Welcome to the Course!

This tutorial is an introduction to the basic ideas, algorithms, and implementation techniques of computational discourse semantics. We will work with Discourse Representation Structures, the well known 'boxes' of Kamp's Discourse Representation Theory. The tutorial deals with the following topics:

- Discourse Representation Theory
- Building Discourse Representations
- Presupposition Projection
- Underspecification

Discourse Representation Theory has long been one of the most important approaches to theoretical semantics, and more recently it has also become important in computational semantics. Computational applications have led to the development of new techniques for working with DRSs: in particular the development of underspecification techniques and new resolution strategies. This course aims to give an overview of these ideas, and their implementation in Prolog.

Readers with relatively modest backgrounds in linguistics and logic should have no problems to follow this tutorial. However a working knowledge of Prolog will be assumed.

The tutorial material consists of selected chapters from *Representation and Inference for Natural Language*, an introduction to computational semantics written by Patrick Blackburn and Johan Bos. The reader will find here (drafts of) Chapter 2 (an introduction to semantic construction), Chapter 3 (underspecification), Chapter 7 (an overview of DRT), Chapter 8 (building DRSs), and Chapter 10 (presupposition projection). The book is not yet finished, and feedback on this material would be greatly appreciated.

Johan Bos,
Computerlinguistik, Universität des Saarlandes,
July 1998.

Information on this course, as well as drafts of the full version of the book are available on the World Wide Web via URL:

`http://www.coli.uni-sb.de/~bos/comsem/`

Here you will also find the source code of the Prolog programs that are used in the course.

In addition, for your computational semantics exploring pleasure, we've put all the tools we've developed in this course together in one system: *DORIS*, a discourse representation and inference system. DORIS is available on-line and waiting for your input via:

`http://www.coli.uni-sb.de/~bos/atp/doris.html`

This system takes as input an English sentence, and then computes the DRSs, which are checked for consistency by feeding their first-order translations to a theorem prover, following by filtering out the disallowed readings.

The system is still under development and we hope to improve and extend it as we continue working on later versions of the book.

Representation and Inference for Natural Language

A First Course in Computational Semantics

Patrick Blackburn & Johan Bos

July 17, 1998

Contents

Preface **v**

I Working with First-Order Logic 1

1 First-Order Logic **3**

1.1 First-Order Logic 3

1.2 A Simple Model Checker 16

1.3 Some Refinements 23

1.4 First-Order Logic and Natural Language 29

2 Lambda Calculus **31**

2.1 Compositionality 31

2.2 Two Experiments 35

2.3 The Lambda Calculus 43

2.4 Implementing Lambda Calculus 49

2.5 Grammar Engineering 57

3 Underspecified Representations **67**

3.1 Scope Ambiguities 67

3.2 Montague's Approach 69

3.3 Storage Methods 74

3.4 Underspecification 87

4 Propositional Inference **99**

4.1 Propositional Tableaux 101

4.2 Implementing Propositional Tableaux 112

4.3 Equality Constraints 115

4.4 Theoretical Remarks 115

5 First-Order Inference **121**

5.1 First-Order Tableaux 121

5.2 Unification and Free Variable Tableaux 126

5.3 Implementing Free Variable Tableaux 136

5.4 Problems and Prospects 141

6 Applications I **143**

6.1 Questions 143

6.2 Basic Argumentation 145

6.3 Argumentation with Scope Ambiguities 147

6.4 Other Ideas 148

II Working with Discourse Representations **149**

7 Discourse Representation Theory **151**

7.1 An Overview of DRT 152

7.2 Interpreting DRSs 167

7.3 DRT and First-Order Logic 172

7.4 DRT in Prolog 178

8 Building Discourse Representations **185**

8.1 DRS-Threading 186

8.2 Building DRSs with Lambdas 195

8.3 Threading in λ-DRT 201

8.4 Underspecified DRSs 203

9 Pronoun Resolution **211**

9.1 Constraints on Resolution 211

9.2 Top-Down 214

9.3 Bottom-Up 216

10 Presupposition Projection **221**

10.1 Introducing Presuppositions 221

10.2 Dealing with Presupposition in DRT 224

10.3 Presupposition Projection in Prolog 230

11 VP Ellipsis **237**

12 Applications II **239**

12.1 Consistency and Informativity 239

12.2 Presupposition Accommodation 245

Bibliography **251**

III Appendices **257**

A Propositional Languages **259**

B Type Theory **261**

C Prolog in a Nutshell **265**

D Listing of Programs **281**

Preface

This book introduces a number of fundamental techniques for computing semantic representations for fragments of natural language and performing inference with the result. Both the underlying theory and their implementation in Prolog are discussed. We believe that the reader who masters these techniques will be in a good position to appreciate (and critically assess) ongoing developments in computational semantics.

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

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

Two comments should immediately be made. First, we've presented semantic construction and inference as if they were independent, but of course they're not. Indeed, how semantic construction and inference can best be interleaved is a deep and important problem. We won't solve it, but by the end of the book we will have seen some nice examples of semantic construction and inference harnessed together in a single architecture.

Second, our working definition of computational semantics isn't quite as innocent as it looks. Many semanticists claim that intermediate levels of logical representation are essentially redundant. Whether or not this is true, the move to a computational perspective on formal semantics certainly increases the *practical* importance of the representation level. Logical representations — that is, formulas of a logical language — encapsulate meaning in a clean compact way. They make it possible to use well understood proof systems to

perform inference, and we shall learn how to exploit this possibility. Models may be the heart of semantics, but representations are central to its computational cousin. However, as will become clear in the course of the book, we feel that the computational perspective vividly brings out *theoretical* importance of representations. The success of Discourse Representation Theory, the explosion of interest in underspecification, and the exploration of alternative glue languages such as linear logic to drive the process of semantic construction all bear witness to an important lesson: semanticists ignore representations at their peril. Representations are well-defined mathematical entities that (among other things) can be formed into more abstract abstract entities, explored geometrically, and specified indirectly with the aid of constraints; we'll see examples of all these things in this book. Even if representations are eliminable, they need to be taken seriously.

This book is divided into two parts. The key aim of Part 1, *Working with First-Order Logic*, is to develop a basic tool-kit for doing computational semantics. The chapter by chapter breakdown of this part is as follows:

Chapter 1. First-Order Logic. Here we introduce the syntax and semantics of first order logic (the representation language used in the first half of the book), show how formulas and (simple finite) models may be represented in Prolog, build a simple model checker, and discuss the problems that arise when Prolog variables are used to represent first-order variables.

Chapter 2. Lambda Calculus. Here we start studying semantic construction. We outline the methodology underlying our work (namely, *compositionality*) and motivate our use of Prolog's Definite Clause Grammars. We then write two rather naive programs that build semantic representations for a very small fragment of English. These experiments lead us to the lambda calculus, the tool that drives this books approach to semantic construction. We implement β-conversion, the computational core of the lambda calculus, in two different ways, and then integrate it into the grammatical architecture that will be used throughout the book.

Chapter 3. Underspecified Representations. Here we investigate a fundamental problem for computational semantics: scope ambiguities. These are semantic ambiguities that can arise in syntactically unambiguous expressions, and they pose a problem for compositional approaches to semantic construction. We illustrate the problem, and present four (increasingly more sophisticated) solutions: Montague's use of quantifier raising, Cooper storage, Keller storage, and a more recent underspecification-based method called Hole Semantics. We implement all four approaches.

Chapter 4. Propositional Inference. Here we turn to the second major theme of the book: inference. Our approach to inference will be based on first-order theorem proving and model building, and in this chapter we start developing the necessary tools. We introduce a *signed tableaux* system for propositional calculus, show how to implement it in Prolog, discuss some extensions, and conclude with an informal discussion of a number of theoretical issues.

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

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

Having developed our toolkit, it is time to put it to work, and this is what Part II does. First, 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 Part I without difficulty. We then investigate some interesting semantic phenomena, namely pronoun resolution, presupposition, and VP ellipsis, and integrate — at least partially — our work on inference with our work on semantic construction. Chapter by chapter, Part II looks like this:

Chapter 7. 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 8. 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 Chapter 2 to the setting of DRT; we call the resulting blend λ-DRT. Finally, we show how threading and lambdas can be combined. We then show that the hole semantics developed in Part I adapts straightforwardly to DRT, thus giving us a mechanism for copiung with scope ambiguities.

Chapter 9. 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.

Chapter 10. Presupposition Projection. 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 need to be accommodated, offering a novel perspective on presupposition projection. In this chapter we show how the pronoun resolution techniques introduced in Chapter 9 can be extended to handle presupposition accommodation.

Chapter 11. VP Ellipsis. "Mia betrayed her husband, and Jody did too" is ambiguous. Either this discourse conveys that Jody betrayed Mia's husband, or that she betrayed her own husband. In this chapter we provide a computational analysis of the strict and sloppy readings that arise with verb phrase ellipsis.

Chapter 12. Applications II. Here we combine the techniques we have acquired in Part II. In particular, we show how how the inference techniques we have discussed can used to filter out ambiguities. To be more concrete, in the discourse "Mia betrayed her husband. Vincent did too." only the strict reading is available. We'll try to put together system that predicts that.

Each chapter concludes with two sections: *Software Summary* lists the programs developed in the chapter, and *Further Reading* lists references the reader may find helpful.

We have tried to make this book self-contained, and believe that readers with relatively modest backgrounds in linguistics and logic should be able to follow the theoretical discussion. But the fact remains that this is a book in *computational* semantics, and getting the most out of it requires a working knowledge of Prolog. We have included an appendix which outlines the main ideas of Prolog, and we hope that this will be useful — but, bluntly, the reader who has never thought about computational problems from the declarative perspective typical of logic or constraint-based programming will find parts of it tough going. We develop many theoretical ideas by thinking problems through from a declarative perspective. For example, our account of the lambda calculus in Chapter 2 makes no appeal to types, function valued functions, or higher-order logic — rather, lambda calculus is presented as a beautiful piece of data-abstraction that emerges naturally from a declarative analysis of semantic construction. Computational ideas are not an optional extra in computational semantics; they are the heart of the enterprise.

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 Part 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. Verb*mobil* aims for automatic speech-to-speech translation of negotiation dialogues between English, German, and Japanese, where the linguistic transfer is carried out via semantic representations. A lot of the material presented in this book, especially the techniques in semantic construction, underspecification, and evaluation, benefited from the experience gained throughout this project.

Conversations with Martin Kay and Reinhard Muskens were helpful in clarifying our goals. We're grateful to David Beaver and Björn Gambäck for their comments on an early draft of this book. Finally, 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.

Patrick Blackburn and Johan Bos,
Computerlinguistik, Universität des Saarlandes,
June 1998.

Part I

Working with First-Order Logic

Chapter 2

Lambda Calculus

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

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

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

2.1 Compositionality

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

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

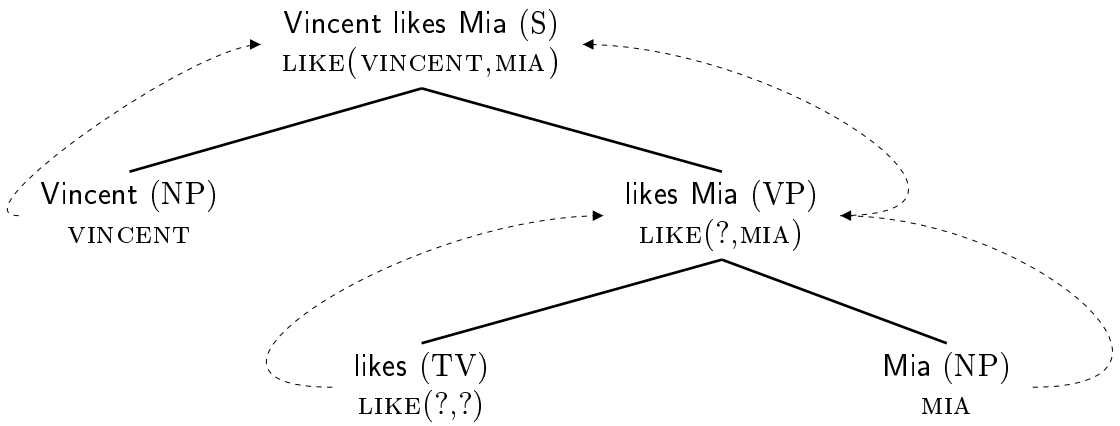
VINCENT to the representation, the transitive verb 'likes' contributes the relation symbol LIKE, and 'Mia' contributes MIA.

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

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

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

(1)



Why is MIA in the second argument slot of LIKE? Because, when we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP (in this case, MIA) in the *second* argument slot of the VP's semantic representation (in

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

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

Compositionality is a simple and natural concept that underlies most work in natural language semantics and the semantics of programming languages. Nonetheless, in spite of its simplicity, it raises a number of interesting issues. For example, is every 'systematic' semantics a compositional one? In Part II we discuss the standard top-down algorithm for Discourse Representation Theory. This algorithm is a systematic approach to semantic construction (under any reasonable interpretation of the word 'systematic') but many semanticists have argued that it is not truly compositional.

Syntax via Definite Clause Grammars

So, is there a systematic way of translating simple sentences such as 'Vincent likes Mia' and 'A woman snorts' into first-order logic? We don't yet have a method, but at least we now have a plausible strategy for finding one. We need to:

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

Task 2 Specify semantic representations for the lexical items.

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

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

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

We have opted for a particularly simple solution: in this book the syntactic analysis of a sentence will be a tree whose non-leaf nodes represent *complex syntactic categories* (such as S, NP and VP) and whose leaves represent *lexical items* (these are associated with *basic syntactic categories* such as noun, transitive verb, determiner, proper name and intransitive verb). The tree the reader has just seen is a typical example. This approach has an obvious drawback (namely, the reader won't learn anything interesting about syntax) but it also has one overwhelming advantage: we will be able to make use of Definite Clause Grammars (DCGs), the in-built Prolog mechanism for grammar specification.

Here is a DCG for the fragment of English we shall work with in the semantic construction experiments which follow. (This DCG, decorated with semantic construction code, can be found in `experiment1.pl` and `experiment2.pl`. The reader unfamiliar with DCGs is advised to consult Appendix C right away, as otherwise this chapter will shortly stop making much sense.)

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

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

Vincent walks

because 'Vincent' is declared as a proper name, and proper names are noun phrases according to this grammar; 'walks' is an intransitive verb, and hence a verb phrase; and sentences can consist of a noun phrase followed by a verb phrase.

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

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

we can test whether 'Mia like a foot massage' is accepted by the grammar, and the query

```
s(X,[])
```

generates all grammatical sentences.

Exercise 2.1.1 How many sentences are accepted by this grammar? How many noun phrases? How many verb phrases? Check your answer by generating the relevant items.

With a little effort, we can do a lot more. In particular, by making use of extra arguments (again, see Appendix C if you're uncertain what this means) we can associate semantic representations with lexical items very straightforwardly. The normal Prolog unification mechanism then gives us the basic tool needed to combine semantic representations, and to pass them up towards sentence level. In short, working with DCGs both frees us from having to implement parsers, and makes available a powerful mechanism for combining representations, so we'll be able to devote our attention to semantic construction.

The semantic construction methods developed in this book are compatible with a wide range of syntactical theories; indeed they are also compatible with statistical approaches to parsing. Much of this book is about exploiting the recursive structure of trees to build representations compositionally; where the trees actually come from is relatively unimportant. In short, we have have chosen to fill in the syntactical 'black box' using DCGs — but there is a wide variety of options and we strongly encourage our readers to experiment.

2.2 Two Experiments

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

Experiment 1

First the lexical items. We need to associate 'Vincent' with the constant VINCENT, 'Mia' with the constant MIA, 'walks' with the unary relation symbol WALK, and 'loves' with

the binary relation symbol LOVE. The following piece of DCG code makes these associations. Note that the arity of `walk` and `love` are explicitly included as part of the Prolog representation.

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

pn(mia)-->
   [mia].

iv(snort(_))-->
   [snorts].

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

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

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

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

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

vp(Sem)-->
```

```
    tv(Sem),
    np(SemNP),
    {
     arg(2,Sem,SemNP)
    }.

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

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

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

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

we obtain the (correct) response:

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

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

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

det(forall(_,_>_))-->
    [every].

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

```
noun(footmassage(_))-->
     [foot,massage].
```

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

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

```
np(Sem)-->
   det(Sem),
   noun(SemNoun),
   {
    arg(1,SemNoun,X),      % The 1st argument of the noun semantics is X
    arg(1,Sem,X),          % The 1st argument of the determiner is X
    arg(2,Sem,Matrix),     % The 2nd argument of the determiner is Matrix
    arg(1,Matrix,SemNoun)  % The 1st argument of Matrix is SemNoun
   }.
```

The key idea is that the representation associated with the NP will be the representation associated with the determiner (note that the `Sem` variable is shared between `np` and `det`), but with this representation fleshed out with additional information from the noun. The Prolog variable `X` is a name for the existentially quantified variable the determiner introduces into the semantic representation; the code `arg(1,SemNoun,X)` and `arg(1,Sem,X)` unifies the argument place of the noun with this variable. The code `arg(2,Sem,Matrix)` simply says that the second argument of `Sem` will be the matrix of the NP semantic representation, and `arg(1,Matrix,SemNoun)` then adds more detail: it says that the first slot of the matrix will be filled in by the semantic representation of the noun. So if we pose the query

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

we obtain the response

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

Note that this representation is an *incomplete* first-order formula. We don't yet have a full first-order formula (the Prolog variable `Y` has yet to be instantiated) but we do know that we are existentially quantifying over the set of women.

Given that such incomplete first-order formulas are the semantic representations associated with quantified NPs, it is fairly clear what must happen when we combine a quantified NP with a VP to form an S: the VP must provide the missing piece of information. (That is, it must provide an instantiation for `Y`.) The following clause does this:

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

Unfortunately, while the underlying idea is essentially correct, things have just started going badly wrong. Until now, we've simply been extending the rules of our original DCG with semantic information — and we've already dealt with `s --> np, vp`. If we add this second version of `s --> np, vp` (and it seems we need to) we are duplicating syntactic information. This is uneconomical and inelegant. Worse, this second sentential rule interacts in an unintended way with the rule

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

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

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

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

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

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

Exercise 2.2.1 The code for experiment 1 is in `experiment1.pl`. Generate the semantic representations of the sentences and noun phrases yielded by the grammar.

Experiment 2

Although our first experiment was ultimately unsuccessful, it did teach us something useful: to build representations, we need to work with *incomplete* first-order formulas, and we need a way of manipulating the missing information. Consider the representations associated with determiners. In experiment 1 we associated 'a' with `exists(_,_&_)`. That is, this determiner contributes the skeleton of a first-order formula whose first slot needs to be instantiated with a variable, whose second slot needs to be filled with the semantic representation of a noun, and whose third slot needs to be filled by the semantic representation of a VP. However in experiment 1 we didn't manipulate this missing information directly. Instead we took a shortcut: we thought in terms of argument position so that we could make use of `arg/3`. Let's avoid plausible looking short cuts. The idea of missing information is evidently important, so let's take care to always associate it with an explicit Prolog variable. Perhaps this direct approach will make semantic construction easier.

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

```
det(X,Restr,Scope,exists(X,Restr & Scope))-->
   [a].

det(X,Restr,Scope,forall(X,Restr > Scope))-->
   [every].
```

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

and again, we should keep track of the choices we made. So the following lexical entries are called for:

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

iv(Y,snort(Y))-->
   [snorts].

