Sterling & Shapiro, MIT Press 1986, page 152.

notOccursIn(X,Term)
   Succeeds if variable X does not occur in Term.

notOccursInComplexTerm(N,X,Term)
   Succeeds if variable X does not occur in complex Term with arity N

termUnify(Term1,Term2)
   Unify the complex terms Term1 and Term2.

========================================================================*/

unify(X,Y):-
   var(X), var(Y), X=Y.
unify(X,Y):-
   var(X), nonvar(Y), notOccursIn(X,Y), X=Y.
unify(X,Y):-
   var(Y), nonvar(X), notOccursIn(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), termUnify(X,Y).
```

```
notOccursIn(X,Term):-
   var(Term), X \== Term.
notOccursIn(_,Term):-
   nonvar(Term), atomic(Term).
notOccursIn(X,Term):-
   nonvar(Term), compound(Term),
   functor(Term,_,Arity), notOccursInComplexTerm(Arity,X,Term).

notOccursInComplexTerm(N,X,Y):-
   N > 0, arg(N,Y,Arg), notOccursIn(X,Arg),
   M is N - 1, notOccursInComplexTerm(M,X,Y).
notOccursInComplexTerm(0,_,_).

termUnify(X,Y):-
   functor(X,Functor,Arity), functor(Y,Functor,Arity),
   unifyArgs(Arity,X,Y).

unifyArgs(N,X,Y):-
   N > 0, M is N - 1,
   arg(N,X,ArgX), arg(N,Y,ArgY),
   unify(ArgX,ArgY), unifyArgs(M,X,Y).
unifyArgs(0,_,_).

/*========================================================================

   Substitution Predicates
   -----------------------

substitute(?Term,?Variable,+Exp,-Result)
   Result is the result of substituting occurrences of Term for each
   free occurrence of Variable in Exp.

========================================================================*/

substitute(Term,Var,Exp,Result):-
   Exp==Var, !, Result=Term.
substitute(_Term,_Var,Exp,Result):-
   \+ compound(Exp), !, Result=Exp.
substitute(Term,Var,Formula,Result):-
   compose(Formula,Functor,[Exp,F]),
   member(Functor,[lambda,forall,exists]), !,
   (
    Exp==Var, !,
    Result=Formula
   ;
    substitute(Term,Var,F,R),
    compose(Result,Functor,[Exp,R])
   ).
substitute(Term,Var,Formula,Result):-
   compose(Formula,Functor,ArgList),
```

```
   substituteList(Term,Var,ArgList,ResultList),
   compose(Result,Functor,ResultList).

substituteList(_Term,_Var,[],[]).
substituteList(Term,Var,[Exp|Others],[Result|ResultOthers]):-
   substitute(Term,Var,Exp,Result),
   substituteList(Term,Var,Others,ResultOthers).

/*========================================================================

   Skolem Function Counter
   -----------------------

funtionCounter(?N)
   N is the current Skolem function index. Declared as dynamic,
   and set to value 1.

newFunctionCounter(?N)
   Unifies N with the current Skolem function index, and increases
   value of the counter.

========================================================================*/

:- dynamic(functionCounter/1).

functionCounter(1).

newFunctionCounter(N):-
   functionCounter(N), M is N+1,
   retract(functionCounter(N)),
   asserta(functionCounter(M)).

/*========================================================================

   Pretty Print Predicates
   -----------------------

========================================================================*/

printRepresentation(Rep):-
   nl, \+ \+ (numbervars(Rep,0,_), write(Rep)), nl.

printReadings(Readings):-
   nl, write('Readings: '), nl, printReading(Readings,0).

printReading([],N):-
   nl, (N=0, write('no readings'); true), nl.
printReading([Reading|OtherReadings],M):-
   N is M + 1, write(N), tab(1),
   \+ \+ (numbervars(Reading,0,_), write(Reading)), nl,
```

```
printReading(OtherReadings,N).
```

```
/*************************************************************************

        name: readLine.pl
     version: March 31, 1998
 description: Converting input line to list of atoms, suitable for
              DCG input.
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(readLine,[readLine/1]).

/*========================================================================

   Read Predicates
   ---------------

readLine(-WordList)
   Outputs a prompt, reads a sequence of characters from the standard
   input and converts this to WordList, a list of strings. Punctuation
   is stripped.

readWords(-WordList)
   Reads in a sequence of characters, until a return is registered,
   and converts this to WordList a list of strings.

readWord(+Char,-Chars,?State)
   Read a word coded as Chars (a list of ascii values), starting
   with with ascii value Char, and determine the State of input
   (`ended' = end of line, `notended' = not end of line).
   Blanks and full stops split words, a return ends a line.

checkWords(+OldWordList,-NewWordList)
   Check if all words are unquoted atoms, if not convert them
   into atoms.

convertWord(+OldWord,-NewWord)
   OldWord and NewWord are words represented as lists of ascii values.
   Converts upper into lower case characters, and eliminates
   non-alphabetic characters.

========================================================================*/

readLine(WordList):-
   nl, write('> '),
   readWords(Words),
   checkWords(Words,WordList).

readWords([Word|Rest]):-
   get0(Char),
```

```
   readWord(Char,Chars,State),
   atom_chars(Word,Chars),
   readRest(Rest,State).

readRest([],ended).
readRest(Rest,notended):-
   readWords(Rest).

readWord(32,[],notended):-!.      %%% blank
readWord(46,[],notended):-!.      %%% full stop
readWord(10,[],ended):-!.         %%% return
readWord(Code,[Code|Rest],State):-
   get0(Char),
   readWord(Char,Rest,State).

checkWords([],[]):- !.
checkWords([''|Rest1],Rest2):-
   checkWords(Rest1,Rest2).
checkWords([Atom|Rest1],[Atom2|Rest2]):-
   atom_chars(Atom,Word1),
   convertWord(Word1,Word2),
   atom_chars(Atom2,Word2),
   checkWords(Rest1,Rest2).

convertWord([],[]):- !.
convertWord([Capital|Rest1],[Small|Rest2]):-
   Capital > 64, Capital < 91, !,
   Small is Capital + 32,
   convertWord(Rest1,Rest2).
convertWord([Weird|Rest1],Rest2):-
   (Weird < 97; Weird > 122), !,
   convertWord(Rest1,Rest2).
convertWord([Char|Rest1],[Char|Rest2]):-
   convertWord(Rest1,Rest2).
```

```
/*************************************************************************

         name: englishLexicon.pl
      version: November 12, 1997; March 9, 1999.
  description: Lexical entries for a small coverage of English
      authors: Patrick Blackburn & Johan Bos

This file contains the lexical entries for a small fragment of
English.  Entries have the form lexicon(Cat,Sym,Phrase,Misc), where
Cat is the syntactic category, Sym the predicate symbol introduced
by the phrase, Phrase a list of the words that form the phrase, and
Misc miscellaneous information depending on the the type of entry.

*************************************************************************/

/*========================================================================
   Determiners: lexicon(det,_,Words,Type)
========================================================================*/

lexicon(det,_,[every],uni).
lexicon(det,_,[a],indef).
lexicon(det,_,[the],def).
lexicon(det,_,[one],card(1)).
lexicon(det,_,[another],alt).
lexicon(det,_,[his],poss(male)).
lexicon(det,_,[her],poss(female)).
lexicon(det,_,[its],poss(nonhuman)).

/*========================================================================
   Nouns: lexicon(noun,Symbol,Words,{[],[Hypernym],Hypernym})
========================================================================*/

lexicon(noun,abstraction,[abstraction],[top]).
lexicon(noun,act,[act],[top]).
lexicon(noun,animal,[animal],[organism]).
lexicon(noun,artifact,[artifact],[object]).
lexicon(noun,beverage,[beverage],[food]).
lexicon(noun,building,[building],[artifact]).
lexicon(noun,container,[container],[instrumentality]).
lexicon(noun,cup,[cup],[container]).
lexicon(noun,device,[device],[instrumentality]).
lexicon(noun,edible,[edible,food],[food]).
lexicon(noun,bkburger,[big,kahuna,burger],[edible]).
lexicon(noun,boxer,[boxer],human).
lexicon(noun,boss,[boss],human).
lexicon(noun,car,[car],[vehicle]).
lexicon(noun,chainsaw,[chainsaw],[device]).
lexicon(noun,criminal,[criminal],human).
lexicon(noun,customer,[customer],human).
lexicon(noun,drug,[drug],[artifact]).
```

```
lexicon(noun,entity,[entity],[top]).
lexicon(noun,episode,[episode],abstraction).
lexicon(noun,female,[female],[human]).
lexicon(noun,fdshake,[five,dollar,shake],[beverage]).
lexicon(noun,food,[food],[object]).
lexicon(noun,footmassage,[foot,massage],[act]).
lexicon(noun,gimp,[gimp],human).
lexicon(noun,glass,[glass],[container]).
lexicon(noun,gun,[gun],[weaponry]).
lexicon(noun,hammer,[hammer],[device]).
lexicon(noun,hashbar,[hash,bar],[building]).
lexicon(noun,human,[human],[organism]).
lexicon(noun,husband,[husband],male).
lexicon(noun,instrumentality,[instrumentality],artifact).
lexicon(noun,joke,[joke],abstraction).
lexicon(noun,man,[man],male).
lexicon(noun,male,[male],[human]).
lexicon(noun,medium,[medium],[instrumentality]).
lexicon(noun,needle,[needle],[device]).
lexicon(noun,object,[object],[entity]).
lexicon(noun,organism,[organism],[entity]).
lexicon(noun,owner,[owner],human).
lexicon(noun,piercing,[piercing],[act]).
lexicon(noun,plant,[plant],[organism]).
lexicon(noun,qpwc,[quarter,pounder,with,cheese],[edible]).
lexicon(noun,radio,[radio],[medium]).
lexicon(noun,restaurant,[restaurant],[building]).
lexicon(noun,robber,[robber],human).
lexicon(noun,suitcase,[suitcase],[container]).
lexicon(noun,shotgun,[shotgun],[weaponry]).
lexicon(noun,sword,[sword],[weaponry]).
lexicon(noun,vehicle,[vehicle],[instrumentality]).
lexicon(noun,weaponry,[weaponry],[instrumentality]).
lexicon(noun,woman,[woman],female).