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

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

```
s(Sem)-->
   np(X,SemVP,Sem),
   vp(X,SemVP).

vp(X,Sem)-->
   tv(X,Y,SemTV),
   np(Y,SemTV,Sem).

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

The semantic construction rule associated with the sentential rule, for example, tells us that `Sem`, the semantic representation of the sentence, is essentially going to be that of the noun phrase (that's where the value of the `Sem` variable will trickle up from) but that, in addition, the bound variable `X` used in `Sem` must be the same as the variable used in the verb phrase semantic representation `SemVP`. Moreover, it tells us that `SemVP` will be used to fill in the information missing from the semantic representation of the noun phrase.

So far so good, but now we need a little trickery. Experiment 1 failed because there was no obvious way of making use of the semantic representations supplied by quantified noun phrases and proper names in a single sentential rule. Here we only have a single sentential rule, so evidently the methods of experiment 2 avoid this problem. Here's how it's done:

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

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

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

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

The second rule is trickier. The vital work is performed by the doubled `Sem` variable. Roughly speaking, when we combine the noun phrase representations formed by this rule with a verb phrase, the effect of this doubling is to 'move the verb phrase representation rightwards' so that it occupies the slot that will ultimately be used in forming the sentential semantics. Intuitively, whereas the representation for sentences that have a quantified noun phrase as subject are essentially the subject's representation filled out by the verb phrase representation, the reverse is the case when we have a proper name as subject. When that happens, the verb phrase is boss. The sentence representation is essentially the verb phrase representation, and the role of the proper name is simply to obediently instantiate the marked verb phrase slot. The rightwards shuffle performed by the doubled variables is the Prolog mechanism which captures this role-reversal.

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

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

Exercise 2.2.2 Using either pen and paper or a tracer, compare the sequence of variable instantiations this program performs when building representations for 'Vincent snorts' and 'A woman snorts', and 'Vincent loves Mia' and 'Vincent loves a woman'.

2.3 The Lambda Calculus

For present purposes we shall view lambda calculus as a notational extension of first order logic that allows us to bind variables using a new variable binding operator λ. Occurrences of variables bound by λ should be thought of as placeholders for missing information: they explicitly mark where we should substitute the various bits and pieces obtained in the course of semantic construction. An operation called β-conversion performs the required substitutions. We suggest that the reader think of the lambda calculus as a special programming language dedicated to a single task: gluing together the items needed to build semantic representations. This glue language turns out to remarkably flexible: it is compatible with the underspecified representations introduced in Chapter 3, and with the Discourse Representation Structures studied in Part II.

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

λx.MAN(x)

Here the prefix λx. binds the occurrence of x in MAN(x). We sometimes say that the prefix λx. *abstracts over* the variable x. We call expressions with such prefixes *lambda abstractions* (or, more simply, *abstractions*). In our example, the binding of the free x variable in MAN(x) explicitly indicates that MAN has an argument slot where we may perform substitutions. More generally, the purpose of abstracting over variables is to mark the slots where we want substitutions to be made.

Concatenation indicates that we wish to perform substitution. Consider the following expression:

(λx.MAN(x))VINCENT.

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

MAN(VINCENT).

Abstraction, functional application, and β-conversion underly much of our subsequent work. The business of specifying semantic representations for lexical items is essentially

going to boil down to devising lambda abstractions that specify the missing information, while functional application coupled with β-conversion will be the engine used to combine semantic representations compositionally.

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

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

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

We are almost ready to examine some linguistic examples, but let's first clarify one point. The lambda expressions $\lambda x.\textsc{man}(x)$, $\lambda y.\textsc{man}(y)$, and $\lambda z.\textsc{man}(z)$ are equivalent, as are the expressions $\lambda Q.\exists x(\textsc{woman}(x) \wedge Q(x))$ and $\lambda Y.\exists x(\textsc{woman}(x) \wedge Y(x))$. All these expressions are functors which when applied to an argument, replace the bound variable by the argument. No matter which argument $\mathcal{A}$ we choose, the result of applying any of the first three expressions to $\mathcal{A}$ and then β-converting is $\textsc{man}(\mathcal{A})$, and the result of applying either of the last two expressions to $\mathcal{A}$ is $\exists x(\textsc{woman}(x) \wedge \mathcal{A}(x))$. In short, relabeling bound variables yields equivalent lambda expressions.

The process of relabeling bound variables is called *α-conversion*. If a lambda expression $\mathcal{E}$ can be obtained from a lambda expression $\mathcal{E}'$ by α-conversion then we say that $\mathcal{E}$ and $\mathcal{E}'$ are *α-equivalent*. In what follows we often treat α-equivalent expressions as if they were identical. For example, we will sometimes say that the lexical entry for some word is a lambda expression $\mathcal{E}$, but when we actually work out some semantic construction, we might use an α-equivalent expression $\mathcal{E}'$ instead of $\mathcal{E}$ itself. As λ-bound variables are merely placeholders for substitution slots, this is clearly semantically acceptable. But the reader needs to understand that it's not merely *permissible* to do this, it can be *vital* to do so if β-conversion is to work as intended.

Suppose that the expression $\mathcal{F}$ in $\lambda V.\mathcal{F}$ is a complex expression containing many λ operators. Now, it could happen that when we apply a functor $\lambda V.\mathcal{F}$ to an argument $\mathcal{A}$, some occurrence of a variable that is is free in $\mathcal{A}$ becomes bound by a lambda operator when we substitute it into $\mathcal{F}$. *We don't want this to happen*. Such accidental bindings (as they are usually called) defeat the purpose of working with the lambda calculus. The whole point of developing the lambda calculus was to gain control over the process of performing substitutions. We don't want to lose control by foolishly allowing unintended interactions.

Such interactions need never be a problem. We don't need to use $\lambda V.\mathcal{F}$ as the functor; any α-equivalent formula will do. By suitably relabeling the bound variables in $\lambda V.\mathcal{F}$ we can always obtain an α-equivalent functor that doesn't bind any of the variables that occur free in $\mathcal{A}$, and accidental binding is prevented. Thus, strictly speaking, it is not merely functional application coupled with β-conversion that drives the process of semantic construction in this book, but functional application and β-conversion coupled with (often tacit) use of α-conversion.

That's all we need to know about the lambda calculus for now. Let's see how to put it to work in computational semantics. For the time being, we'll continue to work with the simple grammar of experiments 1 and 2.

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

'every': λP.λQ.$\forall$x(P(x)$\rightarrow$Q(x))

'boxer': λy.BOXER(y)

'walks': λx.WALK(x).

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

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

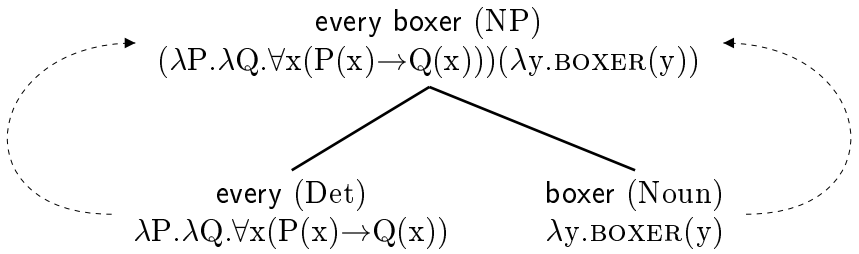
If we use the Prolog variable `P` instead of `Restr`, and `Q` instead of `Scope` this becomes

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

which is clearly analogous to λP.λQ.$\forall$x(P(x)$\rightarrow$Q(x)).

But there are also important differences. The experiment 2 representation is a Prolog-specific encoding of missing information. In contrast, lambda expressions are programming language independent (we could work with them in C++ or Lisp, for example). Moreover, experiment 2 solved the problem of combining missing information on a rule-by-rule basis. However, as will soon be clear, in functional application and β-conversion we have isolated two crucial combinatorial mechanisms which drive the process of semantic construction, and this is going to simplify matters considerably.

Let's return to 'every boxer walks'. According to our grammar, a determiner and a common noun can combine to form a noun phrase. Our semantic analysis couldn't be simpler: we will associate the NP node with the functional application that has the determiner representation as functor and the noun representation as argument.



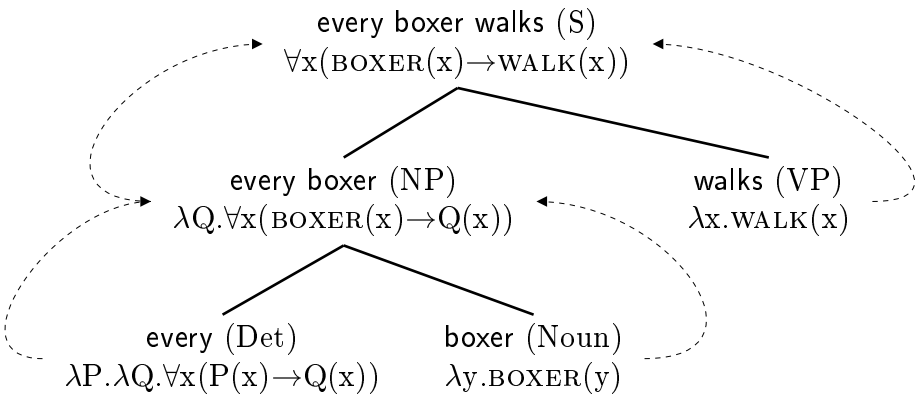
Now, applications are instructions to carry out β-conversion, so let's do what is required. (Note that as there are no free-occurrences of variables in the argument expression, there is no risk of accidental variable capture, so we don't need to α-convert the functor.) Performing the demanded substitution yields:

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

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

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

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



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

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

'every boxer walks': $(\lambda Q.\forall x(\text{BOXER}(x)\rightarrow Q(x)))(\lambda x.\text{WALK}(x))$.

Performing β-conversion yields:

'every boxer walks': $\forall x(\text{BOXER}(x)\rightarrow (\lambda x.\text{WALK}(x))(x))$.

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

'every boxer walks': $\forall x(\text{BOXER}(x)\rightarrow\text{WALK}(x))$.

It is worth reflecting on this example, for it shows that in two important respects semantic construction is getting simpler. First, the process of combining two representations is now uniform: we simply say which of the representations is the functor and which the argument, whereupon combination is carried out by applying functor to argument and β-converting. Second, more of the load of semantic analysis is now carried by the lexicon: it is here that we use the lambda calculus to make the missing information stipulations.

Are there clouds on the horizon? For example, while the semantic representation of a quantifying noun phrase such as 'a woman' can be used as a functor, surely the semantic representation of an NP like 'Vincent' will have to be used as an argument? We avoided this problem in experiment 2 by the variable doubling trick used in the NP rule for proper names — but that was a Prolog specific approach, incompatible with the use of lambda calculus. Maybe — horrible thought! — we're going to be forced to duplicate syntactic rules again, just as we did in experiment 1.

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

'Mia': $\lambda P.P(\text{MIA})$

'Vincent': $\lambda P.P(\text{VINCENT})$

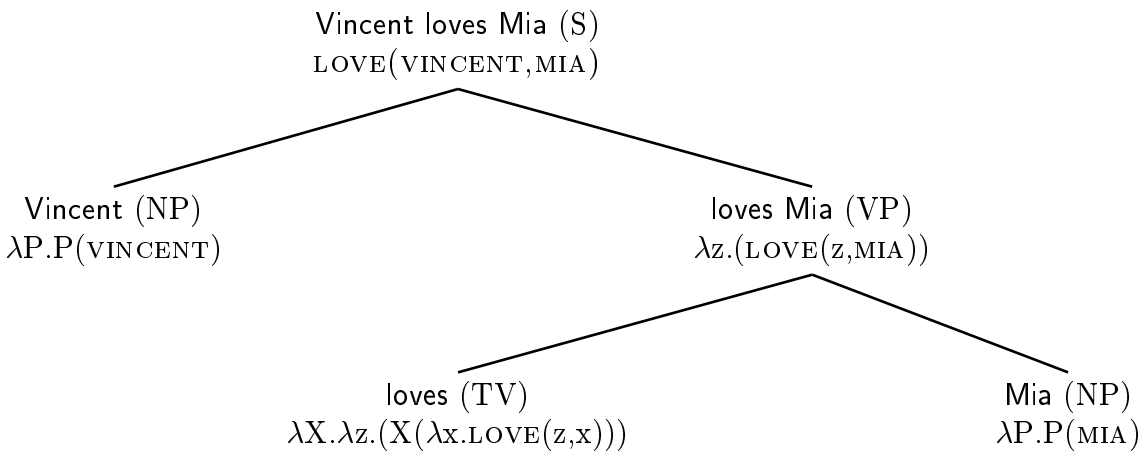
These representations are abstractions, thus they can be used as functors. However note what such functors do. They are essentially instructions to substitute their argument in P, which amounts to apply their own arguments to themselves! Because the lambda calculus

offers us the means to specify such role-reversing functors, the specter of syntactic rule duplication vanishes.

As an example of these new representations in action, let us build a representation for 'Vincent loves Mia'. We shall assume that TV semantic representations take their object NP's semantic representation as argument, so we assign 'loves' the following lambda expression:

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

And as in the previous example, the subject NP semantic representation takes the VP semantic representations as argument, so we can build the following tree:



Exercise 2.3.1 Work through the functional applications and β-conversions required to build the VP and S representations. Make sure you understand the role-reversing idea used in the TV semantic representation.

Let's sum up what we have achieved. Our decision to move beyond the approach of experiment 2 to the more disciplined approach of the lambda calculus was sensible. For a start, we don't need to spend any more time thinking about how to combine two semantic representations — functional application and β-conversion give us a general mechanism for doing so. Moreover, much of the real work is now being done at the lexical level; indeed, even the bothersome problem of finding a decent way of handling NP representations uniformly now has a simple lexical solution.

In fact, for the remainder of this book, the following version of the three tasks listed earlier will underly our approach to semantic construction:

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

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

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

We must now show that the second and third tasks lend themselves naturally to computational implementation.

2.4 Implementing Lambda Calculus

Our decision to perform semantic construction with the aid of an abstract glue language (namely, lambda calculus) has pleasant consequences for grammar writing. Because we have isolated the key combinatorial mechanisms (namely, functional application and β-conversion), we can implement them separately. That is, we can make these mechanisms available as black boxes to the grammar writer. From a grammar engineering perspective, this is extremely sensible: when writing fragments we should be free to concentrate on linguistic issues.

In this section we build the required black boxes. In fact, we build two versions of each. First we discuss a strikingly simple (but, alas, not fully correct) unification-based implementation of functional application and β-conversion, and then we present a correct implementation.

A Unification-Based Implementation

By making 'clever' use of Prolog variables, it is almost trivial to implement a black box for performing functional application and β-conversion. And once such a black box is available, we can decorated our little DCG with extremely natural semantic construction code. As the scare quotes indicate, our implementation of β-conversion is not quite as clever as it may at first glance seem, but we'll defer till later a discussion of the problems it can lead to.

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

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

Here `L` is intended to be either a Prolog variable, or is itself the Prolog representation of a lambda expression, while `F` is either a first-order formula, or is itself the Prolog representation of a lambda expression.

Second, we define what happens when we functionally apply an abstracted formula to an argument and carry out β-conversion. In fact, with clever use of Prolog unification we can do so with a single clause:

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

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

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

Result = walk(vincent)
```

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

In fact, the definition of `apply/3` can be further simplified:

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

And that's all there is to it.

Let's put our black box straight to work in our little DCG. We'll deal with the rules first. Actually, there's practically nothing that needs to said here. If we work with rules in the manner suggested by (our new version of) task 3, all we have to do is add on `apply/3` goals to the original DCG rules:

```
s(SemS)-->
   np(SemNP),
   vp(SemVP),
   {
    apply(SemNP,SemVP,SemS)
   }.

np(SemPN)-->
```

```
   pn(SemPN).

np(SemNP)-->
   det(SemDet),
   noun(SemNoun),
   {
    apply(SemDet,SemNoun,SemNP)
   }.

vp(SemIV)-->
   iv(SemIV).

vp(SemVP)-->
   tv(SemTV),
   np(SemNP),
   {
    apply(SemTV,SemNP,SemVP)
   }.
```

Note that the unary branching rules just percolate up their semantic representation, while the binary branching rules use `apply/3` to build a semantic representations out of their component representations in the manner suggested by task 3. Compared with the code in experiments 1 and 2, this is completely transparent: we simply apply function to argument to get the desired result.

The real work is done at the lexical level. Some of the lexical entries practically write themselves:

```
noun(lambda(X,footmassage(X)))-->
   [foot,massage].

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

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

However, when a lexical entry involves abstractions over complex variables, we add `apply/3` goals as well. For example, the code stating that λP.P(VINCENT) is the translation of 'Vincent', and λP.P(MIA) the translation of 'Mia', is as follows:

```
pn(lambda(P,Sem))-->
```

```
    [vincent],
    {
     apply(P,vincent,Sem)
    }.

pn(lambda(P,Sem))-->
    [mia],
    {
     apply(P,mia,Sem)
    }.
```

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

```
det(lambda(P,lambda(Q,exists(X,Restr & Scope))))-->
    [a],
    {
     apply(P,X,Restr),
     apply(Q,X,Scope)
    }.

det(lambda(P,lambda(Q,forall(X,Restr > Scope))))-->
    [every],
    {
     apply(P,X,Restr),
     apply(Q,X,Scope)
    }.
```

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

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

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

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

Exercise 2.4.1 Give the Prolog code for the lexical entry of the transitive verb 'likes'.

Exercise 2.4.2 Find a suitable lambda expression for the lexical entry of the determiner 'no', and then give the corresponding Prolog code.

Exercise 2.4.3 Extend the DCG with a simple treatment of adjectives. Your grammar should be able to assign formulas to expressions like 'A pretty woman dances with a tough guy'.

Exercise 2.4.4 Extend the DCG so that it covers negated sentences such as 'Vincent does not walk' or 'It is not the case that Vincent walks'.

Exercise 2.4.5 Eliminate all `apply/3` goals from the DCG.

Exercise 2.4.6 This exercise is about Pereira & Shieber's trick to provide a simple semantics for transitive verbs, i.e., `lambda(X,lambda(Y,love(X,Y)))` for the verb 'love'. The second part of the trick is to change the grammar rule that deals with transitive verbs:

```
vp(lambda(X,VP))-->
   tv(lambda(X,TV)),
   np(NP),
   {
    apply(NP,TV,VP)
   }.
```

Explain why this works and discuss this particular treatment of transitive verbs.

A Substitution-Based Implementation

Our previous implementation of functional application and β-conversion made 'clever' use of Prolog unification to implement. Unfortunately, although its simplicity is striking, the implementation is not fully correct, and it fails for reasons directly relevant to semantic construction. We will go through an example to make the problem clear, and then implement functional application and β-conversion properly. .

Our previous implementation is unable to hand *coordination* correctly. Consider the sentence 'Vincent and Mia dance'. We already know what the lambda expressions are for 'Vincent', 'Mia', and 'dance'. We represent 'and' by the following lambda expression:

(2) $\lambda X.\lambda Y.\lambda P.(X(P) \wedge Y(P))$.

This is a sensible representation. If we apply it to the representations for 'Mia' and 'Vincent' we get the following result:

(3) λP.((λQ.Q(VINCENT))(P)$\wedge$(λR.R(MIA))(P)).

Applying β-conversion twice yields:

(4) λP.(P(VINCENT)$\wedge$P(MIA)).

That is, one and the same lambda expression P is to be applied to both VINCENT and MIA. And this is exactly what we need. In particular, applying this expression to DANCE yields the formula DANCE(VINCENT)$\wedge$DANCE(MIA), which is the representation we would like to have.

So far so good. But now consider what happens when turn this into Prolog. First, the lexical entry for the coordinator 'and' will be as follows:

```
coord(lambda(X,lambda(Y,lambda(P,F1 & F2))))-->
   [and],
   {
    apply(X,P,F1),
    apply(Y,P,F2)
   }.
```

Next, assume that we want to analyze 'Vincent and Mia dance' and that in the process of combining the lexical entries `X` unifies with the semantic representation of 'Vincent', `Y` with the semantic representation of 'Mia'. Further, assume that `P` gets instantiated with `lambda(X,dance(X))`. Then the two `apply/3` statements, which both use a copy of `P`, try to force the occurrence of `X` to unify with both `mia` *and* `vincent`. As distinct atoms do not unify, this fails.

Exercise 2.4.7 Try to cover verb phrase coordination cases like 'Vincent walks and talks' in the unification-based approach. Does the unification problem that appears in noun phrase coordination arise here as well? Give an explanation for your observation.

So clearly we can't hope to use Prolog unification directly to cover all cases of functional application; what we need is β-conversion predicate that genuinely substitutes the argument expression for all free occurrences in the scope of the lambda expression, and keeps track of the bound and free variables in a formula.

In `comsemLib.pl` there is a version of the Sterling and Shapiro `substitute/4` predicate. This predicate takes a term, a variable, and a formula as its first three arguments, and

returns in its fourth argument the result of substituting the term for each free occurrence of the variable in the formula. This is an important predicate (we shall use it again when we implement a first-order theorem prover) so let's look at the way it's defined:

```
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]),
   memberOfList(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).
```

The `substituteList/4` predicate used in here recursively applies `substitute/4` down the length of the input list to produce the output list:

```
substituteList(_Term,_Var,[],[]).
substituteList(Term,Var,[Exp|Others],[Result|ResultOthers]):-
   substitute(Term,Var,Exp,Result),
   substituteList(Term,Var,Others,ResultOthers).
```

And that's it. Neat, but can we use this to implement β-conversion? The most obvious approach to our problem is simply to redefine `apply/3` using this predicate:

```
apply(lambda(X,Formula),Argument,Result):-
    substitute(Argument,X,Formula,Result).
```

Unfortunately, this doesn't work. Some of our lexical entries contain explicit `apply/3` goals, which is unproblematic when using unification (remember that it is possible to eliminate all `apply/3` goals in the grammar) but a disaster when it has to do real work, as our new

approach requires. As the example entry for 'and' above illustrates, it makes no sense to prove the two apply/3 goals until the variables X and Y are instantiated.

But this suggests a solution: we shall 'freeze' all apply/3 goals until the sentence is parsed. That is, we shall introduce a new Prolog operator @ to explicitly mark where functional application is to take place: the notation f @ a will mean apply function f to argument a. We will build up our representations using these explicit markings, and delay carrying out the required substitutions until all the required information is to hand. (Incidentally, this is essentially the strategy used in textbook presentations of Montague semantics: students are usually taught to first build the representation, and then to simplify.) Here is our new definition of apply/3:

```
apply(Functor,Argument,Functor @ Argument).
```

And here are some examples of lexical entries where the apply/3 goals are directly replaced by @:

```
det(lambda(P1,lambda(P2,forall(X,(P1 @ X) > (P2 @ X)))))-->
   [every].

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

This makes semantic construction during parsing extremely easy: we simply use @ to record the required function/argument structure. Here is an example query:

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

Sem = lambda(P,P@mia)@lambda(X,snort(X))
```

On the other hand, we need to do more work after parsing, for we want to reduce these complicated lambda expression into readable first-order formulas by carrying out (genuine) β-conversion. But this is now straightforward. Because all the required information is in place, we can use substitute/4, and with the help of this predicate, we can define β-conversion correctly. The following code does this:

```
betaConvert(Var,Result):-
   var(Var), !, Result=Var.
betaConvert(Functor @ Arg,Result):-
   compound(Functor),
   betaConvert(Functor,lambda(X,Formula)), !,
```

```
    substitute(Arg,X,Formula,BetaConverted),
    betaConvert(BetaConverted,Result).
betaConvert(Formula,Result):-
    compose(Formula,Functor,Formulas),
    betaConvertList(Formulas,ResultFormulas),
    compose(Result,Functor,ResultFormulas).
```

The first clause of of `betaConvert/2` simply records the fact that variables cannot be further reduced. The second clause does the most important things: it checks whether the functor is complex term and then reduces it to a lambda expression (of course it may already be a suitable lambda expression, but it could perfectly well be an application that first has to be reduced). If that succeeds, substitution of the argument for all free occurrences of `X` in `Formula` takes place, and this intermediate result is reduced as well. The third and final clause breaks down formulas and predicates and reduces their arguments or subformulas. This is done with the help of:

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

To finish off, we design a driver predicate that calls the parser to analyze a sentence, and to reduce the resulting lambda expression into a first-order formula. The result is directed to the standard output.

```
parse:-
    readLine(Sentence),
    s(LambdaExpression,Sentence,[]),
    betaConvert(LambdaExpression,Formula),
    printRepresentation(Formula).
```

Exercise 2.4.8 Check the new grammar rules using β-conversion, especially on cases of noun and verb phrase coordination.

Exercise 2.4.9 Note that we still use Prolog variables to represent first-order variables, although this is not important for the application predicates as we could use basic atoms instead. Why is it still advantageous? Hint: lexical entries.

2.5 Grammar Engineering

The explicit notation for functional application and the implementation of β-conversion are the basic tools we shall work with in this book, so it is time to define a bigger grammar and

start exploring computational semantics; but let's try to observe some basic principles of grammar engineering as we do so. That is, we should strive for a grammar that is *modular* (each component should have a clear role to play and a clean interface with the other components), *extendible* (it should be straightforward to enrich the grammar should the need arise) and *reusable* (we should be able to reuse a significant portion of the grammar, even when we change the underlying representation language).

Grammar engineering principles have strongly influenced the design of our grammars — and *not*, we would like to stress, purely for pedagogic reasons. We will be experimenting with a wide variety of semantic constructions techniques (for example, in the following chapter we will consider four different techniques for coping with scope ambiguities). Moreover, in Part II we will switch from first-order logic to Discourse Representation Structures as our underlying representation language. As we have learned (often the hard way) incorporating these changes, and keeping track of what is going on, requires a disciplined approach towards grammar design.

We have adopted a fairly simple three-level grammar architecture consisting of a collection of semantically annotated *rules*, a *lexicon*, and what the *rule/lexicon interface*, which we usually simply call the *interface*. The rules are DCG rules annotated with an extra slot which applies semantic functions to arguments using the `apply/3` predicate. Although we shall sometimes add additional rules, the rules given below and their annotations will *not* change in the course of the book; we will reuse them when we work with Discourse Representation Theory in Part II. The lexicon lists information about words belonging most syntactic categories in an easily extractable form; again, this component will stay fixed throughout the book. Finally, we have the crucial interface. This is level at which we state what we have previously called 'lexical entries'. It is here that we will do most of our semantic work, and our modifications will largely be confined to this level.

The Rules

Here are the core DCG rules that we would *like* to use. These rules license a number of semantically important constructions, such as proper names, determiners, pronouns, relative clauses, the copula construction, and coordination.

```
d --> s, d.
d --> s.
s --> np, vp.
np --> np, coord, np.
np --> det, nbar.
np --> pn.
np --> pro.
nbar --> nbar, coord, nbar.
nbar --> noun.
nbar --> noun, rc.
noun --> noun, coord, noun.
vp --> vp, coord, vp.
vp --> aux, vbar.
vp --> [is,a], nbar.
vbar --> vbar, coord, vbar.
vbar --> iv.
vbar --> tv, np.
rc --> relpro, vp.
```

```
nbar --> noun, pp.                    pp --> prep, np.
```

(Readers unfamiliar with syntax may be wondering what NBAR and VBAR are. Essentially they are a level of phrase structure intermediate between the full phrasal level (NP and VP respectively) and the lexical level (N and V). There is a variety of evidence for the existence of this intermediate level and we refer the reader to any mainstream syntax textbook for further discussion. We introduce this intermediate level because it will be the source of some nice examples in Part II.)

However these are not quite the rules we shall actually use, for the following reason. The coordination rules are left-recursive, hence the standard Prolog DCG interpreter will loop when given this grammar. As we *do* want to give coordination examples, and as we *don't* want to implement a better parser, we're simply going to adopt the following simple fix which will make a limited form of coordination available to us. We'll add a feature `+` (which can be read as "can be coordinated") to the categories we want to coordinate. For example, the rules which have something to say about NPs will be replaced by the following:

```
s --> np(+), vp(+).
np(+) --> np.
np(+) --> np, coord, np.
np --> det, nbar(+).
np --> pn.
np --> pro.
```

Similarly, the rules controlling NBAR will become:

```
nbar(+) --> nbar.
nbar(+) --> nbar, coord, nbar.
nbar --> noun(+).
nbar --> noun(+), pp.
nbar --> noun(+), rc.
```

This is a brutal way of dealing with the problem, but it enables us to generate the examples we want without having to worry about left-recursion.

One other shortcoming should be mentioned. We have not implemented any inflectional morphology — all our examples are relentlessly third-person present-tense. This is a shame (tense and its interaction with temporal reference is a particularly rich source of semantic examples), nonetheless, we shall not be short of interesting things to do.

But for all its shortcomings, this small set of rules assigns tree structures to an interesting range of English sentences or small discourses, including:

Mia knows every owner of a hash bar.
Vincent or Mia dances.
Every boxer that kills a criminal loves a woman.
Vincent does not loves a boxer or criminal that snorts.
She does not loves a boxer or criminal that snorts and dances.
A boxer growls. He collapses.

Let's turn to the semantic annotations. Here the news is extremely pleasant. For a start, the required semantic annotations are utterly straightforward; they are simply the obvious "apply the function to the argument statements" expressed with the help of `apply/3`. Here, for example, is how we semantically annotate `s --> np(+), vp(+).`:

```
s(S)--> np(+,NP), vp(+,VP), {apply(NP,VP,S)}.
```

That is, we simply apply the NP semantics to the VP semantics. None of the rules are much more complex than this. The rules for coordination are the most complex, but even these are straightforward as the following example shows:

```
np(+,NP)-->
   np(NP1),
   coord(Coord),
   np(NP2),
   {
    apply(Coord,NP1,Sem),
    apply(Sem,NP2,NP)
   }.
```

However, not all branching rules use `apply/3` as semantic operation to combine the semantics of their daughters. For some rules, such as conjoining two sentences of a discourse, functional application is *not* the underlying semantic operation. As this differs from the choice of representation language, we introduce a predicate `conjoin/3` that abstracts way from representations and use that in the rules for discourses:

```
d(C)--> s(S), d(D), {conjoin(S,D,C)}.
```

But what should `conjoin/3` do? For first-order representations, we simply combine the two formulas by conjuncting them (but in Part II of the book, where we work with discourse representations, we will change this definition).

```
conjoin(Sem1,Sem2,Sem1 & Sem2).
```

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

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

Finally, we've got the lexical rules. These are rules that apply to terminal symbols, the actual strings in the input of the parser, and need to call the lexicon to check if a string belongs to the syntactic category searched for.

```
noun(Noun)-->
   {
    lexicon(noun,Relation,Phrase,_),
    nounSem(Relation,Noun)
   },
   Phrase.
```

In this example for nouns, the semantic macro `nounSem` is used to construct the actual semantic representation for a noun. Each lexical category is associated with such a macro, and this enables us to abstract away from specific types of structures. So we have set up the grammar rules in such a way that we are independent from the semantic theory we want to work with, and this is an attractive property as we are going to change the underlying semantic representations in the following chapters.

All these rules will work for us unchanged throughout the book. The reader can find a complete list in the file `englishRules.pl`. Furthermore, the files `unificationInterface.pl` and `substitutionInterface.pl`, that contain the unification- and substitution-based approaches to the lambda calculus of this chapter, already use these rules in the way described here.

The Lexicon

Our lexicon lists information about the words belonging to most syntactic categories in a form useful for the interface component that we shall shortly define. The general format of a lexical entry is

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

where `Cat` is the syntactic category, `Sem` the semantic information introduced by the phrase (normally a relation symbol or a constant, though sometimes we leave this field empty), `Phrase` the string of words that span the phrase, and `Misc` miscellaneous information

depending on the type of entry. In particular, `Misc` lists gender information for nouns, pronouns, and proper names (we are going to classify nouns as `male`, `female` or `neuter`, for this information will be important for pronoun resolution in Part II).

Typical entries for intransitive verbs are:

```
lexicon(iv,collapse,[collapses],_).
lexicon(iv,dance,[dances],_).
```

Nouns, on the other hand, are listed in the following format:

```
lexicon(noun,boxer,[boxer],male).
lexicon(noun,bkburger,[big,kahuna,burger],neuter).
```

Typical entries for pronouns and determiners are as follows:

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

lexicon(pro,_,[he],male).
lexicon(pro,_,[she],female).
```

Note that these entries contain no semantic information. This is because the semantic contribution of pronouns and determiners is not simply a constant or relation symbol, but rather a relatively complex expression that is dependent on the underlying representation language (quantifiers are dealt with very differently in the DRT based work of Part II, for example). Hence we shall specify the semantics of these categories in the interface.

A small number of important words — in particular, copula and the verb phrase modifier construct '`does not`', are *not* listed in the lexicon at all. This is because they are not associated with either a relation symbol or a constant and they are not marked for gender, thus their lexicon entry would simply consist of a `Phrase` entry. The interface is the sole source of information about such words.

This way of setting up a lexicon offers natural expansion options. For example, if decided to develop a grammar that dealt with inflectional morphology (which we won't do so in this book), it is just a matter of extending the general format of entries with one or more fields, and if we did this, it would be natural to list all words in the lexicon.

Refer to `englishLexicon.pl` for a complete listing of the lexicon.

The Interface

We now come to the most important part of the grammar, the lexicon/rules interface. Essentially, this component is where we state what we called semantic macros in the text. As the introduction of `apply/3` and `conjoin/3` has reduced the process of combining semantic representations to an elegant triviality, and as the only semantic information the lexicon supplies is the relevant constant and relation symbols, the interface is where the real semantic work will be done. Let's consider some examples of semantic macros right away.

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

pnSem(Constant,lambda(P,P @ Constant)).

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

The first macro, `nounSem/2`, builds a semantic representation for any noun given the relation symbol `Relation`, turning this into a formula lambda abstracted with respect to a single variable (for example `lambda(X,Formula)`). The representation is built using `compose/3` to incorporate it into a the required lambda expression. The semantic macro for proper names (`pnSem/2` is even simpler, and the one for transitive verbs (`tvSem/2` is similar to that of the macro for nouns, apart from handling two variables rather than just one.

As we've already mentioned, the interface also contains self-contained entries for the determiners. Here they are:

```
detSem(uni,lambda(P1,lambda(P2,forall(X,(P1 @ X) > (P2 @ X))))).

detSem(indef,lambda(P1,lambda(P2,exists(X,(P1 @ X) & (P2 @ X))))).
```

These, of course, are just the old-style 'lexical entries' we are used to.

From now on, we will always use the lexicon and (with the exceptions of a few pedagogic explorations of other formats) the rules listed above too. To put it another way: *from now on the primary locus of change will be the interface.* For example, it is here that we will develop treatments of quantifier scope, pronoun resolution, presupposition accommodation, and VP-ellipsis. For a complete listing of the interface we have just been discussing, see `substitutionInterface.pl`, but we shall see many more interfaces as we work our way through the book.

Software Summary of Chapter 2

`experiment1.pl` The code of our first experiment in semantic construction for a small fragment of English. (page 304)

`experiment2.pl` The second experiment in semantic construction. (page 307)

`unificationInterface.pl` This interface carries out semantic construction, using our standard rules and lexicon, that makes use of the (simple but incorrect) unification-based implementation of β-reduction. (page 309)

`betaConversion.pl` The substitution-based implementation of β-conversion. (page 311)

`substitutionInterface.pl` Semantic construction, using our standard rules and lexicon, that makes use of the substitution-based implementation of beta-conversion. (page 312)

`englishLexicon.pl` Our standard English lexical entries. Contains entries for nouns, proper names, intransitive and transitive verbs, prepositions, and pronouns. (page 294)

`englishRules.pl` Our standard grammatical rules for a fragment of English. Rules cover basic sentences, noun phrases, relative clauses and modification of prepositional phrases, verb phrases, and a limited form of coordination. (page 298)

Further Reading

Compositionality is a simple and natural idea — and one capable of arousing a enormous amount of passion and controversy. Traditionally attributed to Gottlob Frege (the formulation "the meaning of the whole is a function of the meaning of its parts" is often called Frege's principle) it received a precise mathematical formulation in the late 1960s. For detailed and accessible overview of the compositionality concept, the reader should consult Janssen 1997.

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

Montague's original papers are well worth reading (they are collected in Thomason 1974) nonetheless they are dense and most readers will be better off approaching Montague semantics via either Dowty, Wall and Peters (1981) or volume 2 of Gamut (1991b). Both contain good, careful, textbook level expositions of Montague's key ideas. A more demanding (and correspondingly more rewarding) exposition is due to Janssen (see Janssen 1986a and Janssen 1986b). These are well worth reading, but rather hard to get hold of. If you see them, grab them. For a good overview of Montague's work, and an account of the major directions in which his work has been developed, Partee 1997 is a must. This account is rich in historic detail, and is an excellent starting point for deeper forays into the semantic literature.

The mathematical and logical literature on the lambda calculus is huge. Here are some of the more obvious points of entry. The bible of untyped lambda calculus is Barendregt 1984. Barendregt 1991 is a good starting point for material on typed systems. However these are very technical accounts, perhaps best suited for occasional reference. For a more approachable account of both typed and untyped systems, try Hindley and Seldin 1986. Turner 1997 is fairly technical, but usefully broad, and notes applications in linguistics and computer science. Finally, for logical work on the systems Montague himself used, and a number of interesting variants, Gallin 1975 is indispensable.

The references given so far approach lambda calculus and its applications in natural language semantics from a logical perspective. In this book we have tried to emphasize that there is a very natural computational perspective too. Not only is the lambda calculus a useful tool for gluing representations together, but the basic ideas emerges, with seeming inevitability, when one sits down and actually tries to do semantic construction in Prolog — or at least, that is what we have tried to suggest by approaching lambda calculus via experiments 1 and 2. We hasten to add that this 'seeming inevitability' is clear only with the benefit of hindsight. The links between the ideas of logic programming and Montague semantics seem to have first been explicitly drawn in Pereira and Shieber 1987.

Lambda calculus is probably the mostly widely used tool for semantic construction, and it is the tool of choice in this book — nonetheless, it is *not* the only tool available. Indeed, it has a most interesting rival: the use of *feature structures* and *feature structure unification*. The reader who wants to go further in computational semantics really should be acquainted with this approach. Gazdar and Mellish 1989 contains a good text book level introduction to its use in building syntactic and semantic representations, and indeed, on its uses in pragmatics too. Another readable paper on the topic is Moore 1989. This paper compares the approach with the use of lambda calculus. More recently, linear logic has been proposed as a suitable glue language for semantic construction; see Dalrymple et al. 1997.

Chapter 3

Underspecified Representations

This chapter develops methods for dealing with an important semantic phenomenon: *scope ambiguities*. Sentences with scope ambiguities are often semantically ambiguous (that is, they have at least two non-equivalent first-order representations) but fail to exhibit any syntactic ambiguity. As our approach to semantic construction is firmly based on the idea of using syntactic structure to guide semantic construction, we face an obvious problem here: if there is no syntactic ambiguity, we will only be able to build one of the possible representations. As scope ambiguities are very common, we need to develop ways of coping with them right away.

We are going to investigate and implement four different approaches to scope ambiguities: Montague's original method, two storage based methods, and finally a more recent approach based on the idea of *underspecified representations*. Each approach is interesting in its own right, and the underspecification-based approach we shall eventually describe is a powerful tool that we shall use throughout the book. But as well as developing practical solutions to a pressing problem, this chapter also tells an important story. Computational semanticists are adopting an increasingly abstract perspective on what representations are and how they should be built. Once we have studied the evolutionary line leading from Montague's method to contemporary underspecification-based methods, we will be in a better appreciate to appreciate why.

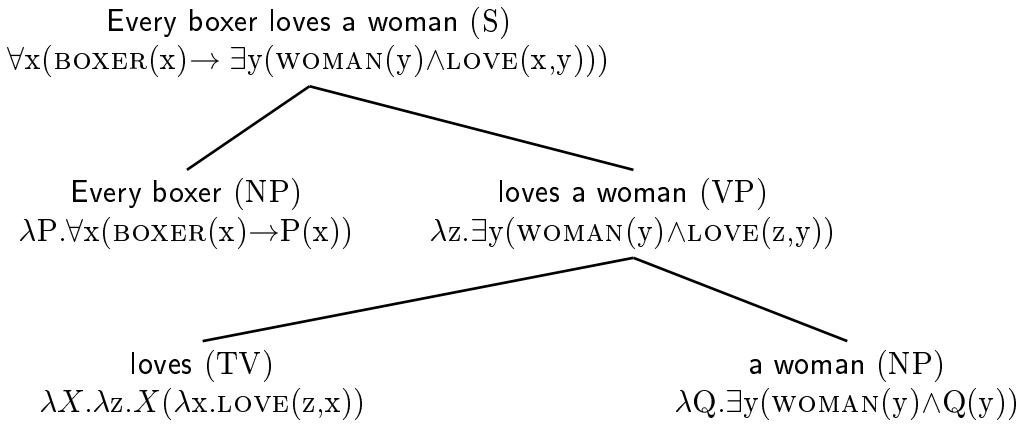
3.1 Scope Ambiguities

Scope ambiguity is a common phenomenon and can arise from a wide variety of sources. In this chapter we will mostly be concerned with *quantifier scope ambiguities*. These are ambiguities that arise in sentences containing more than one quantifying noun phrase; for example, 'Every boxer loves a woman'.

Now, the methods of the previous chapter allow us to assign a representation to this

sentences as follows:

(1)



The first-order formula we have constructed states that for each boxer there is a woman that he loves; there might be different women for different boxers. However, 'Every boxer loves a woman' has a second meaning (or to use the linguistic terminology, a second *reading*) that is captured by the following formula:

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

This says that there is one woman who is loved by all boxers.

It is clear that these readings are somehow systematically related, and that this relation has something to do with the relative scopes of the quantifiers: both first-order representations have the same components, but somehow the parts contributed by the two quantifying noun phrases have been shuffled around. In the first representation the existential quantifier contributed by 'a woman' has ended up inside the scope of the universal quantifier contributed by 'every boxer' and in the second representation the nesting is reversed. In fact it is usual to say that in the first reading 'every boxer' *has scope over* (or *outscopes*) 'a woman', while in reading (2) it is the other way around.

Unfortunately, these scoping possibilities are not reflected syntactically: the only plausible parse tree for this sentence is the one just shown. Thus while it makes good semantic sense to say that in reading (2) 'a woman' outscopes 'every boxer', we can't point to any syntactic structure that would explain why this scoping possibility exists. As each word in the sentence is associated with a fixed lambda expression, and as semantic construction is simply functional application guided by the parse tree, this means there is no way for us to produce this second reading. This difficulty clearly strikes at the very heart of our semantic construction methodology. Moreover, scope ambiguities are extremely common, so we urgently need a solution.

In this chapter we examine in detail four increasingly sophisticated (and increasingly abstract) approaches to the problem. The first of these, Montague's original method, introduces some important ideas, but as it relies on the use of additional rules it isn't compatible with the approach to grammar engineering adopted in this book. However by introducing a more abstract form of representation — the *store* — we will be able to capture Montague's key ideas. Stores were introduced by Robin Cooper and are arguably the earliest example example of *underspecified representations*. Nonetheless, stores are relatively concrete. By simultaneously moving to more abstract underspecified representations, and replacing the essentially generative perspective underlying storage methods with a constraint based perspective, we arrive at *hole semantics*, the underspecified representation we shall use in this book. This will enable us to handle quantifier scope ambiguities, and the scope ambiguities created by constructs such as negation, in a uniform way.