/*========================================================================
   Proper Names: lexicon(pn,Symbol,Words,{male,female})
========================================================================*/

lexicon(pn,butch,[butch],male).
lexicon(pn,honey_bunny,[honey,bunny],male).
lexicon(pn,jimmy,[jimmy],male).
lexicon(pn,jody,[jody],female).
lexicon(pn,jules,[jules],male).
lexicon(pn,lance,[lance],male).
lexicon(pn,marsellus,[marsellus],male).
lexicon(pn,marsellus,[marsellus,wallace],male).
lexicon(pn,marvin,[marvin],male).
lexicon(pn,mia,[mia],female).
lexicon(pn,mia,[mia,wallace],female).
```

```
lexicon(pn,pumpkin,[pumpkin],male).
lexicon(pn,thewolf,[the,wolf],male).
lexicon(pn,vincent,[vincent],male).
lexicon(pn,vincent,[vincent,vega],male).

/*========================================================================
   Intransitive Verbs: lexicon(iv,Symbol,Words,{fin,inf})
========================================================================*/

lexicon(iv,collapse,[collapses],fin).
lexicon(iv,collapse,[collapse],inf).
lexicon(iv,dance,[dances],fin).
lexicon(iv,dance,[dance],inf).
lexicon(iv,die,[dies],fin).
lexicon(iv,die,[die],inf).
lexicon(iv,growl,[growls],fin).
lexicon(iv,growl,[growl],inf).
lexicon(iv,okay,[is,okay],fin).
lexicon(iv,outoftown,[is,out,of,town],fin).
lexicon(iv,married,[is,married],fin).
lexicon(iv,playairguitar,[plays,air,guitar],fin).
lexicon(iv,playairguitar,[play,air,guitar],inf).
lexicon(iv,smoke,[smokes],fin).
lexicon(iv,smoke,[smoke],inf).
lexicon(iv,snort,[snorts],fin).
lexicon(iv,snort,[snort],inf).
lexicon(iv,shriek,[shrieks],fin).
lexicon(iv,shriek,[shriek],inf).
lexicon(iv,walk,[walks],fin).
lexicon(iv,walk,[walk],inf).

/*========================================================================
   Transitive Verbs: lexicon(tv,Symbol,Words,{fin,inf})
========================================================================*/

lexicon(tv,clean,[cleans],fin).
lexicon(tv,clean,[clean],inf).
lexicon(tv,drink,[drinks],fin).
lexicon(tv,drink,[drink],inf).
lexicon(tv,date,[dates],fin).
lexicon(tv,date,[date],inf).
lexicon(tv,discard,[discards],fin).
lexicon(tv,discard,[discard],inf).
lexicon(tv,eat,[eats],fin).
lexicon(tv,eat,[eat],inf).
lexicon(tv,enjoy,[enjoys],fin).
lexicon(tv,enjoy,[enjoy],inf).
lexicon(tv,hate,[hates],fin).
lexicon(tv,hate,[hate],inf).
lexicon(tv,have,[has],fin).
```

```
lexicon(tv,have,[have],inf).
lexicon(tv,donewith,[is,done,with],fin).
lexicon(tv,kill,[kills],fin).
lexicon(tv,kill,[kill],inf).
lexicon(tv,know,[knows],fin).
lexicon(tv,know,[know],inf).
lexicon(tv,like,[likes],fin).
lexicon(tv,like,[like],inf).
lexicon(tv,love,[loves],fin).
lexicon(tv,love,[love],inf).
lexicon(tv,pickup,[picks,up],fin).
lexicon(tv,pickup,[pick,up],inf).
lexicon(tv,shoot,[shot],fin).
lexicon(tv,shoot,[shoot],inf).
lexicon(tv,tell,[told],fin).
lexicon(tv,tell,[tell],inf).
lexicon(tv,worksfor,[works,for],fin).
lexicon(tv,worksfor,[work,for],inf).

/*========================================================================
   Copula
========================================================================*/

lexicon(cop,'=',[is],fin).

/*========================================================================
   Prepositions: lexicon(prep,Symbol,Words,_)
========================================================================*/

lexicon(prep,in,[in],_).
lexicon(prep,of,[of],_).
lexicon(prep,with,[with],_).

/*========================================================================
   Pronouns: lexicon(pro,Sym,Words,{refl,nonrefl})
========================================================================*/

lexicon(pro,male,[he],nonrefl).
lexicon(pro,female,[she],nonrefl).
lexicon(pro,nonhuman,[it],nonrefl).
lexicon(pro,male,[him],nonrefl).
lexicon(pro,female,[her],nonrefl).
lexicon(pro,male,[himself],refl).
lexicon(pro,female,[herself],refl).
lexicon(pro,nonhuman,[itself],refl).

/*========================================================================
   Relative Pronouns: lexicon(relpro,_,Words,_)
========================================================================*/
```

```
lexicon(relpro,_,[who],_).
lexicon(relpro,_,[that],_).

/*========================================================================
   Coordinations: lexicon(coord,_,Words,{conj,disj})
========================================================================*/

lexicon(coord,_,[and],conj).
lexicon(coord,_,[or],disj).

/*========================================================================
   Discontinious Coordinations: lexicon(dcoord,W1,W2,{conj,cond,disj})
========================================================================*/

lexicon(dcoord,[if],[then],cond).
lexicon(dcoord,[if],[],cond).
lexicon(dcoord,[either],[or],disj).
lexicon(dcoord,[],[or],disj).
lexicon(dcoord,[],[and],conj).
lexicon(dcoord,[],[],conj).

/*========================================================================
   Modifiers: lexicon(mod,_,Words,Type)
========================================================================*/

lexicon(mod,_,[does,not],neg).
lexicon(mod,_,[did,not],neg).
```

```
/*************************************************************************

         name: englishGrammar.pl
      version: November 12, 1997
  description: Grammar rules for a small coverage of English
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
   Grammar Rules
========================================================================*/

d(S) --> s(S).
d((C@S1)@S2)--> dcoord(D,C), s(S1), D, d(S2).

s(NP@VP)--> np2(NP), vp2(VP).

np2(NP)--> np1(NP).
np2((C@NP1)@NP2)--> np1(NP1), coord(C), np1(NP2).

np1(Det@Noun)--> det(Det), n2(Noun).
np1(NP)--> pn(NP).
np1(NP)--> pro(NP).

n2(N)--> n1(N).
n2((C@N1)@N2)--> n1(N1), coord(C), n1(N2).

n1(N)--> noun(N).
n1(PP@N)--> noun(N), pp(PP).
n1(RC@N)--> noun(N), rc(RC).

vp2(VP)--> vp1(VP).
vp2((C@VP1)@VP2)--> vp1(VP1), coord(C), vp1(VP2).

vp1(Mod@VP)--> mod(Mod),  v2(inf,VP).
vp1(VP)--> v2(fin,VP).

v2(fin,Cop@NP)--> cop(Cop), np2(NP).
v2(fin,Neg@(Cop@NP))--> cop(Cop), neg(Neg), np2(NP).

v2(I,V)--> v1(I,V).
v2(I,(C@V1)@V2)--> v1(I,V1), coord(C), v1(I,V2).

v1(I,V)--> iv(I,V).
v1(I,TV@NP)--> tv(I,TV), np2(NP).

pp(Prep@NP)--> prep(Prep), np2(NP).

rc(RP@VP)--> relpro(RP), vp2(VP).
```

```
iv(I,IV)--> {lexicon(iv,Sym,Word,I),ivSem(Sym,IV)}, Word.

tv(I,TV)--> {lexicon(tv,Sym,Word,I),tvSem(Sym,TV)}, Word.

cop(Cop)--> {lexicon(cop,Sym,Word,_),tvSem(Sym,Cop)}, Word.

det(Det)--> {lexicon(det,_,Word,Type),detSem(Type,Det)}, Word.

pn(PN)--> {lexicon(pn,Sym,Word,G),pnSem(Sym,G,PN)}, Word.

pro(Pro)--> {lexicon(pro,Gender,Word,Type),proSem(Gender,Type,Pro)}, Word.

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

relpro(RP)--> {lexicon(relpro,_,Word,_),relproSem(RP)}, Word.

prep(Prep)--> {lexicon(prep,Sym,Word,_),prepSem(Sym,Prep)}, Word.

mod(Mod)--> {lexicon(mod,_,Word,Type),modSem(Type,Mod)}, Word.

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

coord(C)--> {lexicon(coord,_,Word,Type), coordSem(Type,C)}, Word.

dcoord(D,C)--> {lexicon(dcoord,Word,D,Type), dcoordSem(Type,C)}, Word.
```

```
/*************************************************************************

         name: modelDRT.pl (Chapter 7)
      version: July 24, 1997
  description: Model Evaluation for DRSs
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(modelDRT,[evaluate/2]).

:- use_module(comsemOperators),
   use_module(comsemPredicates,[member/2]),
   use_module(exampleModels,[constant/1,example/2]).