Exercise 3.1.1 Some readers may be wondering whether scope ambiguities really are a genuine problem at all. Consider 'Every boxer loves a woman'. In a sense, representation (1) is sufficient to cover both readings of our example, since it is entailed by the other reading (2). Arguably, that makes the stronger representation superfluous: perhaps it inferred from the weaker one with the help of contextual knowledge. So do we really need techniques for coping with quantifier ambiguity?

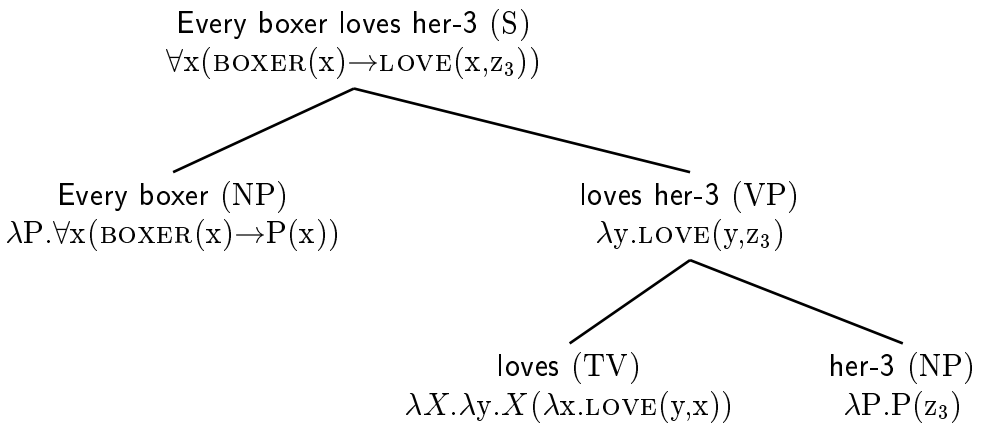
We do, as we want the reader to demonstrate here. First, find examples of quantifier scope ambiguities that give rise to logically *independent* readings (that is, neither implies the other). Secondly, find an example where direct construction doesn't lead to the weakest reading (that is, the reading entailed by the others). Feel free to use determiners such as 'one', 'many' and 'most' to construct your examples.

3.2 Montague's Approach

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

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

(3)

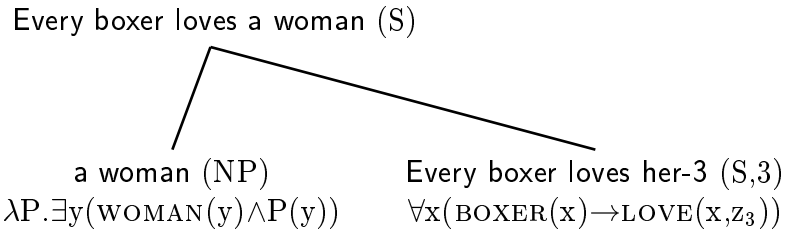


Instead of combining 'loves' with the quantifying term 'a woman' we have combined it with the placeholder pronoun 'her-3'. This pronoun bears an 'index', namely the numeral '3'. The placeholder pronoun is associated with a 'semantic placeholder', namely $\lambda P.P(z_3)$. As we shall see, it is the semantic placeholder that does most of the real work for us. Note that the pronoun's index appears as subscript on the free variable in the semantic placeholder. From a semantic perspective, choosing an indexed pronoun really amounts to opting to work with the semantic placeholder (instead of the semantics of the quantifying NP) and stipulating which free variable the semantic placeholder should contain.

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

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

(4)



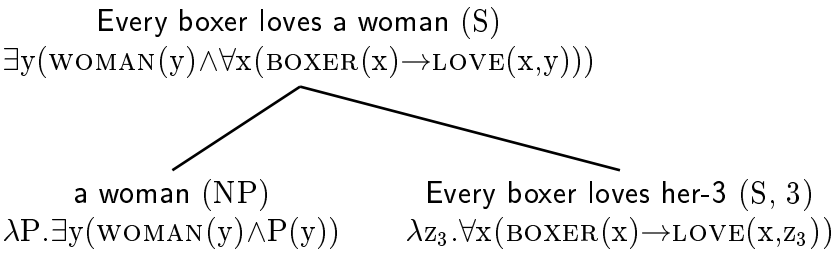
But what's happening semantically? We know what formula we want to be assigned to the top S node (namely, the formula 2) but how can we ensure it gets there? Let's think the matter through.

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

(5) $\lambda z_3.\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,z_3))$

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

(6)



That's it. Summing up, Montague's approach makes use of syntactic and semantic placeholders so that we can place quantifying NPs in parse trees at exactly the level required to obtain the desired scope relations. A neat piece of 'lambda programming' (we call it

Montague's trick) ensures that the semantic information recorded by the placeholder is re-introduced into the semantic representation correctly.

Enough theory; can we write a Prolog program which integrates the previous chapter's work on lambda-based semantic construction with quantifier raising? Up to a point, we can. Unfortunately, most of the work is going to have to take place in our DCG rules, and the result is not particularly attractive from a grammar engineering perspective. Still, let's go through it systematically; it's interesting, and we shall be able to re-use some ideas and notation in our later work on storage. To keep the code as simple as possible we shall work with our (slick but faulty) unification-based implementation of β-conversion.

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

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

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

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

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

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

```
     apply(SemDet,SemNoun,Sem)
   }.

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

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

```
vp([],Sem)-->
   iv(Sem).

vp(Q,SemVP)-->
   tv(SemTV),
   np(Q,SemNP),
   {
    apply(SemTV,SemNP,SemVP)
   }.
```

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

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

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

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

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

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

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

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

no.
```

It gives us exactly the two readings that we want.

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

Exercise 3.2.1 Extend the quantifier raising analysis to ditransitive verbs, and check how many readings it gives for sentences like 'A boxer gives every woman a foot massage'. Are these the readings you would expect?

3.3 Storage Methods

Storage methods are an elegant way of coping with quantifier scope ambiguities: they neatly decouple scope considerations from syntactic issues, making it unnecessary to add new grammar rules. Moreover, both historically and pedagogically, they are the natural approach to explore next, for they draw on the key ideas of Montague's approach (in essence, they exploit semantic placeholders and Montague's trick in a computationally more natural way) and at the same time they anticipate key themes of modern underspecification-based methods.

Cooper Storage

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

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

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

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

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

The crucial thing to note is that the index associated with ϕ is identical with the subscript on the free variable in $\lambda P.P(\mathrm{z}_i)$. (After all, if we decide to store ϕ away for later use, it's

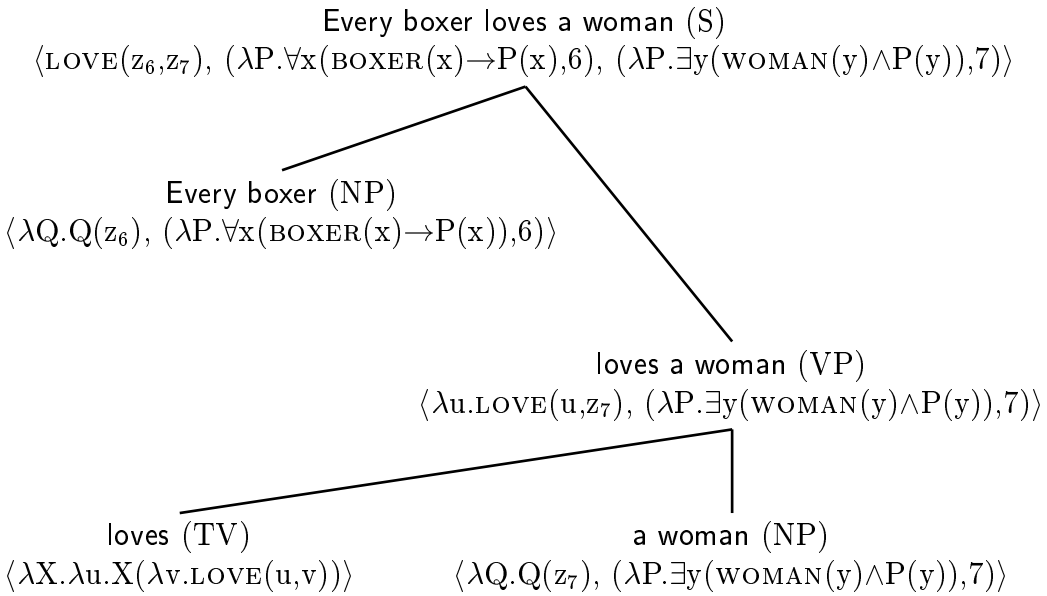
sensible to keep track of what its argument is.)

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

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

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

(7)



Note that the nodes for '**a woman**' and '**a boxer**' are both associated with 2-place stores. Why is this? Consider the node for '**a woman**'. We know from our previous work that the lambda expression associated with '**a woman**' is $\lambda P.\exists y(\mathrm{WOMAN}(y) \wedge P(y))$. In our new representations-are-stores world view, this means that the 1-place store

$\langle \lambda P.\exists y(\text{WOMAN}(y) \wedge P(y)) \rangle$

is a legitimate interpretation for the NP 'a woman'. But remember — this is not our only option. We are free to use the storage rule, and this is what we did when building the above tree. We've picked a brand new free variable (namely z_7), used the placeholder $\lambda Q.Q(z_7)$ as the first item in the store, and 'iced' $\lambda P.\exists y(\text{WOMAN}(y) \wedge P(y))$, first recording the fact that z_7 is the variable relevant to this expression. Essentially the same story could be told for the NP 'every boxer', save that there we chose the new free variable z_6.

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

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

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

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

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

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

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

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

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

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

Using lambda conversion, this simplifies to:

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

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

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

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

(11) $\langle \exists y(\textsc{woman}(y) \wedge \forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,y))\rangle$

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

Exercise 3.3.1 Show the steps involved in applying retrieval that yield the reading where '*every boxer*' has scope over '*a woman*'.

Let's now implement Cooper storage in Prolog.

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

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

But now we need to think a little. The semantic representations we want to work with are stores, not just plain lambda expressions. Moreover, we want to reuse our lexicon and rules, and the idea of combining representations with the `@` operator is hard-wired into our rules. But we have said nothing about combining stores using `@`: do we face a problem here? No: its fairly obvious that we simply need to 'lift' our use of `@` so that we can meaningfully talk of combining stores with stores — and indeed, stores with quantifiers — using `@`. As stores are just lists of lambda expressions and binding operators, this shouldn't be too difficult to do. But one matter deserves to be stressed: *the results of these 'lifted' uses of* `@` *should always be stores.* This will guarantee that the sentence ends up being associated with a store, and hence that store-retrieval can proceed as explained above. Let's define a predicate `buildStore/2` that carries out these combinations for us:

```
buildStore(quant(Quant) @ Store,NewStore):-
   buildStore(Store,[Arg|S]),
   npStorage(Quant,[Arg|S],NewStore).

buildStore(Store1 @ Store2,[F@A|S]):-
   buildStore(Store1,[F|S1]),
   buildStore(Store2,[A|S2]),
   appendLists(S1,S2,S).

buildStore([Sem|Store],[Sem|Store]).
```

This predicate has three clauses. Now, the second clause simply combines two stores using `@` in the obvious way (it makes use of the familiar fact that the first slot of the store really is where the core representation is built) and the result is clearly a new store. The third clause ends the recursion. So far so good — but what about the first clause, which lets us combine a quantifier and a store using `@`?

To understand what is going on here, let's examine the predicate `npStorage/3` that the first clause uses:

```
npStorage(Quant,[Arg|Store],[lambda(P,P@X),bo(Quant@Arg,X)|Store]).
```

That is, this predicate simply stores the quantifier in the expected way. But then it is clear that the first clause of `buildStore/2` simply says that combining a quantifier and a store using `@` simply amounts to storing that quantifier — thus once again, our extended notion of functional application yields a new store.

Two other remarks should to be made. First, storage should is a possibility that should *only* be available to quantified noun phrases, and this explains the presence of the marker `quant/1`. This marker (or feature) signals that the representation we have in our hands is of a genuine quantifier. Second, note that this version of `buildStore/2` makes storage

compulsory for quantifiers, not *optional*. This turns out not to be a good idea, and we shall we shall eventually modify our definition of `buildStore/2` accordingly — but more on that topic later.

With `buildStore/2` at our disposal, it now makes sense to talk of combining stores using `@`, and moreover we know that the result will always be a new store. So we are ready to define the interface. Here's the semantic macro for a universal determiner:

```
detSem(uni,quant(lambda(P1,lambda(P2,forall(X,(P1 @ X) > (P2 @ X)))))).
```

Apart from the `quant`-marker, there is nothing really new in this entry — indeed it looks exactly like our previous interface entries. As for the other categories, the only change we make is that the representations we work with are stores, as the following macros show:

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

pnSem(C,[lambda(P,P @ C)]).
```

Thus we have all that we need to create stores: all that remains is to implement retrieval.

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

```
sRetrieval([S],S).
sRetrieval([Sem|Store],S):-
   removeFromStore(bo(Q,X),Store,NewStore),
   sRetrieval([Q@lambda(X,Sem)|NewStore],S).
```

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

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

The driver is transparent: we use `buildStore` to build a store, `sRetrieval` to generate the various representations, and then the familiar β-conversion predicate to simplify:

```
parse:-
   readLine(Sentence),
   s(Sem,Sentence,[]),
   setof(Result,Store^Retrieved^(buildStore(Sem,Store),
                                 sRetrieval(Store,Retrieved),
                                 betaConvert(Retrieved,Result)),
         SemSet),
   printReadings(SemSet).
```

Again, it is time to test our program:

```
?- parse.

> Every boxer loves a woman.

Readings:
1 exists(A,woman(A)&forall(B,boxer(B)>love(B,A)))
2 forall(A,boxer(A)>exists(B,woman(B)&love(A,B)))
```

Unsurprisingly, these are exactly the results we want.

Summing up, Cooper storage is a generalization of Montague's rule of quantification that makes use of abstract representations called stores. From the perspective of computational semantics, it has a distinct advantage over Montague's approach: we don't need to to make use of additional grammar rules. Indeed, when you get down to it, all we really needed to do to implement Cooper storage was to define what it meant to combine our new representations using `@` (this is what `buildStore/2` did for us) and then provide two additional predicates, `npStorage` and `sRetrieval`, which gave us the ability to manipulate our new representations in ways that plain old functional application couldn't. Cooper storage is indeed a natural and useful technique.

Exercise 3.3.2 How many scoped representations are retrieved for 'every piercing that is done with a needle is okay'? (Take 'is done with' as a two place relation, and view 'is okay' as a one place predicate.) Are they all correct?

Exercise 3.3.3 What does the store (before retrieval) for 'every piercing is done with a needle' look like? And after retrieval?

Exercise 3.3.4 In the original discussion of Cooper storage the storage of quantifiers was an *optional* alternative to functional application. In our implementation, all quantified NPs were forced to use storage. Why is there no difference between the two strategies for the given fragment of grammar. Can you think of natural language examples where it actually does differ?

Keller Storage

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

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

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

(13) Mia knows every owner of a hash bar

Both examples give rise to scope ambiguities. For example, in (13) there is a reading where Vincent knows all owners of (possibly different) hash bars, and a reading where Vincent knows all owners that own one and the same hash bar. Moreover, both examples contain nested NPs. In the first example '**a gun**' is a sub-NP of '**every piercing that is done with a gun**', while in the second, '**a hash bar**' is a sub-NP of '**every owner of a hash bar**'.

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

$$\langle \textsc{know}(\textsc{mia},z_2),\ (\lambda P.\forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1)\rightarrow P(y)),2),\ (\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q(x)),1)\rangle$$

Exercise 3.3.5 Verify this.

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

(14) $\langle \forall y(\textsc{owner}(y)\wedge \textsc{of}(y,z_1)\rightarrow \textsc{know}(\textsc{mia},y)),\ (\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q(x)),1)\rangle$

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

(15) $\langle \exists x(\textsc{hashbar}(x)\wedge \forall y(\textsc{owner}(y)\wedge \textsc{of}(y,x)\rightarrow \textsc{know}(\textsc{mia},y)))\rangle$

This states that there is a hash bar of which Vincent knows every owner. This is one of the readings we would like to have. So let's explore the other option. If we pull the existential quantifier from the S store first we obtain:

(16) $\langle \exists x(\text{HASHBAR}(x) \wedge \text{KNOW}(\text{MIA}, z_2)), (\lambda P.\forall y(\text{OWNER}(y) \wedge \text{OF}(y, z_1) \rightarrow P(y)), 2) \rangle$

Pulling the remaining quantifier off the store then yields:

(17) $\langle \forall y(\text{OWNER}(y) \wedge \text{OF}(y, z_1) \wedge \exists x(\text{HASHBAR}(x) \wedge \text{KNOW}(\text{MIA}, y))) \rangle$

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

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

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

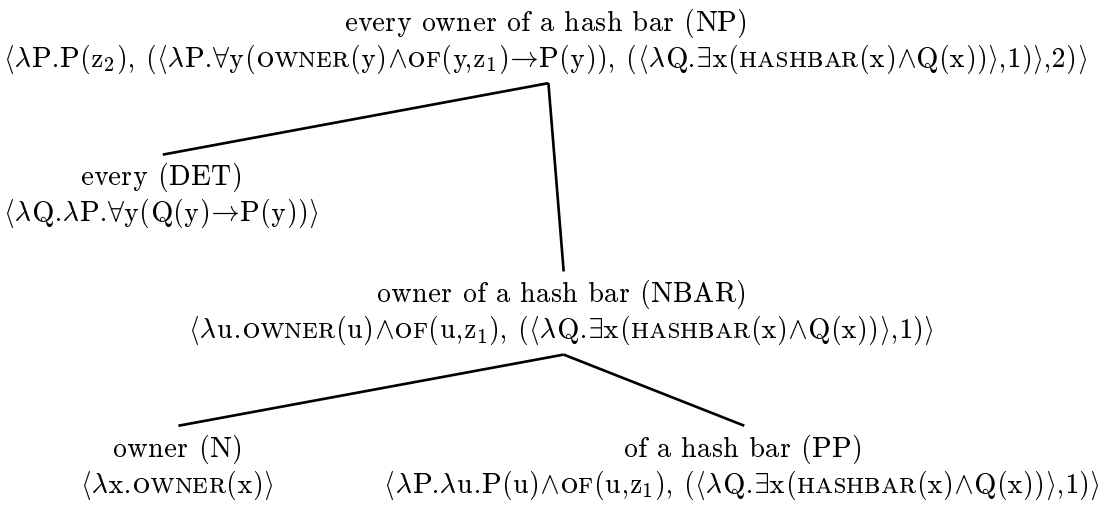
But this isn't a very principled; it deals with the symptoms, not the cause. The heart of the problem is that Cooper storage rides roughshod over the hierarchical structure of NPs; we should try to deal with this head on. (Incidentally, arguably there's also an empirical problem: the free variable solution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

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

Here's the new storage rule:

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

To see how this new storage rule works, consider how we assemble the representation associated with the complex noun phrase 'every owner of a hash bar'.



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

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

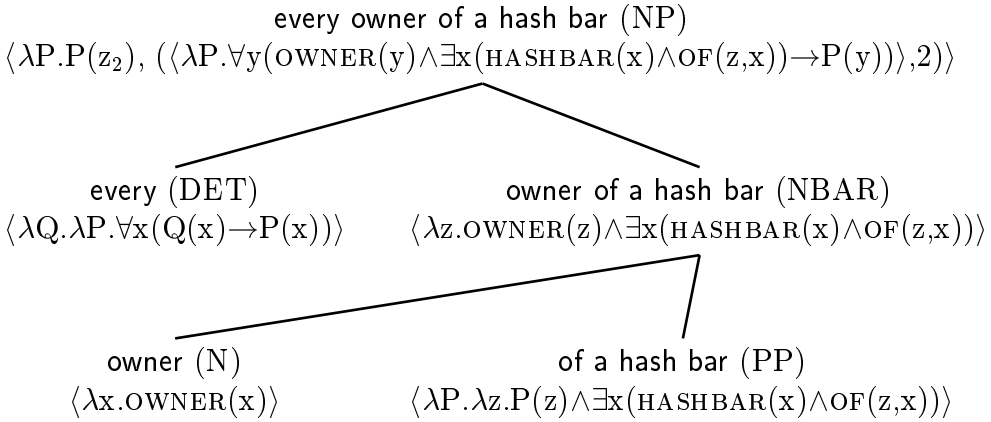
The new retrieval rule ensures that any operators stored while processing β become accessible for retrieval only after β itself has been retrieved. Nesting automatically overcomes the problem of generating readings with free variables. To see how it works in practice, let's return to our original example.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

$$\langle \text{KNOW}(\text{MIA},z_2), (\langle \lambda P.\forall y(\text{OWNER}(y)\wedge\text{OF}(y,z_1)\rightarrow P(y)), (\langle \lambda Q.\exists x(\text{HASHBAR}(x)\wedge Q(x))\rangle,1)\rangle,2)\rangle$$

There is only one way to perform retrieval: first pulling of the universal quantifier, followed by the existential quantifier, resulting in (15). Since this is the only possibility, the unwanted reading (17) is not generated.

But wait a minute: how do we get the reading where Mia knows all owners of possible different hash bars? In fact, this couldn't be easier. All we have to do is avoid storing the sub-NP 'a hash bar'. If we do this, we can produce the following tree:

(18)



This leads to the following analysis for 'Mia knows every owner of a hash bar':

(19) $\langle \text{KNOW}(\text{MIA},z_2),(\langle \lambda P.\forall y(\text{OWNER}(y)\wedge\exists x(\text{HASHBAR}(x)\wedge\text{OF}(y,x))\rightarrow P(y))\rangle,2)\rangle$

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

(20) $\langle \forall y(\text{OWNER}(y)\wedge\exists x(\text{HASHBAR}(x)\wedge\text{OF}(y,x))\rightarrow\text{KNOW}(\text{MIA},y))\rangle$

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

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

```
npStorage(Quant,[Arg|Store],[lambda(P,P@X),bo([Quant@Arg|Store],X)]).
```

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

```
removeFromStore(bo(O,I),[bo([O|T1],I)|T2],T):-
   appendLists(T1,T2,T).
removeFromStore(X,[Y|T],[Y|R]):-
   removeFromStore(X,T,R).
```

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

The last thing to do is thinking of a way to incorporate the optionality of the storage rule into the Prolog program. Naturally, we will implement this by making use of the following disjunctive rule.