/*========================================================================
   Semantic Interpretation
========================================================================*/

satisfyDrs(drs(Dom,C),Model):-
   assignReferents(Dom),
   satisfyConditions(C,Model).

assignReferents([]).
assignReferents([Referent|Others]):-
   constant(Referent),
   assignReferents(Others).

satisfyConditions([],_).
satisfyConditions([Condition|Others],Model):-
   satisfyCondition(Condition,Model),
   satisfyConditions(Others,Model).

satisfyCondition(~ Drs,Model):-
   \+ satisfyDrs(Drs,Model).

satisfyCondition(Drs1 > Drs2,Model):-
   (
    satisfyDrs(Drs1,Model),
    \+ satisfyDrs(Drs2,Model),
    !,
    fail
   ;
    true
   ).

satisfyCondition(Drs1 v Drs2,Model):-
   (
    satisfyDrs(Drs1,Model)
   ;
```

```
    satisfyDrs(Drs2,Model)
   ).

satisfyCondition(X=Y,_Model):-
   X=Y.

satisfyCondition(BasicCond,Model):-
   member(BasicCond,Model).

/*========================================================================
   Evaluation
========================================================================*/

evaluate(Drs,Example):-
   example(Example,Model),
   satisfyDrs(Drs,Model).
```

```
/*************************************************************************

         name: drs2fol.pl (Chapter 7)
      version: June 8, 1998
  description: From DRSs to First-Order Logic
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(drs2fol,[drs2fol/2]).

:- use_module(comsemOperators),
   use_module(comsemPredicates,[compose/3,simpleTerms/1]).

/*========================================================================

   Translation Predicates
   ----------------------

drs2fol(+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.

========================================================================*/

drs2fol(drs([],[Cond]),Formula):-
   cond2fol(Cond,Formula).

drs2fol(drs([],[Cond1,Cond2|Conds]),Formula1 & Formula2):-
   cond2fol(Cond1,Formula1),
   drs2fol(drs([],[Cond2|Conds]),Formula2).

drs2fol(drs([X|Referents],Conds),exists(X,Formula)):-
   drs2fol(drs(Referents,Conds),Formula).

cond2fol(~ Drs, ~ Formula):-
   drs2fol(Drs,Formula).

cond2fol(Drs1 v Drs2, Formula1 v Formula2):-
   drs2fol(Drs1,Formula1),
   drs2fol(Drs2,Formula2).

cond2fol(drs([],Conds) > Drs2, Formula1 > Formula2):-
   drs2fol(drs([],Conds),Formula1),
   drs2fol(Drs2,Formula2).

cond2fol(drs([X|Referents],Conds) > Drs2, forall(X,Formula)):-
   cond2fol(drs(Referents,Conds) > Drs2,Formula).
```

```
cond2fol(BasicCondition,AtomicFormula):-
   AtomicFormula=BasicCondition,
   compose(BasicCondition,_Symbol,Arguments),
   simpleTerms(Arguments).
```

```
/*************************************************************************

         name: threadingDRT.pl (Chapter 8)
      version: June 8, 1998
  description: DRS-threading (Johnson & Klein 1986)
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(threadingDRT,[parse/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,compose/3]).

:- [englishLexicon].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   d(drs([],[])-Drs,Discourse,[]),
   printRepresentation(Drs).

/*========================================================================
   Grammar Rules
========================================================================*/

d(DrsIn-DrsOut)-->
   s(DrsIn-Drs),
   d(Drs-DrsOut).
d(Drs-Drs)-->
   [].

s(DrsIn-DrsOut)-->
   np(X,DrsIn-DrsOut,Scope),
   vp(X,Scope).

np(X,Sem,Scope)-->
   det(X,Sem,Restr,Scope),
   noun(X,Restr).

np(X,DrsIn-DrsOut,Drs-DrsOut)-->
   pn(X,DrsIn-Drs).

vp(X,DrsIn-DrsOut)-->
   iv(X,DrsIn-DrsOut).
```

```
vp(X,DrsIn-DrsOut)-->
   tv(X,Y,Scope),
   np(Y,DrsIn-DrsOut,Scope).

/*========================================================================
   Determiners
========================================================================*/

det(X,DrsIn-DrsOut,RestrIn-RestrOut,ScopeIn-ScopeOut)-->
   {
    lexicon(det,_,Phrase,indef),
    DrsIn = drs(Dom,Conds),
    DrsOut = ScopeOut,
    RestrIn = drs([X|Dom],Conds),
    RestrOut = ScopeIn
   },
   Phrase.

det(X,DrsIn-DrsOut,RestrIn-RestrOut,ScopeIn-ScopeOut)-->
   {
    lexicon(det,_,Phrase,uni),
    DrsIn = drs(Dom,Conds),
    DrsOut = drs(Dom,[RestrOut > ScopeOut|Conds]),
    RestrIn = drs([X],[]),
    ScopeIn = drs([],[])
   },
   Phrase.

/*========================================================================
   Common Nouns
========================================================================*/

noun(X,drs(Dom,Conds)-drs(Dom,[Cond|Conds])) -->
   {
    lexicon(noun,Sym,Phrase,_Type),
    compose(Cond,Sym,[X])
   },
   Phrase.

/*========================================================================
   Proper Names
========================================================================*/

pn(X,drs(Dom,Conds)-drs([X|Dom],[X=Sym|Conds]))-->
   {
    lexicon(pn,Sym,Phrase,_Gender)
   },
   Phrase.

/*========================================================================
```

```
   Intransitive Verbs
==========================================================================*/

iv(X,drs(Dom,Conds)-drs(Dom,[Cond|Conds]))-->
   {
    lexicon(iv,Sym,Phrase,_),
    compose(Cond,Sym,[X])
   },
   Phrase.

/*==========================================================================
   Transitive Verbs
==========================================================================*/

tv(X,Y,drs(Dom,Conds)-drs(Dom,[Cond|Conds]))-->
   {
    lexicon(tv,Sym,Phrase,_),
    compose(Cond,Sym,[Y,X])
   },
   Phrase.
```

```
/*************************************************************************

         name: mainLambdaDRT.pl (Chapter 8)
      version: March 22, 1999
  description: Semantic Construction with Beta Conversion for DRT
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainLambdaDRT,[parse/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,compose/3]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(mergeDRT,[mergeDrs/2]).

:- [englishGrammar], [englishLexicon], [semMacrosLambdaDRT].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   statistics(walltime,_),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,ReducedDrs),
   mergeDrs(ReducedDrs,Drs),
   statistics(walltime,[_,Time]),
   nl, write(Time),
   printRepresentation(Drs).
```

```
/*************************************************************************

        name: semMacroslambdaDRT.pl (Chapter 8)
     version: June 9, 1998
 description: Compositional DRS construction for a fragment of English
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
   Semantic Macros
========================================================================*/

detSem(uni,lambda(P,lambda(Q,drs([],[merge(drs([X],[]),P@X)>(Q@X)])))).
detSem(indef,lambda(P,lambda(Q,merge(merge(drs([X],[]),P@X),Q@X)))).

nounSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

pnSem(Sym,Gender,lambda(P,merge(drs([X],[Cond,X=Sym]),P@X))):-
   compose(Cond,Gender,[X]).

proSem(Gender,Type,lambda(P,alfa(X,Type,Cond,P@X))):-
   compose(Cond,Gender,[X]).

ivSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

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

relproSem(lambda(P1,lambda(P2,lambda(X,merge(P1@X,P2@X))))).

prepSem(Sym,lambda(K,lambda(P,lambda(Y,Drs)))):-
   Drs=merge(K@lambda(X,drs([],[Cond])),P@Y),
   compose(Cond,Sym,[Y,X]).

modSem(neg,lambda(P,lambda(X,drs([],[~(P @ X)])))).

coordSem(conj,lambda(X,lambda(Y,lambda(P,merge(X@P,Y@P))))).
coordSem(disj,lambda(X,lambda(Y,lambda(P,drs([],[(X@P) v (Y@P)]))))).

dcoordSem(cond,lambda(X,lambda(Y,drs([],[X > Y])))).
dcoordSem(conj,lambda(X,lambda(Y,merge(X,Y)))).
dcoordSem(disj,lambda(X,lambda(Y,drs([],[X v Y])))).
```

```
/*************************************************************************

         name: mergeDRT.pl (Chapter 8)
      version: June 9, 1998
  description: Definition of the merge for DRSs
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mergeDRT,[mergeDrs/2]).

:- use_module(comsemOperators),
   use_module(comsemPredicates,[append/3]).

/*========================================================================
   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([],[]).
```

```
/*************************************************************************

         name: mainDRTU.pl (Chapter 8)
      version: May 28, 1998
  description: Plugging Stuff for Underspecified DRSs
       author: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainDRTU,[parse/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(betaConversion,[betaConvert/2]),
   use_module(pluggingAlgorithm,[plugHole/4]),
   use_module(mergeDRT,[mergeDrs/2]),
   use_module(mergeUSR,[mergeUSR/2]),
   use_module(comsemPredicates,[append/3,printRepresentation/1,
                                printReadings/1,compose/3]).

:- [englishGrammar], [englishLexicon], [semMacrosDRTU].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Sentence),
   d(Sem,Sentence,[]),
   betaConvert(merge(usr([Top,Main],[],[]),Sem@Top@Main),Reduced),
   mergeUSR(Reduced,usr(D,L,C)),
   printRepresentation(usr(D,L,C)),
   findall(Drs,(plugHole(Top,L-[],C,[]),mergeDrs(Top,Drs)),DRSs),
   printReadings(DRSs).
```

```
/*************************************************************************

         name: semMacrosDRTU.pl (Chapter 8)
      version: May 28, 1998
  description: Semantic Macros for DRT Unplugged
       author: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
   Semantic Macros
========================================================================*/

detSem(uni,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([F,R,S],
                       [F:drs([],[merge(drs([X],[]),R)>S])],
                       [leq(F,H),leq(L,S)]),P@X@H@R),Q@X@H@L)))))).

detSem(indef,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([E,R,S],
                       [E:merge(merge(drs([X],[]),R),S)],
                       [leq(E,H),leq(L,S)]),P@X@H@R),Q@X@H@L)))))).

nounSem(Sym,lambda(X,lambda(_,lambda(L,usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

pnSem(Sym,_,lambda(P,lambda(H,lambda(M,
      merge(usr([L],[M:merge(drs([X],[X=Sym]),L)],[leq(M,H)]),
            P@X@H@L))))).

proSem(_,_,lambda(P,lambda(H,lambda(M,
       merge(usr([L],[M:merge(drs([X],[]),L)],[leq(M,H)]),
             P@X@H@L))))).

ivSem(Sym,lambda(X,lambda(_,lambda(L,
      usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

tvSem(Sym,lambda(K,lambda(Y,K@lambda(X,lambda(_,lambda(L,
      usr([],[L:drs([],[Cond])],[]))))))):-
   compose(Cond,Sym,[Y,X]).

relproSem(lambda(P,lambda(Q,lambda(X,lambda(H,lambda(L,
          merge(usr([L2,L3,H1] ,
                    [L:merge(L3,H1)],
                    [leq(L2,H1)]),
                merge(P@X@H@L2,Q@X@H@L3)))))))).

prepSem(Sym,lambda(K,lambda(P,lambda(Y,lambda(H,lambda(L3,
        merge(K@lambda(X,lambda(H,lambda(L1,
```

```
            usr([L2,H1],
                [L3:merge(L2,H1),L1:drs([],[Cond])],
                [leq(L1,H1)])))))@H@L1,P@Y@H@L2))))))):-
   compose(Cond,Sym,[Y,X]).

modSem(neg,lambda(P,lambda(X,lambda(H,lambda(L,
       merge(usr([N,S],[N:drs([],[~S])],[leq(N,H),leq(L,S)]),
             P@X@H@L)))))).

coordSem(conj,lambda(X,lambda(Y,lambda(P,lambda(H,lambda(L,
         merge(usr([L1,L2],[L:merge(L1,L2)],[leq(L,H)]),
               merge(X@P@H@L1,Y@P@H@L2)))))))).

coordSem(disj,lambda(X,lambda(Y,lambda(P,lambda(H,lambda(L,
         merge(usr([L1,L2],[L:drs([],[L1 v L2])],[leq(L,H)]),
               merge(X@P@H@L1,Y@P@H@L2)))))))).

dcoordSem(cond,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:drs([],[H1 > H2])],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).

dcoordSem(conj,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:merge(H1,H2)],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).

dcoordSem(disj,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:drs([],[H1 v H2])],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).
```

```
/*************************************************************************

         name: mainPronounsDRT.pl (Chapter 9)
      version: Feb 3, 1999
  description: Pronoun Resolution
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainPronounsDRT,[parse/0,resolveDrs/1]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,simpleTerms/1,
                                compose/3,append/3,member/2]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(bindingDRT,[potentialAntecedent/3,properBinding/3]).

:- [englishGrammar], [englishLexicon], [semMacrosLambdaDRT].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,Drs),
   resolveDrs([Drs]-[ResolvedDrs]),
   printRepresentation(ResolvedDrs).

/*=====================================================================
     Pronoun Resolution
=====================================================================*/

resolveDrs([merge(B1,B2)|A1]-[drs(D,C)|A3]):-
        resolveDrs([B1|A1]-A2),
        resolveDrs([B2|A2]-[drs(D1,C1),drs(D2,C2)|A3]),
        append(D1,D2,D),
        append(C1,C2,C).

resolveDrs([alfa(Referent,Type,Gender,B1)|A1]-A2):-
        potentialAntecedent(A1,Referent,Gender),
        properBinding(Type,Referent,B1),
        resolveDrs([B1|A1]-A2).

resolveDrs([drs(D1,C1)|A1]-A2):-
        resolveConds(C1,[drs(D1,[])|A1]-A2).

resolveConds([~B1|Conds],A1-A3):-
```

```
        resolveDrs([B1|A1]-[B2,drs(D,C)|A2]),
        resolveConds(Conds,[drs(D,[~B2|C])|A2]-A3).

resolveConds([B1 > B2|Conds],A1-A4):-
        resolveDrs([B1|A1]-A2),
        resolveDrs([B2|A2]-[B4,B3,drs(D,C)|A3]),
        resolveConds(Conds,[drs(D,[B3 > B4|C])|A3]-A4).

resolveConds([B1 v B2|Conds],A1-A4):-
        resolveDrs([B1|A1]-[B3|A2]),
        resolveDrs([B2|A2]-[B4,drs(D,C)|A3]),
        resolveConds(Conds,[drs(D,[B3 v B4|C])|A3]-A4).

resolveConds([Basic|Conds],[drs(D,C)|A1]-A2):-
        compose(Basic,_Symbol,Arguments),
        simpleTerms(Arguments),
        resolveConds(Conds,[drs(D,[Basic|C])|A1]-A2).

resolveConds([],A-A).
```

```
/*************************************************************************

         name: bindingDRT.pl (Chapter 9)
      version: Feb 3, 1999
  description: Check Binding Constraints
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(bindingDRT,[potentialAntecedent/3,
                      properBinding/3]).

:- use_module(comsemOperators),
   use_module(semOntology,[consistent/2]),
   use_module(comsemPredicates,[compose/3,member/2]).


/*========================================================================
     Potential Antecedent (Ordinary DRSs)
========================================================================*/

potentialAntecedent(A,X,Gender):-
   member(drs(Dom,Conds),A),
   member(X,Dom),
   compose(Gender,Symbol1,_),
   \+ (
          member(Cond,Conds),
          compose(Cond,Symbol2,[Y]),
          Y==X,
          \+ consistent(Symbol1,Symbol2)
      ).

/*========================================================================
     Potential Antecedent (Focus DRSs)
========================================================================*/

potentialAntecedent(A,X,Gender):-
   member(drs(Dom,_,_,Conds),A),
   member(X,Dom),
   compose(Gender,Symbol1,_),
   \+ (
          member(Cond,Conds),
          compose(Cond,Symbol2,[Y]),
          Y==X,
          \+ consistent(Symbol1,Symbol2)
      ).


/*========================================================================
   Check Binding violation.
```

```
==========================================================================*/

properBinding(Type,X,Drs):-
        Type=refl,
        reflexiveBinding(X,Drs).

properBinding(Type,X,Drs):-
        \+ Type=refl,
        (
            reflexiveBinding(X,Drs),
            !, fail
        ;
            true
        ).

reflexiveBinding(_,[]):- fail.
reflexiveBinding(_,alfa(_,_,_,_)):- fail.
reflexiveBinding(_,merge(_,_)):- fail.
reflexiveBinding(_,~_):- !, fail.
reflexiveBinding(X,drs(_,Conds)):-
   reflexiveBinding(X,Conds).

reflexiveBinding(X,[Basic|Conds]):- !,
        (
            compose(Basic,_Sym,[Subj,Obj]),
            Subj==Obj,
            X==Obj, !
        ;
            reflexiveBinding(X,Conds)
        ).
```

```
/*************************************************************************

        name: mainFocusDRT.pl (Chapter 9)
     version: Feb 3, 1999
 description: Pronoun Resolution with the Focusing Algorithm
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainFocusDRT,[parse/0,resolveDrs/2]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,simpleTerms/1,
                                compose/3,append/3,member/2]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(bindingDRT,[potentialAntecedent/3,properBinding/3]).

:- [englishGrammar], [englishLexicon], [semMacrosLambdaDRT].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   parse(drs([],[],[],[])).

parse(Old):-
   readLine(Discourse),
   d(SDrs,Discourse,[]),
   betaConvert(SDrs,Drs),
   resolveDrs([merge(Old,Drs)]-A,[]-Anaphora),
   updateFocus(Anaphora,A,[New]),
   printRepresentation(New), !,
   parse(New).

/*=====================================================================
    Pronoun Resolution: DRSs
=====================================================================*/

resolveDrs([merge(B1,B2)|A1]-[drs(D,AF,DF,C)|A5],R1-R3):-
   resolveDrs([B1|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3),
   resolveDrs([B2|A3]-A4,R2-R3),
   updateFocus(R3,A4,[drs(D1,AF1,DF1,C1),drs(D2,AF2,DF2,C2)|A5]),
   append(D1,D2,D),
   append(AF1,AF2,AF),
   append(DF1,DF2,DF),
   append(C1,C2,C).
```

```
resolveDrs([drs(D,AF,DF,C)|A1]-A3,R1-R2):-
   resolveConds(C,[drs(D,AF,DF,[])|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3).

resolveDrs([drs(D,C)|A1]-A3,R1-R2):-
   resolveConds(C,[drs(D,[],[],[])|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3).

/*========================================================================
    Pronoun Resolution: if pronoun appears as agent, then choose the
    actor focus as preferred antecedent
========================================================================*/

resolveDrs([alfa(X,Type,Gender,B1)|A1]-A2,R1-[Type:X|R2]):-
   role(agent,X,B1), !,
   (
    currentActorFocus(A1,AF),
    potentialAntecedent(A1,X,Gender),
    X == AF,
    properBinding(Type,X,B1)
   ;
    currentDiscourseFocus(A1,DF),
    potentialAntecedent(A1,X,Gender),
    X == DF,
    properBinding(Type,X,B1)
   ;
    potentialAntecedent(A1,X,Gender),
    properBinding(Type,X,B1)
   ),
   resolveDrs([B1|A1]-A2,R1-R2).