```
buildStore(quant(Quant) @ Store,NewStore):-
   buildStore(Store,[Arg|S]),
   (
    NewStore = [Quant@Arg|S]
   ;
    npStorage(Quant,[Arg|S],NewStore)
   ).
```

Let's test our program (we'll use the `parse/0` predicate of the Cooper implementation) and see what happens:

```
?- parse.

> Mia knows every owner of a hash bar.

Readings:
1 exists(A,hashbar(A)&forall(B,of(B,A)&owner(B)>know(mia,B)))
2 forall(A,exists(B,hashbar(B)&of(A,B))&owner(A)>know(mia,A))
```

Let's sum up what we have learned. First, the original version of Cooper storage doesn't handle storage and retrieval in a sufficiently disciplined way, and this causes it to generate spurious readings (in fact, nonsensical readings) when faced with nested NPs. The problem can be elegantly cured by making use of nested stores, and implementing this idea requires only trivial changes to our earlier Prolog code. Second, storage really must be optional. If we are to generate all the required readings in examples involving nested NPs, we must be free not to store the NPs encountered low down in the hierarchy. If these changes are made, we have a flexible tool for handling quantifier scope ambiguities.

Exercise 3.3.6 Give the PL translations for the five readings of 'a man likes every woman with a five-dollar shake'. You might have noticed that the number of correct readings for this example is less than the combinatorial possibilities of the quantifiers that are involved. Naively, one would expect to have $3! = 6$ readings for this example. Why is one reading excluded? Try similar examples on the program.

Nonetheless, storage methods have their limitations. For a start, they are not as expressive as we might wish: although Keller storage predicts the five readings for

(21) One criminal knows every owner of a hash bar

it doesn't allow us to insist that 'every owner' must out-scope 'a hash bar', while at the same time leaving the scope relation between subject and object noun phrase unspecified. To put it another way, storage is essentially a technique which enables us to represent all possible meanings compactly; it doesn't allow us to express additional *constraints* on possible readings, and this is precisely what most modern underspecified representations let us do.

Moreover, storage is a technique specifically designed to handle *quantifier* scope ambiguities. Unfortunately, many other constructs (for example, negation) also give rise to scope ambiguities, and storage has nothing to say about these. Consider the sentence 'Vincent doesn't clean every car'. This has a reading where for each car it is not the case that Vincent cleans it; and a second, possibly preferred, reading where there are some cars (but not all) that Vincent doesn't clean. We would like an uniform approach to scope ambiguity, not a separate mechanism for each construct — and this as another motive for turning to the more abstract view offered by current work on underspecification.

3.4 Underspecification

There has been a great deal of recent interest in the use of *underspecified representations* to cope with scope ambiguities; so much so that it often seems as if semantics has entered an age of underspecification. With the benefit of hindsight, however, we can see that the idea of underspecified representations isn't really new. In our work on storage, for example, we associated stores, not simply lambda expressions, with parse tree nodes; and as we have already remarked, a store is in essence an underspecified representation. What *is* new is both the sophistication of the new generation of underspecified representations (as we shall see, they offer us a great deal of flexibility and expressive power), and, more importantly, the way such representations are now regarded by semanticists.

In the past, storage style (and other 'funny') representations seem to have been regarded with some unease. They were certainly a useful tool, but they appeared to live in a conceptual no-mans-land — not really semantic representations, but not syntax either — that was hard to classify. The key insight that underlies current approaches to underspecification is that it is both *fruitful* and *principled* to view representations more abstractly. That is, it is becoming increasingly clear that the level of representations is richly structured, that doing semantic construction elegantly demands deeper insight into this structure, and that — far from being a sign of some fall from semantic grace — semanticists should learn to play with representations in more refined ways.

In this book, we shall explore an approach to underspecification called *hole semantics*. We have chosen hole semantics because it is the approach we are most familiar with (the method is due to Johan Bos), it illustrates many of the key ideas of current approaches

to underspecification in an elegant and simple way, and we will be able to use it when we switch to working with Discourse Representation Structures. Other approaches to underspecification are noted in Further Reading at the end of the chapter.

Hole Semantics

Viewed from a distance, hole semantics shares a very obvious similarity with storage methods: at the end of the parsing stage, sentences *won't* be associated with a semantic representation. Rather, they will be associated with abstract representations from which the desired representation can be read off. Viewed closer up, however, it is clear that hole semantics adopts far more radical perspective on representations than storage does.

Hole semantics is essentially a constraint-based approach to semantic representation. That is, at the end of the parsing process the method delivers a set of *constraints*: any first-order representation which fulfills these constraints — which govern how the various bits and pieces produced during the parsing process can be plugged together — is a permissible semantic representation for the sentence. This contrasts sharply to the essentially generative approach offered by storage (*Here's the store! Enumerate the readings like this!*) and the source of much of the method's power.

Exactly what kinds constraints can we state? Answering this question is a two-step process.

1. We first need to *unplug* our underlying representation formalism (here, first-order logic) by permitting formulas to be built which contain *holes* (essentially variables over missing structure).

2. Then, with the help of such unplugged formulas, we can define a simple constraint language which governs the way semantic representations are *plugged together*.

(That is, we first break the underlying representation language down into convenient chunks, and then we define a simple constraint language governing how these chunks can be assembled into bigger units. The resulting representations we call underspecified semantic representations, USRs for short.)

Let's take care of the first-step right away. We'll now define what we mean by a PLU formula (this stands for for *Predicate Logic Unplugged*). In Part II of the book we'll unplug the language of Discourse Representation Structures in a similar way.

Suppose we have chosen some vocabulary. (We'll assume that the vocabulary is function free; this is purely for simplicity an has no bearing on the work that follows.) We must then chose a countably infinite set H of holes. These are simply any convenient set of new symbols (that is, symbols distinct from those used to build the first-order formulas) which we will usually write as h_0, h_1, h_2, h_3, ..., and so on. The hole h_0 will play a special role in the work that follows — it's used to state constraints, but it *won't* be used in PLU

formulas. So let's give a name to the set containing just the ordinary holes (that is h_1, h_2, h_3. ..., and so on) for these are the ones that are relevant at this stage. We'll use H^+ to denote this set and call it the set of *internal holes*. Note that $H = \{h_0\} \cup H^+$.

Now, what are the PLU-*terms* over this signature and H^+? Simply the ordinary first-order terms over this signature: the holes don't play any role at this level. That is, the terms we have at our disposal are simply the constants and variables.

Next, what are the *atomic PLU-formulas*? Again, simply the familiar ones; the holes don't play any role here either:

> If R is a relation symbol of arity n, and $t_1, ..., t_n$ are terms, then $R(t_1, ..., t_n)$ is an PLU-formula.

Now we are ready for the definition of PLU-*formulas*:

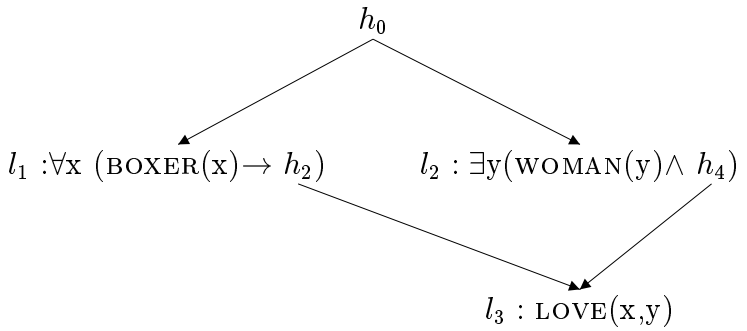
1. All atomic PLU formulas are PLU formulas.
2. If h is an internal hole, then h is a PLU formula.
3. If ϕ and ψ are PLU formulas, then $\neg\phi$, $(\phi \rightarrow \psi)$, $(\psi \vee \phi)$, $(\psi \wedge \phi)$ are PLU formulas.
4. If x is a variable, and ϕ is a PLU formula, then $\forall x\phi$ and $\exists x\phi$ are PLU formulas.
5. Nothing else is a PLU formula.

This completes the first stage in the definition of our constraint language. Let's turn to the second step. Our USRs (underspecified semantics representations) will be built using PLU-formulas, a special new symbol $\leq$, and a set of new symbols called *labels* which we will write as l_1, l_2, l_3, ..., and so on. What kind of USRs can we form? Here's an example. As we have already mentioned, sentences are going to be associated with a USR. Here's the USR associated with 'Every boxer loves a woman':

$$(22) \quad < \left\{ \begin{array}{l} l_1 : \forall x(\textsc{boxer}(x) \rightarrow h_2) \\ l_2 : \exists y(\textsc{woman}(y) \wedge h_4) \\ l_3 : \textsc{love}(x,y) \end{array} \right\}, \left\{ \begin{array}{l} l_1 \leq h_0 \\ l_2 \leq h_0 \\ l_3 \leq h_2 \\ l_3 \leq h_4 \end{array} \right\} >$$

Everything the reader needs to know about constraints can be gleaned from this example. USRs have two components. The left hand side consists of a set of of PLU-formulas preceded by a label. The right hand side consists of a set of constraints of the form $l \leq h$, where l is a label and h is a hole. Note that the special hole h_0 occurs in the right-hand component of the USR.

But what does this USR mean? Essentially it tells us how we are permitted to plug together the formulas on the left-hand side to build a first-order formula. The special hole h_0 is a variable that stands for the first-order formula that we will eventually build. This formula will be built by plugging together the three labeled PLU formulas given in the left-hand side of the USR. Plugging together simply means substituting one of of these 3 PLU formulas in another. (Incidentally, when we carry out such a substitution we only substitute the PLU formula, not its label as well. The label is simply an identifier that enables us to state the right hand side constraints succinctly). Any plugging will do — *as long as it satisfies the constraints given on the right hand side*. These constraints are essentially constraints or *subformulas* constraints, and are best thought of visually. Each arrow in the following diagram corresponds to a $\leq$ expression in an obvious way:



For example, a first order representation will only be satisfactory if the formula BOXER(x) is a subformula of the formula used to plug h_1. More interestingly, note that the the constraints are set up in such a way that the basic PLU formula LOVE(x,y) is forced to be out-scoped by the consequent of the universal quantifier's scope, and the second conjunct of the existential quantifier's scope.

Now it's time to say a little more about *resolving* these ambiguous representations. The holes underspecify scope, and in order to give us non-ambiguous interpretations, each hole should be *plugged* with a formula in such a way that all the constraints are satisfied. In other words, we should be sure that each hole gets associated with some label. Obviously, no label can be plugged into two different holes at the same time. Therefore, a so-called *plugging* is a one-to-one correspondence from holes to labels (i.e., a bijective function, with the set of holes as domain and the set of labels as codomain). A plugging for a proper USR is admissible if the instantiations of the holes with labels result in a representation in which there is no contradiction spelled out by the constraints.

There are two pluggings, as is easy to verify:

	h_0	h_1	h_2
P_1	l_1	l_3	l_2
P_2	l_2	l_1	l_3

Plugging P_1 interprets (22) as giving the universal quantifier wide scope, out-scoping the existential quantifier. The corresponding formula in predicate logic is (23), which is true in a model where all boxers love a woman, but not necessarily the same woman.

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

Plugging P_2 interprets (22) as the existential quantifier out-scoping the universal quantifier. In a model where there is some woman that is loved by all boxers, this interpretation denotes truth. A corresponding formula in predicate logic is (24).

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

The Plugging Algorithm

Before turning to the plugging algorithm we have to consider the design of Prolog representations for our PLU-formulas. Labels and holes are represented as Prolog variables (as with first-order variables, this gives us the practical advantage that any introduction of a label or hole appears as a new occurrence). With this we are able to represent PLU formulas. Further, we have `leq(L,H)` as the Prolog notation for the constraint `L` $\leq$ `H`. And finally, labeled formulas are formed by putting together a label with a formula using the Prolog inbuilt : operator.

A USR itself is represented as `usr(LFs,Cons)`, where `LFs` is a list of the labeled formulas, and `Cons` is a list of constraints. Here is a sample representation:

```
usr([l1:exists(X,h1&h2),l2:boxer(X),l3:collapse(Y)],
    [leq(l1,h0),leq(l2,h1),leq(l3,h2),leq(l3,h0)])
```

This Prolog term is our representation in PLU for 'A boxer collapses'. We won't provide predicates to check for properness of USRs — instead we supply a plugging algorithm straightaway and assume that the USRs we work with are connected, have a unique top, and are acyclic.

The heart of the plugging algorithm is centered around `plugHole/4`. Its first argument is the hole to be plugged (normally we start with the top hole of the USR, and the work recursively through the representation until we've plugged all holes). The second argument are the labeled conditions (coded as a different list - using to keep track of which labeled formulas are left for plugging and which are not), and the third argument is the list of constraints imposed by the USR. Its last argument `Scope` is a list of formulas that, under the current reading, outscope `Formula`.

```
plugHole(Formula,LFs1-LFs3,Constraints,Scoped):-
   deleteFromList(Label:Formula,LFs1,LFs2),
   checkConstraints(Label,Constraints,[Formula|Scoped]),
   variablesInTerm(Formula,[]-Arguments),
   checkArguments(Arguments,LFs2-LFs3,Constraints,[Formula|Scoped]).
```

With `deleteFromList/3` (see `basicsLib.pl`) we pick a labeled formula to be plugged and remove it from the list. This predicate is backtrackable — so if it turns out that we picked one that violates the constraints we can pick another instead later. We add `Formula` to the list of scoped formulas and check if no constraints are violated.

Then we continue plugging, by identifying the arguments of the newly plugged formula with `variablesInTerm/2`. This predicate, defined in the library file `basicsLib.pl`, takes a term and returns all its variables. The rest of the work is done with the help of `checkArguments/4`. It traverses through the list of arguments, and according to the type of argument (recall that these can be holes, labels, or ordinary first-order variables), continues the plugging process.

The first clause of `checkArguments/4` takes care of holes. We identify holes by imposing a member-check on the constraints. Next this holes is plugged using `plugHole/4` and the rest of the arguments are checked.

```
checkArguments([Arg|Rest],LFs1-LFs3,Constraints,Scoped):-
   memberOfList(leq(_,Hole),Constraints),
   Hole==Arg, !,
   plugHole(Hole,LFs1-LFs2,Constraints,Scoped),
   checkArguments(Rest,LFs2-LFs3,Constraints,Scoped).
```

Clause 2 cares about labels. In that case the corresponding labeled condition is removed from the list of labeled formulas and its arguments are checked as there might be holes among them.

```
checkArguments([Arg|Rest],LFs1-LFs4,Constraints,Scoped):-
   deleteFromList(Label:Formula,LFs1,LFs2),
   Label==Arg, !,
   Formula=Arg,
   variablesInTerm(Formula,[]-Arguments),
   checkArguments(Arguments,LFs2-LFs3,Constraints,Scoped),
   checkArguments(Rest,LFs3-LFs4,Constraints,Scoped).
```

The third clause carries on the recursion if the argument is neither a hole nor a label, hence a first-order variable, which we don't need to bother about (note that the two clauses above use the cut preventing to enter this third clause).

```
checkArguments([_|Rest],LFs1-LFs2,Constraints,Scoped):-
   checkArguments(Rest,LFs1-LFs2,Constraints,Scoped).
```

Finally, there is a clause that ends the recursion:

```
checkArguments([],LFs-LFs,_,_).
```

A typical Prolog call to start the algorithm is `?- plugHole(Top,LFs-[],Cons,[])`, where `Top` is the hole that out-scopes all other formulas, and `LFs-[]` ensures that all labeled-formulas are 'used' for building the fully specified formula, and moreover, prevents us from getting entangled in cyclic structures. If this goal succeeds, `Top` unifies with the resolved formula.

The constraints are checked each time after a hole is plugged. This is done with the help of `checkConstraints/3`. Its first argument is the label of the formula that is plugged, its second argument is the list of constraints, and its third argument the list of scoped formulas. In fact, it is just matter of working through the $\leq$-constraints and for each `leq(L,H)`, where the plugged label is identical with `L`, checking whether `H` is is part of the scoped formulas.

```
checkConstraints(_,[],_).
checkConstraints(Label,[leq(L,H)|Constraints],Scoped):-
   Label==L, !,
   memberOfList(Formula,Scoped), Formula==H,
   checkConstraints(Label,Constraints,Scoped).
checkConstraints(Label,[_|Constraints],Scoped):-
   checkConstraints(Label,Constraints,Scoped).
```

Note that we use the cut after identifying a relevant constraint. So if a constraint is violated, we force Prolog to backtrack and try a different plugging. And this is all there is to it.

Before putting the plugging algorithm into action, we will show how to build underspecified representations using our standard lexicon and grammar rules.

Building Underspecified Representations

Building USRs is done with the tools we're already familiar way: using lambdas and β-conversion. For USRs, we are going to introduce lambdas for holes and labels as well. Further, we need a *merge*-operation for USRs, that combines two USRs to one big USR.

The way we construct USRs is a basically an extension to our normal systematic way of constructing representations. To keep track of the holes and labels, we will decorate *each*

USR-representation with two additional lambdas: the 'top hole', and the 'main label'. The top hole is the hole that closes the current scope domain. The main label is the label within the current scope domain that is out-scoped by all elements within its scope domain. Abstracting away from these two concepts, it is straightforward to combine USRs (incidentally, interface definitions might get complicated — drawing a picture often eases matters).

Let's introduce the semantic macro for a determiner:

```
detSem(uni,lambda(P,lambda(Q,lambda(H,lambda(L,
       merge(merge(usr([F:forall(X,Restr>Scope)],
                       [leq(F,H),leq(L,Scope)]),
       P@X@H@Restr),Q@X@H@L))))))).
```

The two additional lambdas for `H` and `L` close the scope domain of 'every'. The quantifier itself gets label `F` with restriction `Restr` and nuclear scope `Scope`. Constraint `leq(F,H)` ensures that the quantifier is out-scoped by the top of the scope domain, and `leq(L,Scope)` says that its verbal argument is in its scope. The applications `P@X@H@Restr` and `Q@X@H@L` are crucial — these associate the restriction label of the quantifier with the label of the noun representation, and the top and hole label of the verb phrase argument.

Proper names or pronouns, on the other hand do not introduce holes, and therefore need not to bother about leq-constraints. A new top hole `H` and main label `L` are introduced and applied to its argument `P`.

```
pnSem(Constant,lambda(P,lambda(H,lambda(L,P@Constant@H@L)))).
```

Verbs introduce the 'top' hole and a leq-constraint between itself and the top. This is of importance for sentences where no quantifiers (or other scope-introducing elements) appear, as it establishes a clean link from the top element to the main label, that of the verb.

```
ivSem(X,F,lambda(X,lambda(H,lambda(L,usr([L:F],[leq(L,H)])))))).
```

Underspecification allows us to be very flexible. We can use one and the same technique for both quantifiers and negation. Here is the interface definition for verb negation:

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

There is one thing left we haven't paid any attention to yet. It is combining smaller USRs to bigger ones, by carrying out the stipulated `merge/2` operations in the USR. Actually, this is not a big deal, and done as follows, using `appendLists/2` from the library file.

```
mergeUSR(usr(L,C),usr(L,C)).
mergeUSR(merge(U1,U2),usr(L3,C3)):-
   mergeUSR(U1,usr(L1,C1)),
   mergeUSR(U2,usr(L2,C2)),
   appendLists(L1,L2,L3),
   appendLists(C1,C2,C3).
```

As usual we implement a driver that grasps all the important predicates together. This one combines the plugging algorithm and the construction of USRs, displays the USR on the current output device as well as all the readings it has.

```
parse:-
   readLine(Sentence),
   s(Sem,Sentence,[]),
   betaConvert(Sem@Top@_,Reduced),
   mergeUSR(Reduced,usr(LFs,Cons)),
   printRepresentation(usr(LFs,Cons)),
   findall(Top,plugHole(Top,LFs-[],Cons,[]),Readings),
   printReadings(Readings).
```

Here is an example session:

```
?- parse.