/*========================================================================
    Pronoun Resolution: if pronoun appears as non-agent, then choose
    the discourse focus as preferred antecedent
========================================================================*/

resolveDrs([alfa(X,Type,Gender,B1)|A1]-A2,R1-[Type:X|R2]):-
   (
    currentDiscourseFocus(A1,DF),
    potentialAntecedent(A1,X,Gender),
    X == DF,
    properBinding(Type,X,B1)
   ;
    currentActorFocus(A1,AF),
    potentialAntecedent(A1,X,Gender),
    X == AF,
    properBinding(Type,X,B1)
   ;
    potentialAntecedent(A1,X,Gender),
```

```
    properBinding(Type,X,B1)
   ),
   resolveDrs([B1|A1]-A2,R1-R2).


/*========================================================================
    Pronoun Resolution: DRS-conditions
========================================================================*/

resolveConds([~B1|Conds],A1-A3,R1-R3):-
   resolveDrs([B1|A1]-[B2,drs(D,AF,DF,C)|A2],R1-R2),
   resolveConds(Conds,[drs(D,AF,DF,[~B2|C])|A2]-A3,R2-R3).

resolveConds([B1 > B2|Conds],A1-A5,R1-R4):-
   resolveDrs([B1|A1]-A2,R1-R2),
   updateFocus(R2,A2,A3),
   resolveDrs([B2|A3]-[B4,B3,drs(D,AF,DF,C)|A4],R2-R3),
   resolveConds(Conds,[drs(D,AF,DF,[B3 > B4|C])|A4]-A5,R3-R4).

resolveConds([B1 v B2|Conds],A1-A4,R1-R4):-
   resolveDrs([B1|A1]-[B3|A2],R1-R2),
   resolveDrs([B2|A2]-[B4,drs(D,AF,DF,C)|A3],R2-R3),
   resolveConds(Conds,[drs(D,AF,DF,[B3 v B4|C])|A3]-A4,R3-R4).

resolveConds([Basic|Conds],[drs(D,AF,DF,C)|A1]-A2,R1-R2):-
   compose(Basic,_Symbol,Arguments),
   simpleTerms(Arguments),
   resolveConds(Conds,[drs(D,AF,DF,[Basic|C])|A1]-A2,R1-R2).

resolveConds([],A-A,R-R).


/*========================================================================
     Focus Update
========================================================================*/

updateFocus(Anaphora,DRSs1,Updated):-
   updateActorFocus(DRSs1,DRSs2),
   updateDiscourseFocus(Anaphora,DRSs2,Updated).

/*========================================================================
     Update Actor Focus
========================================================================*/

updateActorFocus([drs(D,AF,DF,C)|A],Updated):-
   expectedActorFocus(drs(D,C),Focus), !,
   addActorFocus(Focus,[drs(D,AF,DF,C)|A],Updated).

updateActorFocus(DRSs,DRSs).
```

```
/*========================================================================
     Update Discourse Focus
========================================================================*/

updateDiscourseFocus(Anaphora,DRSs,Updated):-
   currentDiscourseFocus(DRSs,Current),
   member(_:A,Anaphora), A==Current, !,
   Updated=DRSs.

updateDiscourseFocus(_Anaphora,[drs(D,AF,DF,C)|A],Updated):-
   expectedDiscourseFocus(drs(D,C),Focus), !,
   addDiscourseFocus(Focus,[drs(D,AF,DF,C)|A],Updated).

updateDiscourseFocus(_,DRSs,DRSs).


/*========================================================================
     Add new focus to DRS
========================================================================*/


addActorFocus(Focus,DRSs,Updated):-
   (
       currentActorFocus(DRSs,Current),
       Current==Focus, !,
       Updated=DRSs
   ;
       DRSs=[drs(D,AF,DF,C)|A],
       Updated=[drs(D,[Focus|AF],DF,C)|A]
   ).

addDiscourseFocus(Focus,DRSs,Updated):-
   (
       currentDiscourseFocus(DRSs,Current),
       Current==Focus, !,
       Updated=DRSs
   ;
       DRSs=[drs(D,AF,DF,C)|A],
       Updated=[drs(D,AF,[Focus|DF],C)|A]
   ).


/*========================================================================
     Current Foci
========================================================================*/

currentDiscourseFocus(DRSs,Focus):-
   member(drs(_,_,[Focus|_],_),DRSs), !.

currentActorFocus(DRSs,Focus):-
```

```
   member(drs(_,[Focus|_],_,_),DRSs), !.

/*========================================================================
     Expected Focus
========================================================================*/

expectedActorFocus(drs(Dom,Conds),Focus):-
   member(Focus,Dom),
   role(agent,Focus,Conds).

expectedDiscourseFocus(drs(Dom,Conds),Focus):-
   member(Focus,Dom),
   role(theme,Focus,Conds).


/*========================================================================
     Thematic Roles
========================================================================*/

role(Role,X,drs(_,_,_,Conds)):-
   role(Role,X,Conds).

role(Role,X,drs(_,Conds)):-
   role(Role,X,Conds).

role(Role,X,merge(B1,B2)):-
   role(Role,X,B1);
   role(Role,X,B2).

role(Role,X,alfa(_,_,_,B)):-
   role(Role,X,B).

role(Role,X,[~ B|_]):-
   role(Role,X,B).

role(Role,X,[B1 > B2|_]):-
   role(Role,X,B1);
   role(Role,X,B2).

role(Role,X,[B1 v B2|_]):-
   role(Role,X,B1);
   role(Role,X,B2).

role(Role,X,[_|Conds]):-
   role(Role,X,Conds).

role(agent,X,[C|_]):-
   agent(X,C).

role(theme,X,[C|_]):-
```

```
   theme(X,C).

/*======================================================================
     Agent Position
======================================================================*/

agent(X,C):-
   compose(C,Sym,[Y,_]),
   X==Y,
   lexicon(tv,Sym,_,inf).


/*======================================================================
     Theme Position
======================================================================*/

theme(X,C):-
   compose(C,Sym,[Y]),
   X==Y,
   lexicon(iv,Sym,_,inf).

theme(X,C):-
   compose(C,Sym,[_,Y]),
   X==Y,
   lexicon(tv,Sym,_,inf).

theme(X,C):-
   compose(C,'=',[Y,Z]),
   var(Z),
   X==Y.
```

```
/*************************************************************************

         name: mainPresupDRT.pl
      version: May 5, 1998
  description: Presupposition Projection
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainPresupDRT,[parse/0,projectDrs/1]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printReadings/1,compose/3]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(resolvePresup,[projectDrs/1,accommodate/2]).

:- [englishGrammar], [englishLexicon], [semMacrosPresupDRT].


/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   d(Drs1,Discourse,[]),
   betaConvert(Drs1,Drs2),
   findall(Drs4,(
                 projectDrs([Drs2,pre([])]-[Drs3,pre(P)]),
                 accommodate(P,Drs3-Drs4)
                ),
           Readings),
   printReadings(Readings).
```

```
/*************************************************************************

         name: resolvePresup.pl
      version: May 5, 1998
  description: Presupposition Resolution
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(resolvePresup,[projectDrs/1,accommodate/2]).

:- use_module(comsemOperators),
   use_module(comsemPredicates,[simpleTerms/1,compose/3,
                                append/3,select/3]),
   use_module(bindingDRT,[properBinding/3]),
   use_module(matchDRT,[matchDrs/4]).

/*========================================================================
   Presupposition Projection for DRSs
========================================================================*/

projectDrs([merge(B1,B2)|A1]-[drs(D3,C3)|A3]):-
        projectDrs([B1|A1]-A2),
        projectDrs([B2|A2]-[drs(D1,C1),drs(D2,C2)|A3]),
        append(D1,D2,D3),
        append(C1,C2,C3).

projectDrs([alfa(X,Type,B1,B3)|A1]-A4):-
        projectDrs([B1|A1]-[B2|A2]),
        resolveAlfa(X,B2,A2-A3),
        properBinding(Type,X,B3),
        projectDrs([B3|A3]-A4).

projectDrs([drs(D1,C1)|A1]-A2):-
        projectConds(C1,[drs(D1,[])|A1]-A2).

/*========================================================================
   Presupposition Projection for DRS-Conditions
========================================================================*/

projectConds([~B1|Conds],A1-A3):-
        projectDrs([B1,pre([])|A1]-[B2,pre(Pres),drs(D,C)|A2]),
        accommodate(Pres,B2-B3),
        projectConds(Conds,[drs(D,[~B3|C])|A2]-A3).

projectConds([B1 > B2|Conds],A1-A4):-
        projectDrs([B1,pre([])|A1]-A2),
        projectDrs([B2,pre([])|A2]-[B4,pre(P2),B3,pre(P1),drs(D,C)|A3]),
        accommodate(P1,B3-B5),
        accommodate(P2,B4-B6),
```

```
        projectConds(Conds,[drs(D,[B5 > B6|C])|A3]-A4).

projectConds([B1 v B2|Conds],A1-A4):-
        projectDrs([B1,pre([])|A1]-[B3,pre(P1)|A2]),
        projectDrs([B2,pre([])|A2]-[B4,pre(P2),drs(D,C)|A3]),
        accommodate(P1,B3-B5),
        accommodate(P2,B4-B6),
        projectConds(Conds,[drs(D,[B5 v B6|C])|A3]-A4).

projectConds([Basic|Conds],[drs(D,C)|A1]-A2):-
        compose(Basic,_Symbol,Arguments),
        simpleTerms(Arguments),
        projectConds(Conds,[drs(D,[Basic|C])|A1]-A2).

projectConds([],A-A).

/*========================================================================
   Resolving Alfa-DRSs
========================================================================*/

resolveAlfa(X,AlfaDrs,[drs(D,C)|Others]-[New|Others]):-
        matchDrs(X,AlfaDrs,drs(D,C),New).

resolveAlfa(_,AlfaDrs,[pre(A)|Others]-[pre([AlfaDrs|A])|Others]).

resolveAlfa(X,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others]):-
        select(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New).

resolveAlfa(X,AlfaDrs,[AnteDrs|Others]-[AnteDrs|NewOthers]):-
        resolveAlfa(X,AlfaDrs,Others-NewOthers).

/*=====================================================================
     Accommodation
=====================================================================*/

accommodate([],B-B).

accommodate([drs(D1,C1)|Presups],drs(D2,C2)-B):-
        append(D1,D2,D3),
        append(C1,C2,C3),
        accommodate(Presups,drs(D3,C3)-B).
```

```
/*************************************************************************

        name: matchDRT.pl
     version: Feb 15, 1999
 description: Matching for DRSs, including consistency check
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(matchDRT,[matchDrs/4]).

:- use_module(comsemPredicates,[member/2,compose/3]),
   use_module(semOntology,[consistent/2]).

/*========================================================================
   Partial Match
========================================================================*/

matchDrs(X,drs(D1,C1),drs(D2,C2),drs(D3,C3)):-
        member(X,D2),
        mergeLists(D1,D2,D3),
        mergeLists(C1,C2,C3),
        consistentConditions(X,C3).

consistentConditions(X,Conds):-
   \+ (
          member(Cond1,Conds),
          member(Cond2,Conds), \+ Cond1=Cond2,
          compose(Cond1,Symbol1,[Y]), Y==X,
          compose(Cond2,Symbol2,[Z]), Z==X,
          \+ consistent(Symbol1,Symbol2)
      ).

/*========================================================================
   Merging of Lists
========================================================================*/

mergeLists([],L,L).

mergeLists([X|R],L1,L2):-
        member(Y,L1),
        X==Y, !,
        mergeLists(R,L1,L2).

mergeLists([X|R],L1,[X|L2]):-
        mergeLists(R,L1,L2).
```

```
/*************************************************************************

         name: mainPresupScoreDRT.pl
      version: Feb 6, 1999
  description: Presupposition Resolution with Score Calculation
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainPresupScoreDRT,[parse/0,projectDrs/1]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printReadings/1,compose/3]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(resolvePresupScore,[projectDrs/2,accommodate/2]).

:- [englishGrammar], [englishLexicon], [semMacrosPresupDRT].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   d(Drs1,Discourse,[]),
   betaConvert(Drs1,Drs2),
   findall((Score:Drs4),(
                 projectDrs([Drs2,pre([])]-[Drs3,pre(P)],1-Score),
                 accommodate(P,Drs3-Drs4)
                ),
           Readings),
   printReadings(Readings).
```

```
/*************************************************************************

        name: resolvePresupScore.pl
     version: Feb 6, 1999
 description: Presupposition Projection with Score Calculation
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(resolvePresupScore,[projectDrs/2,accommodate/2]).

:- use_module(comsemOperators),
   use_module(comsemPredicates,[simpleTerms/1,compose/3,
                                append/3,select/3]),
   use_module(bindingDRT,[properBinding/3]),
   use_module(matchDRT,[matchDrs/4]).


/*========================================================================
   Presupposition Projection for DRSs
========================================================================*/

projectDrs([merge(B1,B2)|A1]-[drs(D3,C3)|A3],S1-S3):-
        projectDrs([B1|A1]-A2,S1-S2),
        projectDrs([B2|A2]-[drs(D1,C1),drs(D2,C2)|A3],S2-S3),
        append(D1,D2,D3),
        append(C1,C2,C3).

projectDrs([alfa(X,Type,B1,B3)|A1]-A4,S1-S4):-
        projectDrs([B1|A1]-[B2|A2],S1-S2),
        resolveAlfa(X,Type,B2,A2-A3,S2-S3),
        properBinding(Type,X,B3),
        projectDrs([B3|A3]-A4,S3-S4).

projectDrs([drs(D1,C1)|A1]-A2,S1-S2):-
        projectConds(C1,[drs(D1,[])|A1]-A2,S1-S2).


/*========================================================================
   Presupposition Projection for DRS-Conditions
========================================================================*/

projectConds([~B1|Conds],A1-A3,S1-S3):-
        projectDrs([B1,pre([])|A1]-[B2,pre(Pres),drs(D,C)|A2],S1-S2),
        accommodate(Pres,B2-B3),
        projectConds(Conds,[drs(D,[~B3|C])|A2]-A3,S2-S3).

projectConds([B1 > B2|Conds],A1-A4,S1-S4):-
        projectDrs([B1,pre([])|A1]-A2,S1-S2),
        projectDrs([B2,pre([])|A2]-[B4,pre(P2),B3,pre(P1),drs(D,C)|A3],S2-S3),
```

```
        accommodate(P1,B3-B5),
        accommodate(P2,B4-B6),
        projectConds(Conds,[drs(D,[B5 > B6|C])|A3]-A4,S3-S4).

projectConds([B1 v B2|Conds],A1-A4,S1-S4):-
        projectDrs([B1,pre([])|A1]-[B3,pre(P1)|A2],S1-S2),
        projectDrs([B2,pre([])|A2]-[B4,pre(P2),drs(D,C)|A3],S2-S3),
        accommodate(P1,B3-B5),
        accommodate(P2,B4-B6),
        projectConds(Conds,[drs(D,[B5 v B6|C])|A3]-A4,S3-S4).

projectConds([Basic|Conds],[drs(D,C)|A1]-A2,S1-S2):-
        compose(Basic,_Symbol,Arguments),
        simpleTerms(Arguments),
        projectConds(Conds,[drs(D,[Basic|C])|A1]-A2,S1-S2).

projectConds([],A-A,S-S).


/*========================================================================
   Resolving Alfa-DRSs
========================================================================*/

resolveAlfa(X,Type,AlfaDrs,[drs(D,C)|Others]-[New|Others],S1-S2):-
        global(Others),
        matchDrs(X,AlfaDrs,drs(D,C),New),
        (Type=refl, S2 = S1;
         Type=nonrefl, S2 = S1;
         Type=def, S2 = S1;
         Type=name, S2 = S1).

resolveAlfa(X,Type,AlfaDrs,[drs(D,C)|Others]-[New|Others],S1-S2):-
        nonglobal(Others),
        matchDrs(X,AlfaDrs,drs(D,C),New),
        (Type=refl, S2 is S1 * 1;
         Type=nonrefl, S2 is S1 * 1;
         Type=def, S2 is S1 * 0.5;
         Type=name, S2 is S1 * 0.2).

resolveAlfa(_,Type,Alfa,[pre(A)]-[pre([Alfa|A])],S1-S2):-
        global([pre(A)]),
        (Type=nonrefl, S2 is S1 * 0.5;
         Type=def, S2 is S1 * 0.9;
         Type=name, S2 is S1 * 0.9).

resolveAlfa(_,Type,Alfa,[pre(A)|Others]-[pre([Alfa|A])|Others],S1-S2):-
        nonglobal([pre(A)|Others]),
        (
        %Type=nonrefl, S2 is S1 * 0.1;
         Type=def, S2 is S1 * 0.7;
```

```
         Type=name, S2 is S1 * 0.2).

resolveAlfa(X,Type,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others],S1-S2):-
        global([pre(A1)|Others]),
        select(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New),
        (Type=refl, S2 = S1;
         Type=nonrefl, S2 = S1;
         Type=def, S2 = S1;
         Type=name, S2 = S1).

resolveAlfa(X,Type,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others],S1-S2):-
        nonglobal([pre(A1)|Others]),
        select(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New),
        (Type=refl, S2 is S1 * 1;
         Type=nonrefl, S2 is S1 * 1;
         Type=def, S2 is S1 * 0.5;
         Type=name, S2 is S1 * 0.2).

resolveAlfa(X,Type,AlfaDrs,[AnteDrs|Others]-[AnteDrs|NewOthers],S1-S2):-
        resolveAlfa(X,Type,AlfaDrs,Others-NewOthers,S1-S2).

/*========================================================================
   Global or non-global position with respect to stack
========================================================================*/

global([pre(_)]).
global([drs(_,_)|Stack]):- global(Stack).

nonglobal([pre(_),drs(_,_)|_]).
nonglobal([drs(_,_)|Stack]):- nonglobal(Stack).