> every woman does not snorts.

usr([A:forall(B,C>D),C:woman(B),E: ~F,G:snort(B)],
    [leq(A,H),leq(G,D),leq(E,H),leq(G,F),leq(G,H)])

Readings:

1 forall(X,woman(X)> ~snort(X))
2 ~ forall(Y,woman(Y)>snort(Y))
```

Evaluating Underspecification

So far we have formalized semantic underspecified representations and are able to deal with (quantifier) scope ambiguities. But how does this proposal relate to earlier accounts of scope ambiguities? Are there any benefits in using the new streamlined underspecified representation rather than stick to the old fashioned storage methods? The basic answer is

that underspecification languages are more powerful: they exhibit far more descriptive potential than storage methods. To make this statement more clear, let's consider a concrete example sentence including three quantifiers:

(25) One criminal knows every owner of a hash bar.

With the tools developed in this chapter it is perfectly possible to represent the relative scopes of the three quantifiers fully underspecified. Moreover, it is possible to partially specify relative scopes. For some (linguistic) reason, one might wish to express that 'one criminal' out-scopes 'every owner' and 'a hash bar', but that the relative scopes of the latter two quantifying constructs does not matter. We need not add new machinery to our formalism: we can use the $\leq$-constraint to state such requirements.

How do other approaches to quantifier scope fare with such requirements? Storage techniques (Cooper 1983; Keller 1988), to begin with, are not able to express such constraints. A store is a sequence of the core semantic representation and a list of the binding operators (the quantifiers) which scope is not fixed yet. In the process of constructing a store, quantified noun phrases can either enter the store or combine with the core semantic representation (and form a new core semantic representation, by means of in-situ quantification). At the end of the parse, the different ways in which the quantifiers are retrieved from the store and combined with the core semantic representation result in different readings (the last quantifier which is pulled off the store gets widest scope). Hence, to give a quantifier obligatory wide scope, the only way to pursue is to put it on the store, while performing in-situ application for the remaining quantifiers. But this presumes that the scope order of the remaining quantifiers is fixed!

Software Summary of Chapter 3

`quantMontague.pl` A simple implementation of Montague's approach to quantification. (page 314)

`storageInterface.pl` Contains the specification for functional application for storage approach to quantification, as well as the interface definition for the pre-terminal categories. (page 317)

`cooperStorage.pl` Implementation of Cooper Storage. Defines storage of noun phrases and retrieval of the store. (page 319)

`kellerStorage.pl` Implementation of Nested Storage (Keller). Defines storage of noun phrases and retrieval of the store. (page 320)

`pluggingAlgorithm.pl` The plugging algorithm for hole semantics. (page 322)

`plugUSR.pl` Contains predicates for underspecified semantic representations. (page 323)

Further Reading

More about quantifier scoping in general, and Montague's approach to quantification in particular, can be found in Dowty, Wall, and Peters 1981 or volume 2 of Gamut 1991b. Cooper storage was first used in Cooper 1975; a more refined version is presented in Cooper 1983. Nested Cooper storage is due to Keller (1988). This very readable paper introduces Cooper storage, explains where the problems lie, gives an example (involving an interaction between scope and anaphoric pronouns) which suggests that a simple check for free variables during retrieval isn't an adequate solution, and then introduces nested Cooper storage.

There are other approaches to quantifier scope ambiguity the reader should be aware of. Hobbs and Shieber 1987 provide an alternative to Cooper storage that overcomes some of the problems mentioned in this chapter. An improved version of their algorithm has been developed by Lewin (1990).

As we mentioned in the text, stores (and nested stores) are essentially a more abstract form of semantic representation — representations which encode multiple possibilities, without committing us to any particular choice. Viewed in this light, stores are ancestors of the current crop of *underspecification formalisms*, representation languages specifically

designed to cope with ambiguity by avoiding overcommitment.

One of the first underspecified semantic representations were underspecified discourse representation structures (UDRSs) proposed by Reyle (1993). In fact, hole semantics can be seen as a generalization of this idea, not stuck to DRSs but compatible to all kinds of logical structures. Hole semantics is also applied in the machine translation system Verbmobil (Bos et al. 1996; Bos et al. 1998).

Probably the best way for the reader to find out more about the area of underspecification is to consult the recent collection edited by Van Deemter and Peters (Van Deemter and Peters 1996).

Part II

Working with Discourse Representations

Chapter 7

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, presuppositions, and VP-ellipsis. 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.

7.1 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 1,

in certain respects free variables are like pronouns, thus a first attempt at representation might be LOVE(x, 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:

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

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:

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

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

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

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 2 (where we developed the lambda calculus after realizing that naive computational approaches to noun phrase representations led to trouble) and Chapter 3

(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 and VP-ellipsis 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

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

nor the final resolved representation

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

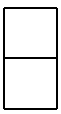
corresponds well with our intuitions about the way the second discourse actually *works*. Intuitively, the sentence '**A woman snorts**' sets the agenda. It paints a simple picture: we are talking about a woman, a woman who snorts, and presumably the subsequent discourse will tell us more about her (perhaps 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 $\exists \text{z}(\text{WOMAN}(\text{z}) \wedge \text{SNORT}(\text{z}))$. 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!

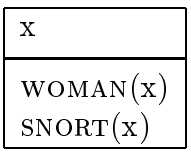
Summing up, we would like a semantic architecture that enables us to build representations that naturally mirror CCP, but which also allows us easy access to the truth-conditional dimension of meaning. And in fact, this is precisely what Discourse Representation Theory offers. For our first encounter 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:



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, VP 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 1.1). 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

$\mathrm{x}_1, \ldots, \mathrm{x}_n$
γ_1
.
.
.
γ_m

   is a DRS.

2. If R is a relation symbol of arity n, and τ_1,...,τ_n are terms, then $\mathrm{R}(\tau_1,...,\tau_n)$ is a condition.

3. If τ_1 and τ_2 are terms then $\tau_1 = \tau_2$ is a condition.

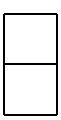
4. If B is a DRS, then $\neg B$ is a condition.

5. If B_1 and B_2 are DRSs, then $B_1 \vee B_2$ is a condition.

6. If B_1 and B_2 are DRSs, then $B_1 \rightarrow B_2$ 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



(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 DRSs 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

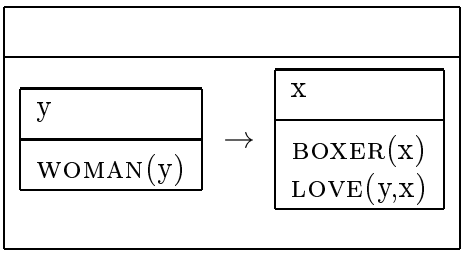
x y
WOMAN(x) BOXER(y) ADMIRE(x,y)

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 7.1.1 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.*

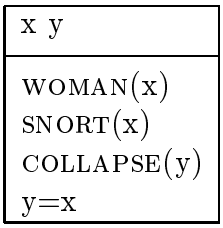
This can be simplified. When using DRSs to analyze natural language, we usually end up working with DRSs in which each discourse referent occurs in exactly one universe. When this is the case we don't need to bother talking about occurrences of discourse referents in universes — in effect we can identify 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.

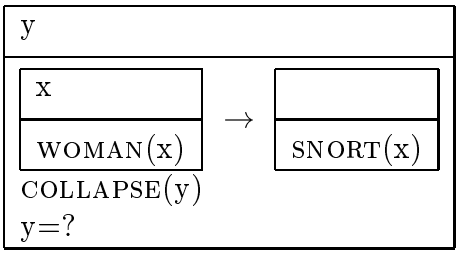
Let's see some examples of the accessibility constraint at work. First, it enables DRT to correctly predict that an anaphoric link is permitted between 'she' and 'a woman' in the first little discourse that follows, but not in the second:

(2) A woman snorts. She collapses.



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.



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.

Exercise 7.1.2 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 2). 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:

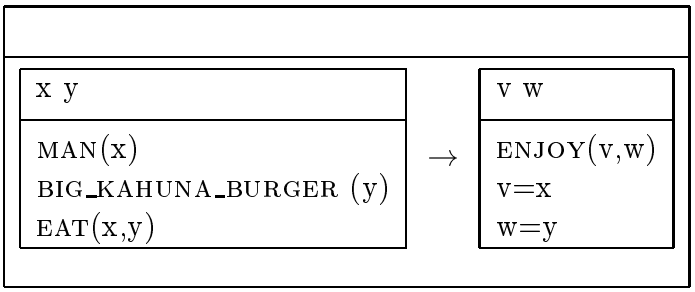
$$\forall x \forall y[\text{MAN}(x) \wedge \text{BIG_KAHUNA_BURGER}(y) \wedge \text{EAT}(x,y) \rightarrow \text{ENJOY}(x,y)].$$

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 2, it's quite likely that you'll wind up generating:

$$\exists x[\text{MAN}(x) \wedge \exists y[\text{BIG_KAHUNA_BURGER}(y) \wedge \text{EAT}(x,y)] \rightarrow \text{ENJOY}(x,y)].$$

Exercise 7.1.3 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:



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:

$$\forall x \forall y[\textsc{man}(x) \wedge \textsc{big_k_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \exists v \exists w(\textsc{enjoy}(x,y) \wedge v{=}x \wedge w{=}y)],$$

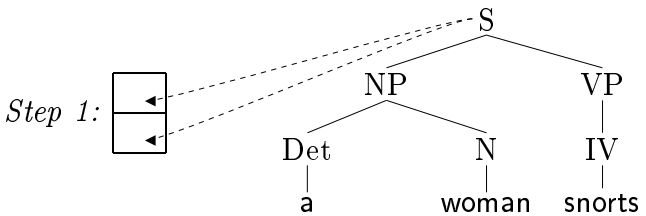
which we is easily seen to be equivalent to the earlier first-order representation.

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.

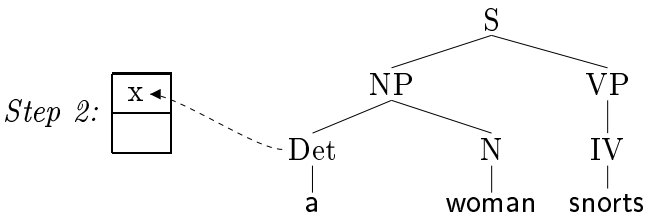
The Standard Construction Algorithm

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.

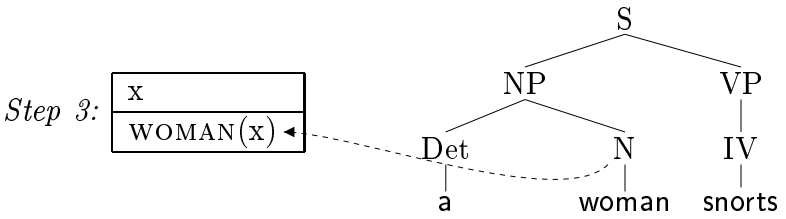
First we start feeding information from the initial sentence into a box:



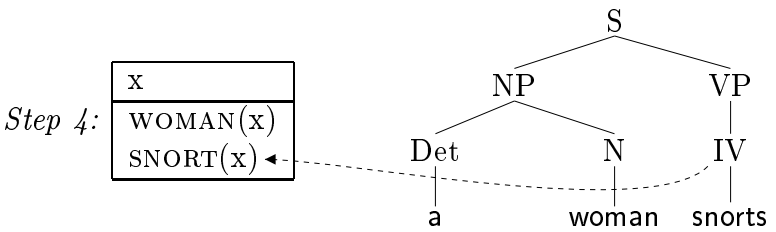
We proceed is follows: we are going to work our way around the tree in a top-down, left-to-right fashion, adding discourse referents as we go. So first we work down from the S node to the NP node. We then peek inside the NP: what sort of NP is it? In fact, it's an indefinite noun phrase; indefinites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



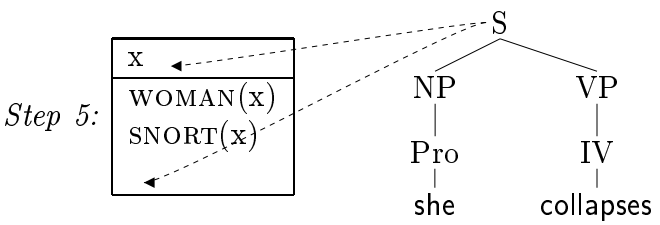
What conditions attach to this discourse referent? To find out, we move around the NP subtree to the N node. We find the word 'woman', thus we constrain x by putting the condition WOMAN(x) in the lower part of the box (Step 3).



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:

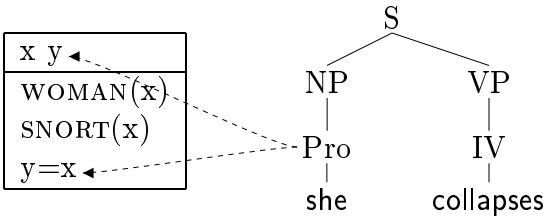


This is the DRS for the first sentence of discourse (2). Let's continue straight on with the second sentence. As with the first sentence, we will explore its parse tree in a top-down, left-to-right fashion. Crucially, the standard construction algorithm insists that we add all the information we obtain 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.



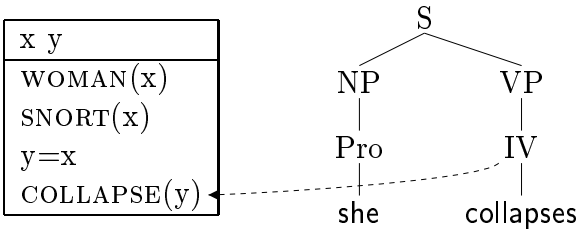
Again we work our way down from S, to NP, and then peek to see what kind of NP we are dealing with. It's a pronoun, so this means we must introduces a new discourse referent (we have chosen y) and a condition of the form y=?, where the question mark will be identified with an accessible, earlier introduced, 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.

7.2 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 DRSs 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 $\mathbf{M}$:

$\mathbf{M}, i \models$

$\mathrm{x}_1, \ldots, \mathrm{x}_n$
K_1
.
.
.
K_m

iff there is an embedding j such that j extends i, and $j(\mathrm{x}_1), \ldots, j(\mathrm{x}_n)$ are all defined, and j satisfies conditions $\mathrm{K}_1, \ldots, \mathrm{K}_m$ in $\mathbf{M}$.

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 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:

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

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 7.2.1 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 7.3.5.)

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 $\mathbf{M} = (D, R)$ 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 moments 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 x_1, $\ldots$, 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 1.1.) 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:

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

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

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

$\mathbf{M}, g \models B_1 \vee B_2$ *iff* there is an h such that $\mathbf{M}, g, h \models B_1$ or $\mathbf{M}, g, h \models B_2$

$\mathbf{M}, g \models B_1 \rightarrow B_2$ *iff* for all f such that $\mathbf{M}, g, f \models B_1$ there is an h such that $\mathbf{M}, f, h \models B_1$

$\mathbf{M}, g, h \models$ | $\mathrm{x}_1, \ldots, \mathrm{x}_1$ | / K_n / . / . / . / K_m *iff* there is an h such that $g[\mathrm{x}_1, \ldots, \mathrm{x}_1]h$ and $\mathbf{M}, h \models \mathrm{K}_1 \,\&\, \cdots \,\&\, \mathbf{M}, h \models \mathrm{K}_m$

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 we 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 $(f, f') \in [\![\mathrm{B}]\!]$.

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 7.2.2 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 $\mathbf{M}$.

7.3 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:

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

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:

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

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

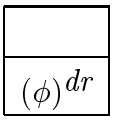
Exercise 7.3.2 We claimed that *fo* was a satisfaction preserving translation. Show that this claim can be made precise in two ways. Let B be a DRS. First, show that B is *dynamically* satisfied in a model $\mathbf{M}$ if and only if $\mathbf{M} \models (\mathrm{B})^{fo}$. Second, show that B is *embedding* 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.

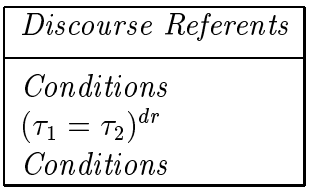
Given a preprocessed first-order formula ϕ, how do we translate it? First we form the following structure:



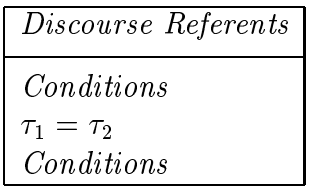
That is, we take an empty box, and put the expression $(\phi)^{dr}$ in the compartment reserved for conditions. The superscripted *dr* is our translation function, and we shall simply keep recursively applying this function; as we do so we will fill the box with conditions and discourse referents until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



(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:



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:

Discourse Referents
Conditions $(\phi \wedge \psi)^{dr}$ *Conditions*

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

Discourse Referents
Conditions $(\phi)^{dr}$ $(\psi)^{dr}$ *Conditions*

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

Discourse Referents
Conditions $(\neg\phi)^{dr}$ *Conditions*

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

Discourse Referents
Conditions $\neg$ [ ] [$(\phi)^{dr}$] *Conditions*

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

Discourse Referents
Conditions $(\exists x\phi)^{dr}$ *Conditions*

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

x, *Discourse Referents*
Conditions $(\phi)^{dr}$ *Conditions*

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 Exercise7.4.4).

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 7.3.3 Preprocess $\forall x(\text{WOMAN}(x) \rightarrow \text{SNORT}(x))$ and use *dr* to translate it into a DRS.

Exercise 7.3.4 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 $\forall x(\text{WOMAN}(x) \rightarrow \text{SNORT}(x))$.

Exercise 7.3.5 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?]

7.4 DRT in Prolog

The architecture that is 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.

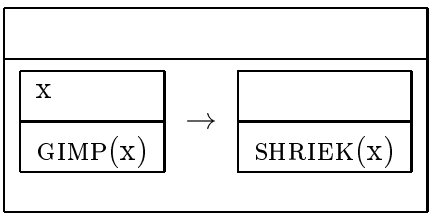
We shall do two things. First, we define a simple model checker for DRS languages. This will enable us to fix our basic Prolog conventions for DRT, and gives us an interesting new perspective on the semantics of DRSs. Following this, we implement the translation of DRT into first-order logic given in Section 7.3. 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 7.4.1 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) `sem/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 `semDrs/2` that succeeds if a DRS can be satisfied in a model, and a predicate `semCondition/2` that succeeds if a DRS-condition can be satisfied in a model. These predicates will be mutually recursive. Here is the code for `semDrs/2`:

```
semDrs(drs(Dom,C),Model):-
   discourseReferents(Dom),
   semConditions(C,Model).

discourseReferents([]).
discourseReferents([DiscourseReferent|Others]):-
   constant(DiscourseReferent),
   discourseReferents(Others).
```

Using the predicate `discourseReferents/1`, `semDrs/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 `semConditions/2`.

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

```
semConditions([],_).
semConditions([Condition|Others],Model):-
   semCondition(Condition,Model),
   semConditions(Others,Model).
```

So it remains to specify what `semConditions/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:

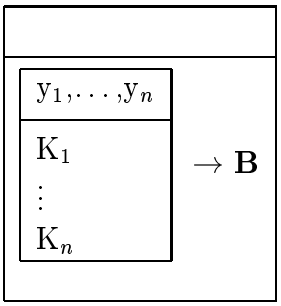
```
semCondition(~ Drs,Model):- \+ semDrs(Drs,Model).

semCondition(Drs1 v Drs2,Model):-
   (
    semDrs(Drs1,Model)
   ;
    semDrs(Drs2,Model)
   ).
```

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

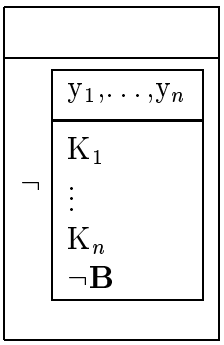
```
semCondition(Drs1 > Drs2,Model):-
   (
    semDrs(Drs1,Model),
    \+ semDrs(Drs2,Model),
    !,
    fail
   ;
    true
   ).
```

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 7.2.1 and 7.3.5, $\rightarrow$ conditions can be eliminated without loss of expressive power. In fact, a configuration of the form



is equivalent to a configuration of the form



A moments 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.

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

semCondition(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 1.2; we'll define a driver that, given a DRS, picks an example model and tries to satisfy it:

```
evaluateDrs(Drs,Example):-
   example(Example,Model),
   semDrs(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 7.4.2 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 `semCondition/2`.

Exercise 7.4.3 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 7.3.

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

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

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:

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

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 12.

Exercise 7.4.4 Implement the translation *dr* discussed in Section 7.3 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 7.3.4).

Further Reading

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 1991b). 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 the 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 8

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 2. So we (temporarily) 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. Finally we show that the threading and lambda based approaches can be straightforwardly combined, though we are forced to use the unification based approach to β-conversion.

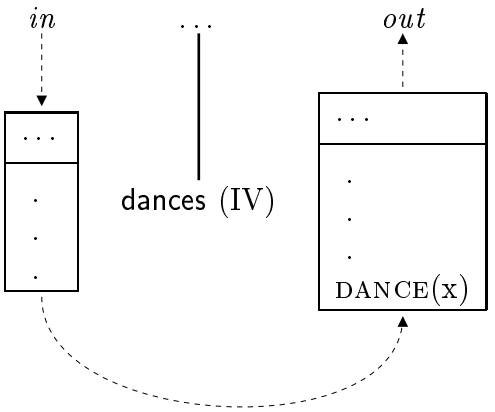
We then change gears. Basic semantic construction is all very well, but how are we to handle quantifier scope ambiguities? As we shall show, the hole semantics approach to underspecification adapts easily DRS languages, and implementing it requires only routine adaptations of our earlier code.

8.1 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 Chapter 7.1, 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:

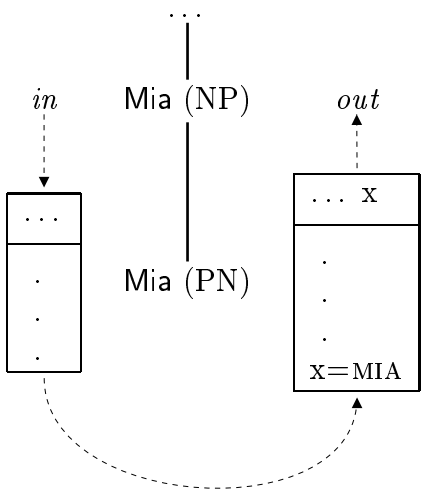
```
iv(X,DrsIn-DrsOut)-->
   {
    DrsIn=drs(Dom,Cond),
    DrsOut=drs(Dom,[dance(X)|Cond])
```

```
    },
    [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 2, 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,Cond)-drs(Dom,[dance(X)|Cond]))-->
    [dances].
```

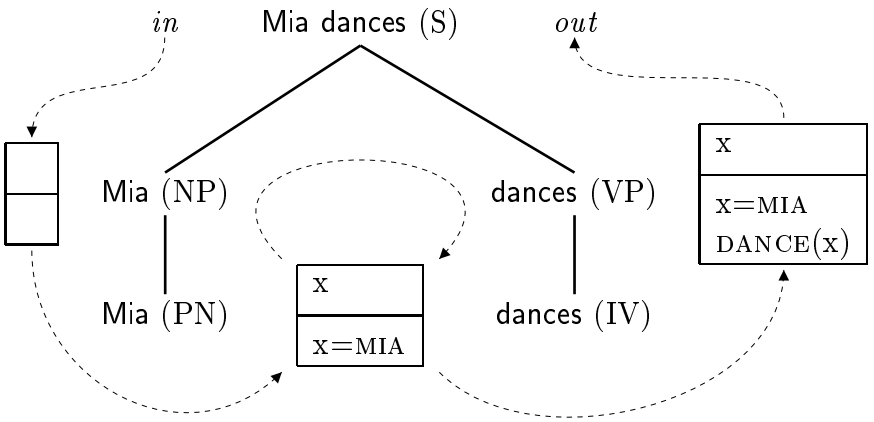
Lets now give a threading analysis of the sentence 'Mia dances'. First, we need to add a lexical entry for the proper name 'Mia'. Proper names introduce both a discourse referent and a condition. The following diagram shows what happens when a node labeled by 'Mia' is threaded through a DRS:



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,Cond)-drs([X|Dom],[X=mia|Cond]))-->
    [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 → PN and VP → 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 → NP VP, the following serves our needs:

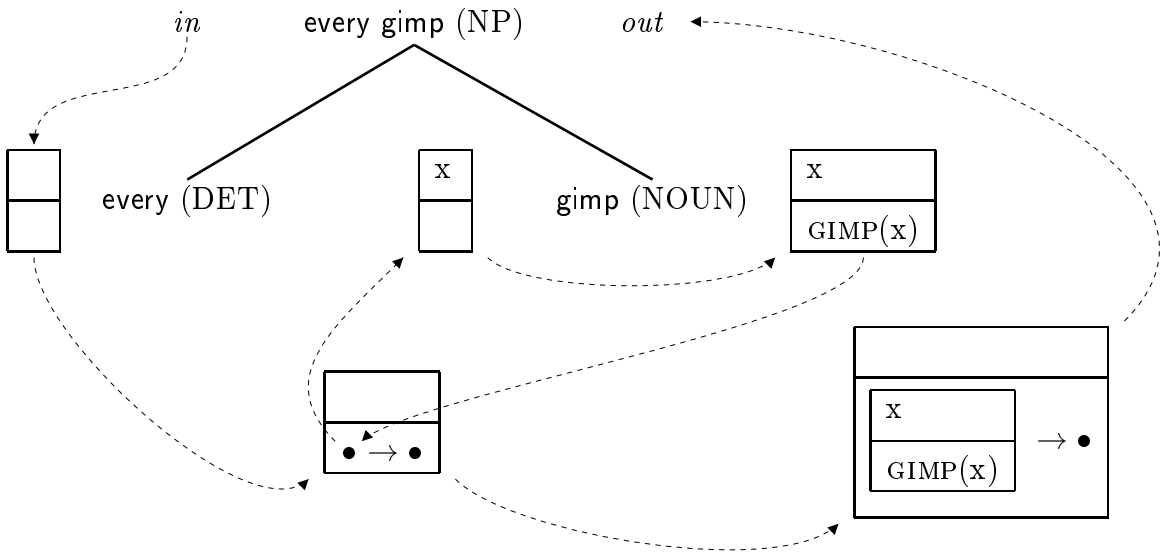
```
s(DrsIn-DrsOut)-->
   np(X,DrsIn-Drs),
   vp(X,Drs-DrsOut).
```

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 2.

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,Cond),
 DrsOut = drs(Dom,[RestrOut > ScopeOut|Cond]),
 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:

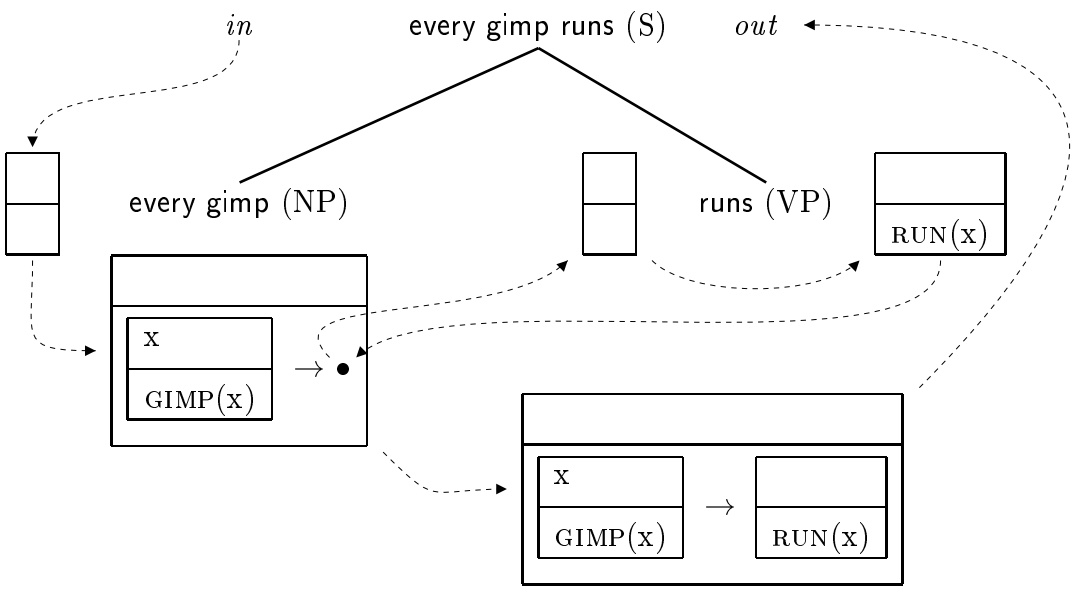
```
np(X,Drs,Scope)-->
   det(X,Drs,Restr,Scope),
   noun(X,Restr).
```

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 8.1.1 [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 8.1.2 [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 2, 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 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 8.1.3 [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 in Experiment 2 in Chapter 2. Make 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 2 with the threading DCG, For example, the entry for a transitive verb is simply:

```
tv(X,Y,drs(Dom,Cond)-drs(Dom,[NewCond|Cond]))-->
   {
    lexicon(tv,Relation,Phrase),
    compose(NewCond,Relation,[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 pair and an output pair — 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 work of the work of 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.

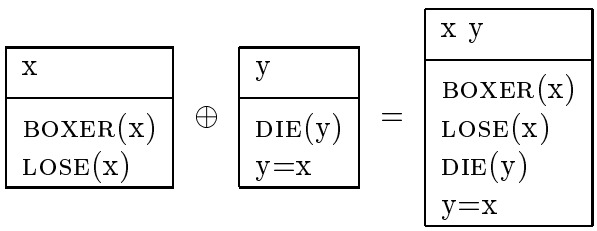
Exercise 8.1.4 [Intermediate] Change the program in such a way that proper names are floated to the top DRS. Hint: use a second DRS pair.

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, from 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?

8.2 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,



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': λy. $\boxed{\begin{array}{|l} \\ \hline \text{BOXER(y)} \end{array}}$

'woman': λy. $\boxed{\begin{array}{|l} \\ \hline \text{WOMAN(y)} \end{array}}$

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': λP.λQ. $\boxed{\begin{array}{|l} \text{z} \\ \hline \\ \end{array}} \oplus \text{P(z)} \oplus \text{Q(z)}$

'every': λP.λQ. $\boxed{\begin{array}{l} \\ \hline \boxed{\begin{array}{|l} \text{z} \\ \hline \\ \end{array}} \oplus \text{P(z)} \rightarrow \text{Q(z)} \end{array}}$

These are clearly analogous to the lexical entries given for these determiner given in Chapter 2. For example, we there assigned the following representation to 'every':

'every': λP.λQ.$\forall$z(P(z)$\rightarrow$Q(z))

In both representations, the abstracted variable P marks the missing restriction information, while Q marks the missing nuclear scope. Moreover, the quantificational force of the λ-DRS representation is obtained by using a combination of $\boxed{\begin{array}{|l} \text{z} \\ \hline \\ \end{array}}$ 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.

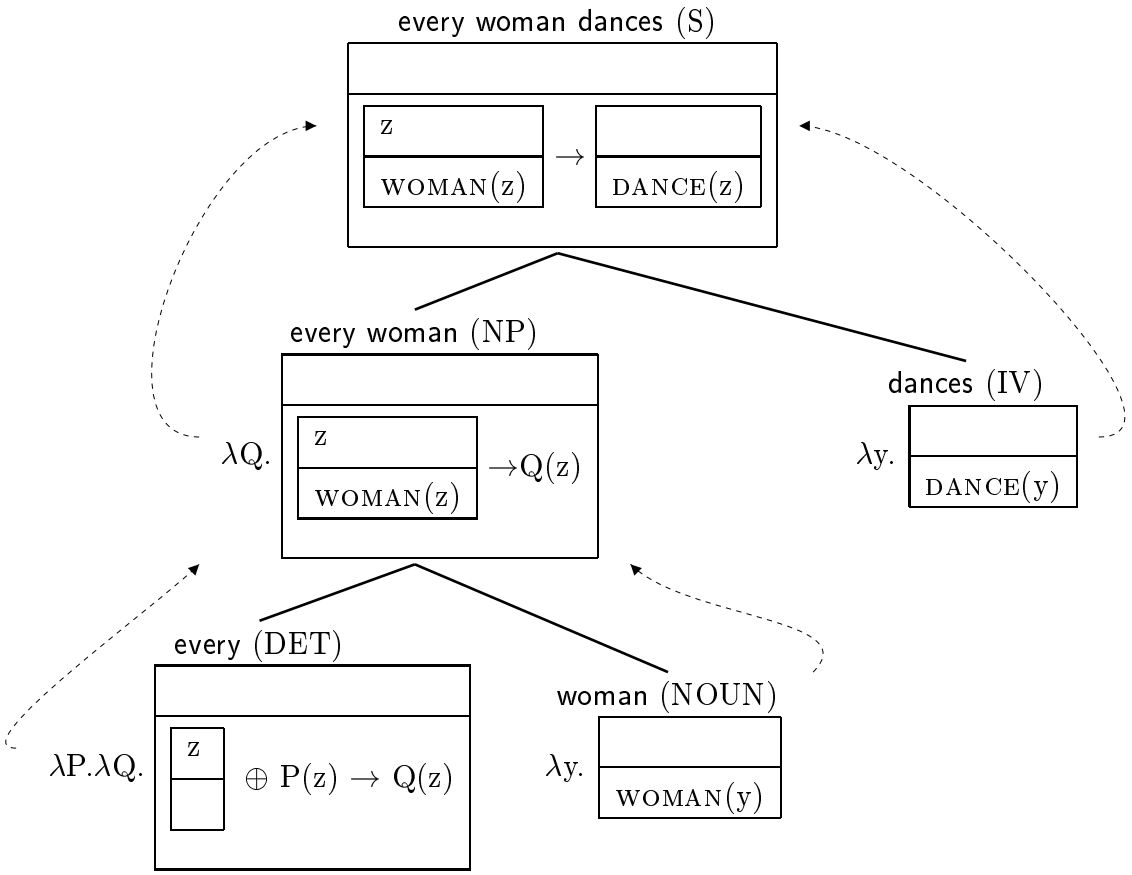
Finally, let's consider the lexical entries for intransitive and transitive verbs:

'dances': λy. $\boxed{\begin{array}{|c|}\hline \\ \hline \text{DANCE(y)} \\ \hline \end{array}}$

'loves': λX.λy.X(λx. $\boxed{\begin{array}{|c|}\hline \\ \hline \text{LOVE(y,x)} \\ \hline \end{array}}$)

Again, the reader should compare these entries with their analogs in Chapter 2.

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 2: 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 8.2.1 [Easy] Write out the steps involved in obtaining the λ-DRS associated with 'every woman' in the above tree.

Exercise 8.2.2 [Easy] Build the DRS representing 'A woman snorts' using the lexical representations given above.

Exercise 8.2.3 [Intermediate] State the needed lexical representations, and then build the DRS representation of the donkey sentence 'If a boxer loses, he dies'.

Exercise 8.2.4 [Easy] Suppose that we tried to avoid using the merge by giving 'a' the following lexical entry instead of the one given above:

'a': $\lambda P.\lambda Q.$ $\boxed{\begin{array}{|c|}\hline x \\ \hline P(x) \\ Q(x) \\ \hline\end{array}}$

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 2, lambda-based construction methods make good sense from a grammar engineering perspective.

So what do we actually have to do? Three things: define the semantic operations `apply/3` and `conjoin/3` (recall that `apply` is used in the grammar rules to combine non-sentence constituents, and `conjoin` for combining sentences with discourses), define the macros for the lexical entries, and wrap everything up in a new driver. Let's start with `apply`; we shall work with the substitution-based approach to β-conversion:

```
apply(Functor,Argument,Functor @ Argument).
```

Let's postpone the definition of `conjoin` for a moment, and think about the lexical entries. It follows from our earlier discussion that they will look like this:

```
noun(lambda(X,drs([],[boxer(X)])))-->
   [boxer].

iv(lambda(X,drs([],[dance(X)])))-->
   [dances].

aux(lambda(P,lambda(X,drs([],[~(P @ X)]))))-->
   [does,not].
```

But of course we don't need to supply lexical entries on this piecemeal basis. We already know how to abstract away from specific lexical entries by using semantic macros:

```
nounSem(Relation,lambda(X,drs([],[BasicCondition]))):-
   compose(BasicCondition,Relation,[X]).

ivSem(Relation,lambda(X,drs([],[BasicCondition]))):-
   compose(BasicCondition,Relation,[X]).

auxSem(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 `appendLists/3` in the semantic macro:

```
detSem(indef,lambda(P,lambda(Q,drs(D3,C3)))):-
   apply(P,X,drs(D1,C1)),
   apply(Q,X,drs(D2,C2)),
   appendLists([X|D1],D2,D3),
   appendLists(C1,C2,C3).
```

This is certainly simple, and it works —- but it is inefficient and causes unwanted backtracking. Another possibility would be to make use of difference lists, and the reader is asked to explore this option in an exercise. Here we shall explore a more straightforward alternative. We shall decide on a Prolog notation for the merge of two DRSs, build Prolog representations which contain such explicit merge markings, and then define a predicate which carries out the (explicitly marked) merges required. That is, we are going to adopt the same strategy we used 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(B,[a,boxer,eats,a,big,kahuna,burger],[]).

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

That is, we 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)),
   appendLists(D1,D2,D3), appendLists(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.

Moreover, note that we can use the `merge` operator to merge the DRSs associated with sentences, and thus to build the DRSs associated with entire discourses.

```
conjoin(Drs1,Drs2,merge(Drs1,Drs2)).
```

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 the 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 8.2.5 [Intermediate] Extend program `lambdaDRT.pl` with a treatment of the determiner 'no'. (Try to handle sentences like 'No boxer retires'.)

Exercise 8.2.6 [Intermediate] Extend the program to handle adjectives. (Try to handle sentences like 'A rich boxer retires'.)

Exercise 8.2.7 [Intermediate] Extend the program with ditransitive verbs (for example, 'give').

Exercise 8.2.8 [Project] Rework the programs given in this section so that they are based on the use of difference lists.

8.3 Threading in λ-DRT

We now turn address a very natural question. Our original approach to threading was inadequate because it adopted a Prolog specific approach to variable handling. Is it possible

to incorporate, in a straightforward and perspicuous way λ-based construction into the threading approach? The the use of lambdas should enable us to avoid fiddling round with extra DCG arguments, while the use use of threading should enable us to implement the merge efficiently (and hopefully) perspicuously. we end up with. Let's try it and see.

As in our the DRS-threading work, we are now going to work with *pairs* of DRSs: the first is the ingoing DRS, and the second the outgoing one. Moreover, we will be allowed to use lambdas to mark the missing information in such pairs. Thus we get the following lexical macros for the nouns, proper names, and verbs:

```
nounSem(Relation,lambda(X,drs(Dom,Cond)-drs(Dom,[NewCondition|Cond]))):-
   compose(NewCondition,Relation,[X]).

pnSem(Constant,lambda(P,drs(Dom,Cond)-DrsOut)):-
   apply(P,X,drs([X|Dom],[X=Constant|Cond])-DrsOut).

ivSem(Relation,lambda(X,drs(Dom,Cond)-drs(Dom,[NewCond|Cond]))):-
   compose(NewCond,Relation,[X]).

tvSem(Relation,lambda(K,lambda(Y,Drs))):-
   compose(NewCond,Relation,[Y,X]),
   apply(K,lambda(X,drs(Dom,Cond)-drs(Dom,[NewCond|Cond])),Drs).
```

And the determiners:

```
detSem(uni,lambda(Restr,lambda(Scope,DrsIn-DrsOut))):-
   apply(Restr,X,RestrIn-RestrOut),
   apply(Scope,X,ScopeIn-ScopeOut),
   DrsIn = drs(Dom,Cond),
   DrsOut = drs(Dom,[RestrOut > ScopeOut|Cond]),
   RestrIn = drs([X],[]),
   ScopeIn = drs([],[]).

detSem(indef,lambda(Restr,lambda(Scope,DrsIn-DrsOut))):-
   apply(Restr,X,RestrIn-RestrOut),
   apply(Scope,X,ScopeIn-ScopeOut),
   DrsIn = drs(Dom,Cond),
   DrsOut = ScopeOut,
   RestrIn = drs([X|Dom],Cond),
   RestrOut = ScopeIn.
```

Once again, we *don't* need to do any work on the grammar rules, and we can make use of the lexicon and grammar rules in Chapter 2 straightforwardly. To put all this to work, we need merely define the basic semantic operations.