/*=====================================================================
     Accommodation
=====================================================================*/

accommodate([],B-B).

accommodate([drs(D1,C1)|Presups],drs(D2,C2)-B):-
        append(D1,D2,D3),
        append(C1,C2,C3),
        accommodate(Presups,drs(D3,C3)-B).
```

```
/*************************************************************************

         name: semMacrosPresupDRT.pl
      version: May 5, 1998
  description: Semantic Macros for Presupposition Projection in DRT
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
   Semantic Macros
========================================================================*/

detSem(uni,lambda(P,lambda(Q,drs([],[merge(drs([X],[]),P@X)>(Q@X)])))).

detSem(indef,lambda(P,lambda(Q,merge(merge(drs([X],[]),P@X),Q@X)))).

detSem(def,lambda(P,lambda(Q,alfa(X,def,merge(drs([X],[]),P@X),Q@X)))).

detSem(card(1),lambda(P,lambda(Q,
       merge(drs([X],[merge(drs([Y],[]),merge(P@Y,Q@Y))>drs([],[X=Y])]),
             merge(P@X,Q@X))))).

detSem(poss(Gender),lambda(P,lambda(Q,
       alfa(Y,def,alfa(X,nonrefl,drs([X],[Basic]),
                       merge(drs([Y],[of(Y,X)]),P@Y)),Q@Y)))):-
   compose(Basic,Gender,[X]).

nounSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

pnSem(Sym,Gender,lambda(P,alfa(X,name,drs([X],[X=Sym,Cond]),P@X))):-
   compose(Cond,Gender,[X]).

proSem(Gender,Type,lambda(P,alfa(X,Type,drs([X],[Cond]),P@X))):-
   compose(Cond,Gender,[X]).

ivSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

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

relproSem(lambda(P,lambda(Q,lambda(X,merge(P@X,Q@X))))).

prepSem(Sym,lambda(K,lambda(P,lambda(Y,Drs)))):-
   Drs=merge(K@lambda(X,drs([],[Cond])),P@Y),
   compose(Cond,Sym,[Y,X]).

modSem(neg,lambda(P,lambda(X,drs([],[~(P @ X)])))).
```

```
coordSem(conj,lambda(X,lambda(Y,lambda(P,merge(X@P,Y@P))))).
coordSem(disj,lambda(X,lambda(Y,lambda(P,drs([],[(X@P) v (Y@P)]))))).

dcoordSem(cond,lambda(X,lambda(Y,drs([],[X > Y])))).
dcoordSem(conj,lambda(X,lambda(Y,merge(X,Y)))).
dcoordSem(disj,lambda(X,lambda(Y,drs([],[X v Y])))).
```

```
/*************************************************************************

         name: curt.pl (Chapter 11)
      version: June 18, 1998
  description: User-System discourse interaction
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(curt,[curt/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printReadings/1,member/2,compose/3]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(resolvePresup,[projectDrs/1,accommodate/2]),
   use_module(acceptabilityConstraints,[consistentReadings/2,
                                         informativeReadings/2,
                                         localConsistentReadings/2,
                                         localInformativeReadings/2]).

:- [englishGrammar], [englishLexicon], [semMacrosPresupDRT].


/*========================================================================
   Start Curt
========================================================================*/

curt:-
   curtInput([drs([],[])]).

/*========================================================================
   System requests user
========================================================================*/

curtInput(Readings):-
   readLine(Input),
   curtOutput(Input,Readings).

/*========================================================================
   Systems responds to user
========================================================================*/

curtOutput([],Readings):- !,
   curtSays('Want to tell me something?'),
   curtInput(Readings).

curtOutput([bye],_):- !,
   curtSays('Bye bye!').
```

```
curtOutput([new],_):- !,
   curt.

curtOutput([help],Readings):- !,
   nl, write('bye - no more talking'),
   nl, write('drs - prints current readings in DRS-format'),
   nl, write('new - starts a new discourse'), nl,
   curtInput(Readings).

curtOutput([drs],Readings):- !,
   printReadings(Readings),
   curtInput(Readings).

curtOutput(Input,Readings):-
   d(MergeDrs,Input,[]),!,
   betaConvert(MergeDrs,ReducedDrs),
   getReadings(ReducedDrs,Readings,PotentialReadings),
   consistentReadings(PotentialReadings,ConsistentReadings),
   (
       ConsistentReadings=[],
       curtSays('No! I do not believe that!'),!,
       curtInput(Readings)
   ;
       informativeReadings(ConsistentReadings,InformativeReadings),
       (
           InformativeReadings=[],
           curtSays('Yes, I knew that!'),!,
           curtInput(Readings)
       ;
           curtSays('Ok.'),
           localInformativeReadings(InformativeReadings,LocalInformative),
           localConsistentReadings(LocalInformative,LocalConsistent),
           (
               LocalConsistent=[], !,
               SelectedReadings=InformativeReadings
           ;
               SelectedReadings=LocalConsistent
           ),
           curtInput(SelectedReadings)
       )
   ).

curtOutput(_,Readings):-
   curtSays('What?'),
   curtInput(Readings).

/*========================================================================
   Compute all readings
========================================================================*/
```

```
getReadings(ProtoDrs,Readings,PotentialReadings):-
   findall((OldDrs,NewDrs),
           (
            member(OldDrs,Readings),
            projectDrs([merge(OldDrs,ProtoDrs),pre([])]-[Drs,pre(P)]),
            accommodate(P,Drs-NewDrs)
           ),
           PotentialReadings).

/*=========================================================================
   Curt's output
=========================================================================*/

curtSays(X):- nl, write(X), nl.
```

```
/*************************************************************************

         name: acceptabilityConstraints.pl (Chapter 11)
      version: June 18, 1998
  description: Consistency and Informativity
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(acceptabilityConstraints,[consistentReadings/2,
                                    informativeReadings/2,
                                    localConsistentReadings/2,
                                    localInformativeReadings/2]).

:- use_module(drs2fol,[drs2fol/2]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[append/3]),
   use_module(mergeDRT,[mergeDrs/2]),
   use_module(callTheoremProver,[callTheoremProver/3,
                                 callModelBuilder/4]),
   use_module(semOntology,[generateOntology/1]).

/*========================================================================
   Select Readings
========================================================================*/

informativeReadings([],[]).
informativeReadings([(DrsSoFar,Drs)|Potential],[Drs|Selected]):-
   informative(DrsSoFar,Drs), !,
   informativeReadings(Potential,Selected).
informativeReadings([_|Potential],Selected):-
   informativeReadings(Potential,Selected).

consistentReadings([],[]).
consistentReadings([(Old,Drs)|Potential],[(Old,Drs)|Selected]):-
   consistent(Drs), !,
   consistentReadings(Potential,Selected).
consistentReadings([_|Potential],Selected):-
   consistentReadings(Potential,Selected).

localConsistentReadings([],[]).
localConsistentReadings([Drs|Potential],[Drs|Selected]):-
   localConsistent(Drs), !,
   localConsistentReadings(Potential,Selected).
localConsistentReadings([_|Potential],Selected):-
   localConsistentReadings(Potential,Selected).

localInformativeReadings([],[]).
localInformativeReadings([Drs|Potential],[Drs|Selected]):-
   localInformative(Drs), !,
```

```
   localInformativeReadings(Potential,Selected).
localInformativeReadings([_|Potential],Selected):-
   localInformativeReadings(Potential,Selected).

/*========================================================================
   Informativity
========================================================================*/

informative(drs([],[]),_):-!.
informative(OldDrs,NewDrs):-
   backgroundKnowledge(Chi),
   drs2fol(OldDrs,Phi),
   drs2fol(NewDrs,Psi),
   callTheoremProver(Chi,~(Phi > Psi),Proof),
   (Proof=yes, !, fail; true).

/*========================================================================
   Consistency
========================================================================*/

consistent(NewDrs):-
   domainSize(NewDrs,Size),
   backgroundKnowledge(Chi),
   drs2fol(NewDrs,Phi),
   (
       callModelBuilder(Chi,Phi,Size,Model),
       Model=1,!
   ;
       callTheoremProver(Chi,Phi,Proof),
       (
           Proof=yes, !, fail
       ;
           true
       )
   ).

/*========================================================================
   Local Informativity
========================================================================*/

localInformative(Drs):-
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalInformative(List).

allLocalInformative([]).
allLocalInformative([(Super,Sub)|Others]):-
   backgroundKnowledge(Chi),
   drs2fol(drs([],[Super>Sub]),Phi),
   callTheoremProver(Chi,~Phi,Proof),
   (Proof=yes, !, fail; allLocalInformative(Others)).
```

```
/*========================================================================
   Local Consistency
========================================================================*/

localConsistent(Drs):-
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalConsistent(List).

allLocalConsistent([]).
allLocalConsistent([(Super,Sub)|Others]):-
   backgroundKnowledge(Chi),
   drs2fol(drs([],[Super>drs([],[~Sub])]),Phi),
   callTheoremProver(Chi,~Phi,Proof),
   (Proof=yes, !, fail; allLocalConsistent(Others)).

/*========================================================================
   Background Knowledge
========================================================================*/

backgroundKnowledge(Formulas):-
   knowledge(Formulas1),
   generateOntology(Formulas2),
   append(Formulas1,Formulas2,Formulas).

knowledge([
           forall(X,forall(Y,have(X,Y)>of(Y,X))),
           forall(X,forall(Y,of(Y,X)>have(X,Y))),
           forall(X,female(X)&married(X)>exists(Y,husband(Y)&have(X,Y))),
           forall(X,forall(Y,husband(Y)&have(X,Y)>married(X)&female(X)))
          ]).

/*========================================================================
   Domain Size
========================================================================*/

domainSize(drs(Dom,_),Size):-
   (
    Dom=[],
    Size=1,!
   ;
    length(Dom,Size)
   ).

/*========================================================================
   Compute super- and subordinated DRSs
========================================================================*/

superSubDrs(drs(D,[Sub > _|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
```

```
superSubDrs(drs(D,[B > Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(merge(drs(D,C),B),Drs),Super).
superSubDrs(drs(D,[B1 > B2|C]),Drs-Super,Sub):-
   superSubDrs(B2,merge(Drs,merge(merge(drs(D,C),B1),B2))-Super,Sub).
superSubDrs(drs(D,[Sub v _|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[_ v Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[B v _|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[_ v B|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[~ Sub|C]),Drs-Super,Sub):-
   mergeDrs(merge(Drs,drs(D,C)),Super).
superSubDrs(drs(D,[~ B|C]),Drs-Super,Sub):-
   superSubDrs(B,merge(Drs,merge(drs(D,C),B))-Super,Sub).
superSubDrs(drs(D,[Cond|C]),Drs-Super,Sub):-
   superSubDrs(drs([],C),Drs-B,Sub),
   mergeDrs(merge(drs(D,[Cond]),B),Super).
```

```
/*************************************************************************

        name: callTheoremProver.pl (Chapter 5)
     version: June 18, 1998
 description: Prolog Interface to Otter (Sicstus required)
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(callTheoremProver,[callTheoremProver/3]).

:- use_module(library(system)),
   use_module(fol2otter,[fol2otter/2]).

/*========================================================================
   Calls to Theorem Prover (Otter)
========================================================================*/

callTheoremProver(Axioms,Formula,Proof):-
   fol2otter(Axioms,Formula),
   shell('./otter < temp.in > temp.out 2> /dev/null',X),
   (X=26368,Proof=yes,!;X=26624,Proof=no).
```

```
/*************************************************************************

         name: callModelBuilder.pl (Chapter 5)
      version: September 3, 1999
  description: Prolog Interface to Mace (Sicstus required)
      authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(callModelBuilder,[callModelBuider/4]).

:- use_module(library(system)),
   use_module(comsemPredicates,[append/3]),
   use_module(fol2otter,[fol2otter/2]).

/*========================================================================
   Calls to Model Generator (Mace)

   Changed in MACE (generate.c):
   #define MAX_SYMBOLS   100  number of functors (was 50)
========================================================================*/

callModelBuilder(Axioms,Formula,DomainSize,Model):-
   fol2otter(Axioms,Formula),
   name('./mace -n',C1),
   name(DomainSize,C2),
   append(C1,C2,C3),
   name(' -p -t2 -m1 < temp.in > temp.out 2> /dev/null',C4),
   append(C3,C4,C5),
   name(Shell,C5),
   shell(Shell,X),
   % this is not correct. It always returns 0!
   (X=0,Model=1,!;write(X),Model=0).
```

```
/*************************************************************************

        name: fol2otter.pl (Chapter 11)
     version: June 18, 1998
 description: Translates a formula in otter syntax to standard output
     authors: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(fol2otter,[fol2otter/2]).

:- use_module(comsemOperators).

/*========================================================================
   Translates formula to otter syntax in file 'temp.in'
========================================================================*/

fol2otter(Axioms,Formula):-
   tell('temp.in'),
   format('set(auto).~n~n',[]),
   format('clear(print_proofs).~n~n',[]),
   format('set(prolog_style_variables).~n~n',[]),
   format('formula_list(usable).~n~n~n',[]),
   printOtterList(Axioms),
   printOtterFormula(Formula),
   format('~nend_of_list.~n',[]),
   told.

/*========================================================================
   Print a list of Otter formulas
========================================================================*/

printOtterList([]).
printOtterList([X|L]):-
        printOtterFormula(X),
        printOtterList(L).

/*========================================================================
   Print an Otter formula
========================================================================*/

printOtterFormula(F):-
          \+ \+ (numbervars(F,0,_), printOtter(F,5)),
          format('.~n',[]).

printOtter(exists(X,Formula),Tab):-
   write('(exists '),write(X),write(' '),!,
   printOtter(Formula,Tab),write(')').

printOtter(forall(X,Formula),Tab):-
```

```
   write('(all '),write(X),write(' '),!,
   printOtter(Formula,Tab),write(')').

printOtter(Phi & Psi,Tab):-
   write('('),!,
   printOtter(Phi,Tab),
   write(' & '), nl, tab(Tab),
   NewTab is Tab + 5,
   printOtter(Psi,NewTab), write(')').

printOtter(Phi v Psi,Tab):-
   write('('),!,
   printOtter(Phi,Tab), write(' | '),
   printOtter(Psi,Tab), write(')').

printOtter(Phi <> Psi,Tab):-
   write('('),!,
   printOtter(Phi,Tab), write(' <-> '),
   printOtter(Psi,Tab), write(')').

printOtter(Phi > Psi,Tab):-
   write('('),!,
   printOtter(Phi,Tab), write(' -> '),
   printOtter(Psi,Tab), write(')').

printOtter(~ Phi,Tab):-
   write('-('),!,
   printOtter(Phi,Tab), write(')').

printOtter(Phi,_):-
   write(Phi).
```

```
/*************************************************************************

         name: mainPresupDRTU.pl (Chapter 12)
      version: May 28, 1998
  description: Combining DRTU with presupposition resolution
       author: Patrick Blackburn & Johan Bos

*************************************************************************/

:- module(mainPresupDRTU,[parse/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,
                                printReadings/1,compose/3]),
   use_module(betaConversion,[betaConvert/2]),
   use_module(pluggingAlgorithm,[plugHole/4]),
   use_module(mergeDRT,[mergeDrs/2]),
   use_module(mergeUSR,[mergeUSR/2]),
   use_module(resolvePresup,[projectDrs/1,accommodate/2]).

:- [englishGrammar], [englishLexicon], [semMacrosPresupDRTU].

/*========================================================================
   Driver Predicate
========================================================================*/

parse:-
   readLine(Discourse),
   d(Sem,Discourse,[]),
   betaConvert(merge(usr([Top,Main],[],[]),Sem@Top@Main),Reduced),
   mergeUSR(Reduced,usr(D,L,C)),
   printRepresentation(usr(D,L,C)),
   findall(Drs,
              (
               plugHole(Top,L-[],C,[]),
               projectDrs([Top,pre([])]-[PresupDrs,pre(P)]),
               accommodate(P,PresupDrs-Drs)
              ),
          Readings),
   printReadings(Readings).
```

```
/*************************************************************************

         name: semMacrosPresupDRTU.pl (Chapter 12)
      version: June 18, 1999
  description: Semantic Macros for DRTU with Presupposition Resolution
       author: Patrick Blackburn & Johan Bos

*************************************************************************/

/*========================================================================
   Semantic Macros
========================================================================*/

detSem(uni,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([F,R,S],
                       [F:drs([],[merge(drs([X],[]),R)>S])],
                       [leq(F,H),leq(L,S)]),P@X@H@R),Q@X@H@L)))))).

detSem(indef,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([E,R,S],
                       [E:merge(merge(drs([X],[]),R),S)],
                       [leq(E,H),leq(L,S)]),P@X@H@R),Q@X@H@L)))))).

detSem(def,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([R,S],
                       [L:alfa(X,def,merge(drs([X],[]),R),S)],
                       [leq(L,H)]),P@X@H@R),Q@X@H@S)))))).

detSem(poss(Gender),lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([R,S],
                       [L:alfa(X,def,alfa(Y,nonrefl,drs([Y],[Cond]),
                                  merge(drs([X],[of(X,Y)]),R)),S)],
                       [leq(L,H)]),P@X@H@R),Q@X@H@S)))))):-
   compose(Cond,Gender,[Y]).

nounSem(Sym,lambda(X,lambda(_,lambda(L,usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

pnSem(Sym,Gender,lambda(P,lambda(H,lambda(L,
      merge(usr([S],
                [L:alfa(X,name,drs([X],[X=Sym,Cond]),S)],
                [leq(L,H)]),
            P@X@H@S))))):-
   compose(Cond,Gender,[X]).

proSem(Gender,Type,lambda(P,lambda(H,lambda(L,
      merge(usr([S],
                [L:alfa(X,Type,drs([X],[Cond]),S)],
                [leq(L,H)]),
            P@X@H@S))))):-
```

```
   compose(Cond,Gender,[X]).

ivSem(Sym,lambda(X,lambda(_,lambda(L,
      usr([],[L:drs([],[Cond])],[]))))):-
   compose(Cond,Sym,[X]).

tvSem(Sym,lambda(K,lambda(Y,K@lambda(X,lambda(_,lambda(L,
      usr([],[L:drs([],[Cond])],[]))))))):-
   compose(Cond,Sym,[Y,X]).

relproSem(lambda(P,lambda(Q,lambda(X,lambda(H,lambda(L,
          merge(usr([L2,L3,H1] ,
                    [L:merge(L3,H1)],
                    [leq(L2,H1)]),
                merge(P@X@H@L2,Q@X@H@L3)))))))).

prepSem(Sym,lambda(K,lambda(P,lambda(Y,lambda(H,lambda(L3,
        merge(K@lambda(X,lambda(H,lambda(L1,
              usr([L2,H1],
                  [L3:merge(L2,H1),L1:drs([],[Cond])],
                  [leq(L1,H1)]))))@H@L1,P@Y@H@L2))))))):-
   compose(Cond,Sym,[Y,X]).

modSem(neg,lambda(P,lambda(X,lambda(H,lambda(L,
       merge(usr([N,S],[N:drs([],[~S])],[leq(N,H),leq(L,S)]),
             P@X@H@L)))))).

coordSem(conj,lambda(X,lambda(Y,lambda(P,lambda(H,lambda(L,
         merge(usr([L1,L2],[L:merge(L1,L2)],[leq(L,H)]),
               merge(X@P@H@L1,Y@P@H@L2)))))))).

coordSem(disj,lambda(X,lambda(Y,lambda(P,lambda(H,lambda(L,
         merge(usr([L1,L2],[L:drs([],[L1 v L2])],[leq(L,H)]),
               merge(X@P@H@L1,Y@P@H@L2)))))))).

dcoordSem(cond,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:drs([],[H1 > H2])],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).

dcoordSem(conj,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:merge(H1,H2)],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).

dcoordSem(disj,lambda(C1,lambda(C2,lambda(H,lambda(L,
          merge(usr([H1,H2],[L:drs([],[H1 v H2])],[leq(L,H)]),
                merge(C1@H1@_,C2@H2@_))))))).
```