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

conjoin(DrsIn-Drs,Drs-DrsOut,DrsIn-DrsOut).
```

Note that we don't have a choice what kind of β-conversion we want to use: threading goes hand in hand with unification-based lambda elimination.

Finally we set up a driver that reads a discourse, feeds it to the DCG, and pretty prints the output DRS:

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

And that's that. With this approach we have an interesting alternative our first version of λ-DRT. It is more efficient as it uses difference lists to build the boxes. It's drawback is we are forced to use Prolog unification for β-conversion.

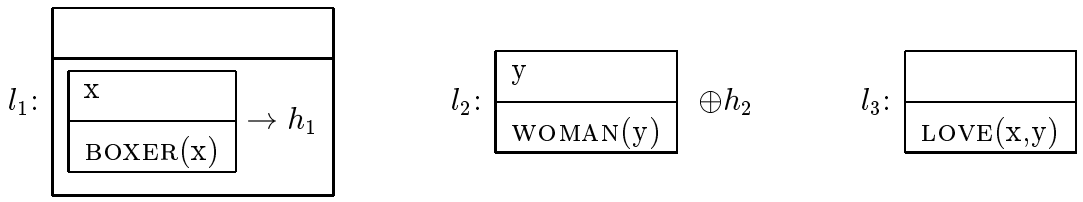
8.4 Underspecified DRSs

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?

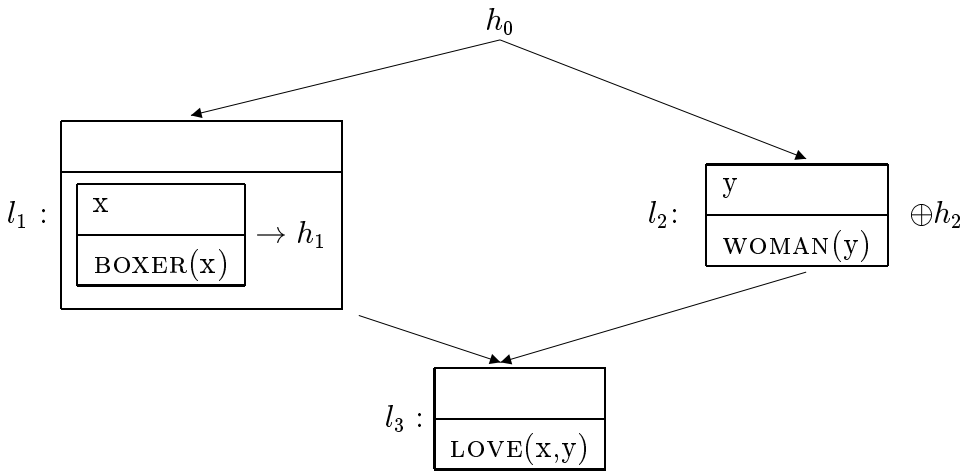
In fact both the storage and underspecification methods developed in Chapter 3 extend straightforwardly to λ-DRT. As far as storage methods are concerned, this is probably clear: from a computational perspective storage methods, essentially required us to work with a rather simple list-based extension 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 8.4.2. 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:



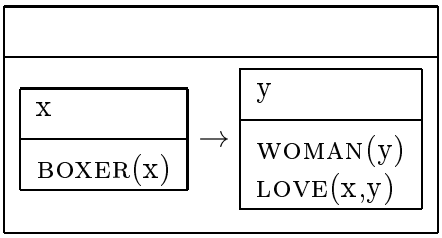
Adding the constraints $l_3 \leq h_1$, $l_3 \leq h_2$ ensures that the main verb is out-scoped by the DRSs introduced by the NPs, and $l_1 \leq h_0$, $l_2 \leq h_0$ enclose all the chunks within the scope domain h_0. We call the labeled DRSs together with the set of constraints an *underspecified discourse representation (UDR)*. Let's draw a picture of this UDR:



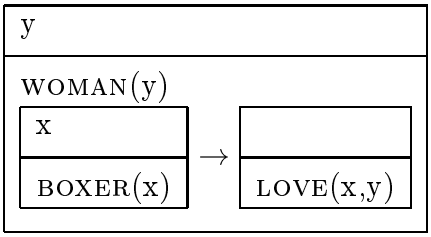
Resolving these ambiguous DRS-representations is not different from resolving the underspecified first-order representations used in Chapter 3. The holes underspecify scope, and in order to give us non-ambiguous interpretations, each hole should be *plugged* with a DRS in such a way that all the constraints are satisfied. In other words, a plugging for a UDR 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 UDR 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? A UDR is represented in Prolog as `udr(LDs,Cons)`, where `LDs` is a list of the labeled DRSs, and `Cons` is a list of constraints:

```
udr([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.

Here are a few examples of macros. They should be essentially familiar to the reader. Here, for example, are the definitions for nouns, proper names, and verbs.

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

pnSem(Constant,lambda(P,lambda(H,lambda(M,
      merge(udr([M:merge(drs([X],[X=Constant]),L)],[]),P@X@H@L))))).

ivSem(Rel,lambda(X,lambda(H,lambda(L,udr([L:drs([],[C])],[leq(L,H)]))))):-
   compose(C,Rel,[X]).
```

```
tvSem(Relation,lambda(K,lambda(Y,K@lambda(X,lambda(H,lambda(L,
                udr([L:drs([],[Formula])],[leq(L,H)])))))):-
   compose(Formula,Relation,[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 UDR, 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(udr([F:drs([],[merge(drs([X],[]),Restr)>Scope])],
                       [leq(F,H),leq(L,Scope)]),P@X@H@Restr),
                                              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.

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

What about the semantic operations? Since we want to use the substitution approach to β-conversion, and application is just an operation on ordinary lambda representations, the simple

```
apply(Functor,Argument,Functor @ Argument).
```

will do. For the conjoin operator, we have to do a bit more thinking. Let's introduce a merge with label `L` when combining two UDRs, and a new scope domain `H`. Then the following definition is accurate:

```
conjoin(Sem1,Sem2,lambda(H,lambda(L,
        merge(udr([L:merge(H1,H2)],[leq(L,H)]),
              merge(Sem1@H1@_,Sem2@H2@_))))).
```

One thing remains to be coded: merging UDRs. Using `appendLists` this boils down to the following, rather trivial, exercise:

```
mergeUDR(udr(L,C),udr(L,C)).
mergeUDR(merge(U1,U2),udr(L3,C3)):-
   mergeUDR(U1,udr(L1,C1)),
   mergeUDR(U2,udr(L2,C2)),
   appendLists(L1,L2,L3),
   appendLists(C1,C2,C3).
```

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

```
parse:-
   readLine(Sentence),
   s(Sem,Sentence,[]),
   betaConvert(Sem@Top@_,Reduced),
   mergeUDR(Reduced,udr(LDs,Cons)),
   printRepresentation(udr(LDs,Cons)),
   findall(Drs,(plugHole(Top,LDs-[],Cons,[]),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 an unreduced UDR. This is UDR is reduced, 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.

udr([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)])])
```

And that's it. The driver is part of the file `plugUDR.pl`, the complete interface definitions for UDRs are in the file `udrInterface.pl`.

Exercise 8.4.1 Running the program on coordinated NPs sometimes brings problems. Find out which cases do this, explain why, and propose a solution.

Exercise 8.4.2 Implement both Cooper storage and Keller storage for λ-DRT. Reuse as much code from Chapter 3 as possible.

Software Summary of Chapter 8

`threadingDRT.pl` Implementation of the Johnson & Klein's threading technique for constructing discourse representation structures. (page 339)

`lambdaDRT.pl` Building DRSs with lambdas. (page 342)

`lambdaThreadingDRT.pl` Combining lambdas and threading. (page 344)

`plugUDR.pl` Underspecified discourse representations. (page 346)

Further Reading

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 10

Presupposition Projection

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.

10.1 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 the discourse. 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.

How do presuppositions come to life? In most cases they are lexically triggered, or at least this is the case in 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 simple. There are three issues we have 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 "canceled" 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, I am not accepting your 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 is certainly robust, 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.

10.2 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*.

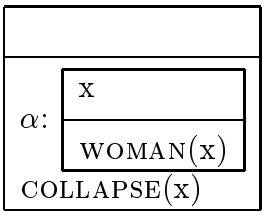
This is a strikingly simple and promising answer. DRSs are pictures of the evolving discourse; what better way to incorporate presuppositional information than by adding a new picture? The key word here is *incorporate*. Simply by placing the DRS yielded by a presupposition into a larger DRS we locate it in a web of discourse referents regulated by accessibility constraints. That is, by proceeding in this way we 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 this 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.

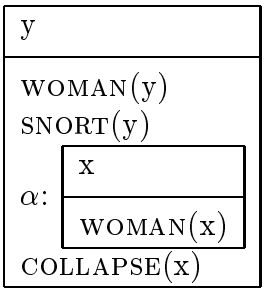
Task 1 appeals to our empirical knowledge of presupposition. For our fragment of English, we classify the definite article, possessive constructions, and proper names as presupposition triggers. For task 2 we extend the core DRS language with an α operator. The α operator takes a normal DRS as argument; it is simply a marker (or if you prefer, a *semantic feature*) that indicates that this DRS is 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 trigger, the DRS we output will generally contain α-DRSs. Here's the DRS we obtain for 'The woman collapses':



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?

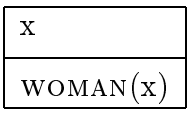
First we merge this new DRS with the DRS that represents the discourse so far; this merging process takes place while the presuppositions are still unresolved. For example, suppose that the previous utterance was 'A woman snorts'. Then after merging we obtain:



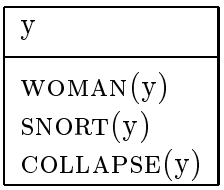
Only after merging do we attempt to resolve the presuppositions. We recursively travel through this new merged DRS and, for each α-marked DRS we encounter, we try to find a

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 elementary presupposition:



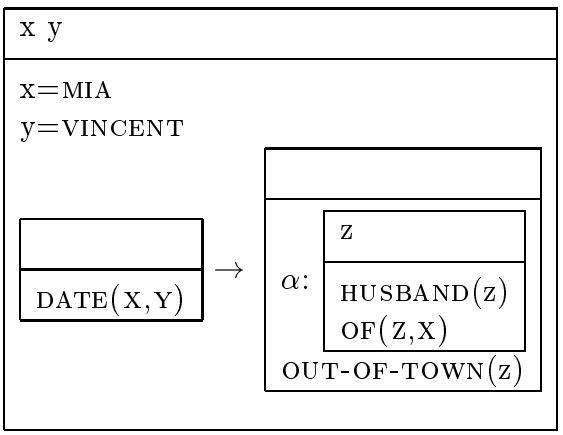
Note that if we identify the discourse referents x and y there is a partial match between the outermost DRS and the α-DRS. Carrying out this identification yields:



In short, we have successfully dealt with the uniqueness presupposition of 'the' by identifying the discourse referent x it introduced with the unique 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:



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:

z
HUSBAND(z) OF(Z,X)

In such cases our robust way of dealing with discourse comes into action: we accommodate the information to the outermost DRS and get the following, final, DRS:

x y z	
x=MIA y=VINCENT HUSBAND(z) OF(Z,X)	
	→
DATE(X,Y)	OUT-OF-TOWN(z)

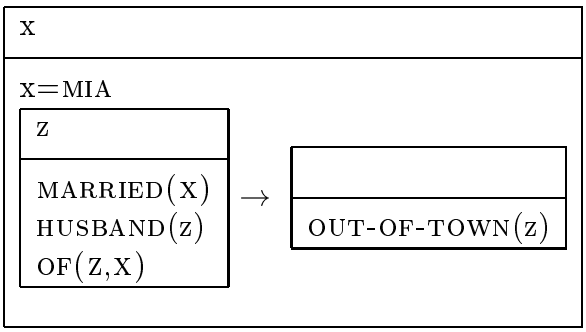
So far so good: our "try to match the presupposed information, and if that fails, accommodate it at the global DRS" strategy has survived its first test. But we were lucky with our example; consider instead 'If Mia is married, then her husband is out of town'. Does this presuppose that Mia has a husband?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

x y z	
x=MIA y=VINCENT HUSBAND(z) OF(Z,X)	
	→
MARRIED(X)	OUT-OF-TOWN(z)

The content of this DRS is "Mia has a husband. If she is married, then he is out of town", which is spectacularly wrong.

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 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.

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 12 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.

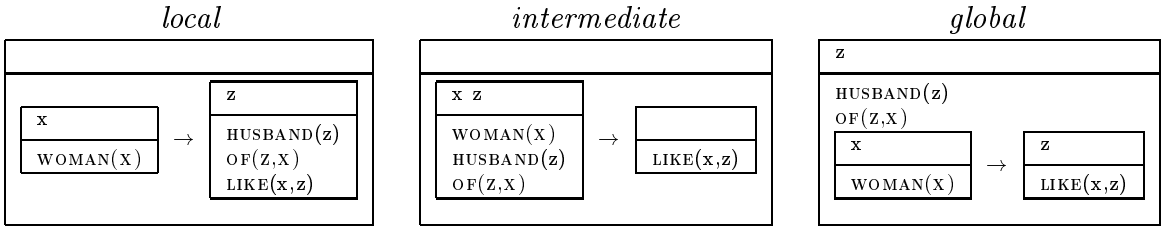
Summing up, both here and in Chapter 12, we are working with the following non-deterministic algorithm:

1. Generate a DRS for the input sentence with the tools of Chapter 8, 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:



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

Exercise 10.2.1 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 10.2.2 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 12.

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 12 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.

10.3 Presupposition Projection 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(B)`, where `B` is a DRS, to represent α-conditions. Here is the semantic macro for the determiner 'the':

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

Since linking is part of our presupposition projection 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(_,lambda(P,merge(drs([],[alfa(drs([X],[]))]),P@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(female),lambda(P,lambda(Q,
       merge(drs([],[alfa(merge(drs([X],[of(X,Y),
                                alfa(drs([Y],[]))]),P@X))]),Q@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 10.3.1 Fill in the the interface definition for the determiner 'another'. (It is already in the lexicon). What kind of presupposition does it trigger?

Implementing Linking and Accommodation

The initial output is a DRS with α-conditions. Let's call this DRS `PDrs` (a "presuppositional" DRS). This will be the input to the predicate `projectDrs/2`, that performs the linking and accommodation operations and finally returns a normal DRS, here we gave it the name `Drs`.

```
projectDrs(PDrs,Drs):-
   presupDrs(PDrs-ResolvedDrs,[]-Presups1),
   addPresuppositions(Presups1,ResolvedDrs-RDrs,[]-Presups2),
   linkOrAccommodate(Presups2),
   mergeDrs(RDrs,Drs),
   admissibleResolution(Drs).
```

The sub-predicates of `projectDrs/2` do the following work: `presupDrs` calculates the presuppositions of `PDrs` in a very special way: it records a list of potential linking and accommodation sites, and these are stored in `Presups1`. `ResolvedDrs` is the initial DRS but without the α-conditions. Before resolving these presuppositions, `addPresuppositions/3` adds potential global accommodation sites to `Presups1`, resulting in `Presups2`. Now we use `linkOrAccommodate/2` to link the presuppositions, or if that is impossible, accommodate them. Finally, after eliminating all `merge`s, we check whether the resolved DRS is admissible using `admissibleResolution/1`.

Now let's have a close look at these predicates.

First we have the predicates that gather the presuppositions with their potential linking and accommodation sites.

```
presupDrs(drs(Dom,Conds1)-drs(Dom,Conds2),P1-P2):-
   presupConditions(Conds1-Conds2,P1-P2).

presupDrs(merge(B1,B2)-merge(B3,B4),P1-P4):-
   presupDrs(B2-B4,[]-Presups1),
   linkPresuppositions(Presups1-P2,B1),
   presupDrs(B1-B3,P2-P3),
   appendLists(P1,P3,P4).


presupConditions([alfa(B1)|Conds1]-Conds2,P1-P3):-
   presupDrs(B1-B2,P1-P2),
   presupConditions(Conds1-Conds2,[adrs(B2,[],[])|P2]-P3).

presupConditions([B1 > B2|C1]-[B5 > B6|C2],P1-P7):-
   presupDrs(B2-B4,[]-P2),
   linkPresuppositions(P2-P3,B1),
   addPresuppositions(P3,B4-B6,[]-P4),
   presupDrs(B1-B3,P4-P5),
   addPresuppositions(P5,B3-B5,P1-P6),
   presupConditions(C1-C2,P6-P7).

presupConditions([B1 v B2|C1]-[B5 v B6|C2],P1-P4):-
   presupDrs(B1-B3,[]-Presups1),
   addPresuppositions(Presups1,B3-B5,P1-P2),
   presupDrs(B2-B4,[]-Presups2),
   addPresuppositions(Presups2,B4-B6,P2-P3),
   presupConditions(C1-C2,P3-P4).

presupConditions([~ B1|C1]-[~ B3|C2],P1-P3):-
```

```
   presupDrs(B1-B2,[]-Presups),
   addPresuppositions(Presups,B2-B3,P1-P2),
   presupConditions(C1-C2,P2-P3).

presupConditions([BasicCondition|C1]-[BasicCondition|C2],P1-P2):-
   compose(BasicCondition,_Symbol,Arguments),
   simpleTerms(Arguments),
   presupConditions(C1-C2,P1-P2).

presupConditions([]-[],P-P).
```

The presuppositions are gathered in a list of terms with the form `adrs(Drs,Links,Sites)`, where `Drs` is the presupposed DRS, `Links` is a list of equations (the potential links), and `Sites` a list of potential accommodation sites (a list of variables).

Creating this list is done with the help of two other predicates. One that adds a potential link, and one that adds a potential accommodation site.

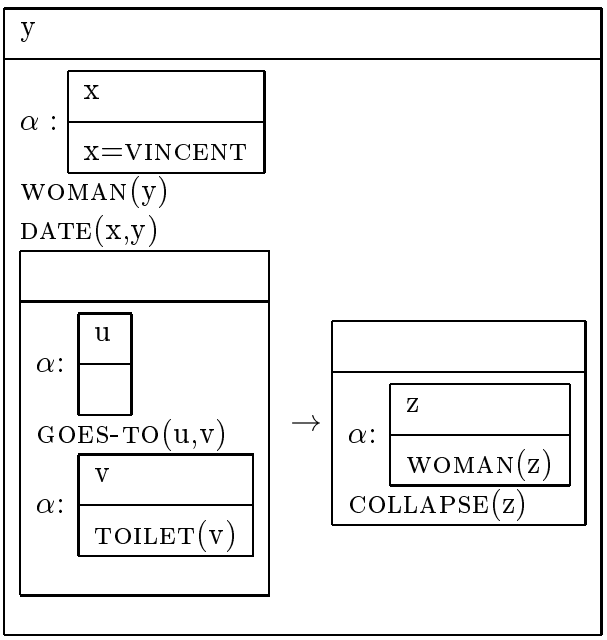
```
linkPresuppositions([]-[],_).
linkPresuppositions([Presup1|Others1]-[Presup2|Others2],ADrs):-
   linkPresupposition(Presup1-Presup2,ADrs),
   linkPresuppositions(Others1-Others2,ADrs).

linkPresupposition(Presup1-Presup2,ADrs):-
   Presup1=adrs(PresupDrs,Links,Projections),
   mergeDrs(PresupDrs,Drs),
   properAntecedentDrs(Drs,ADrs,Links-NewLinks),
   Presup2=adrs(PresupDrs,NewLinks,Projections).


properAntecedentDrs(drs(Dom,Conds),ADrs,Links-NewLinks):-
   accessible(ADrs,[]-ADom,[]-AConds),
   Dom=[AnaphoricReferent|_],
   match(AnaphoricReferent,ADom,Conds,AConds,Links-NewLinks).

match(_,[],_,_,Links-Links).
match(AnaRef,[AntRef|ADom],Conds,AConds,Links-NewLinks):-
   \+ \+ (AnaRef=AntRef,allExactMember(Conds,AConds)),!,
   match(AnaRef,ADom,Conds,AConds,[AnaRef=AntRef|Links]-NewLinks).
match(AnaRef,[_|ADom],Conds,AConds,Links-NewLinks):-
   match(AnaRef,ADom,Conds,AConds,Links-NewLinks).

accessible(B,D-D,C-C):-
```

```
   var(B),!.
accessible(merge(Drs1,Drs2),D1-D3,C1-C3):-
   accessible(Drs1,D1-D2,C1-C2),
   accessible(Drs2,D2-D3,C2-C3).
accessible(drs([D|Dom],Conds),D1-D2,C1-C2):-
   accessible(drs(Dom,Conds),[D|D1]-D2,C1-C2).
accessible(drs([],[alfa(Drs)|Conds]),D1-D3,C1-C3):- !,
   accessible(Drs,D1-D2,C1-C2),
   accessible(drs([],Conds),D2-D3,C2-C3).
accessible(drs([],[C|Conds]),D1-D2,C1-C2):-
   accessible(drs([],Conds),D1-D2,[C|C1]-C2).
accessible(drs([],[]),D-D,C-C).
```

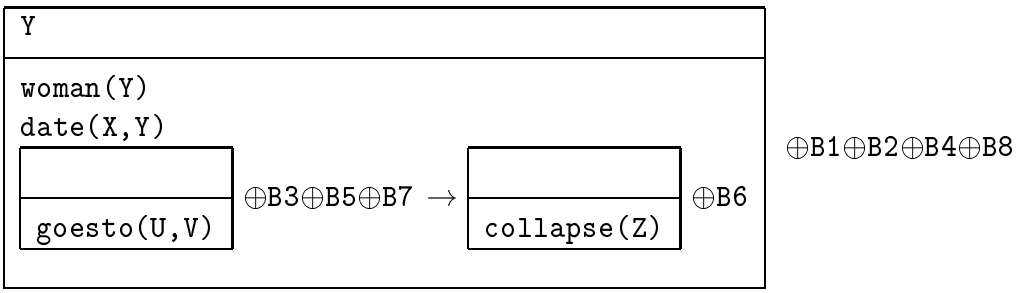
Adding potential accommodation sites is done with the help of the following predicate. The original DRS (`B1`) is replaced by `merge(B1,B2)`, where `B2` is the potential accommodation site. In other words, we leave a trace, which can be picked up later if we decide to use it (by simply unifying this variable with the to be accommodated DRS!).

```
addPresuppositions([],B-B,P-P).
addPresuppositions([adrs(D,L,Ps)|Others],B1-B3,P1-P2):-
   addPresuppositions(Others,merge(B1,B2)-B3,[adrs(D,L,[B2|Ps])|P1]-P2).
```

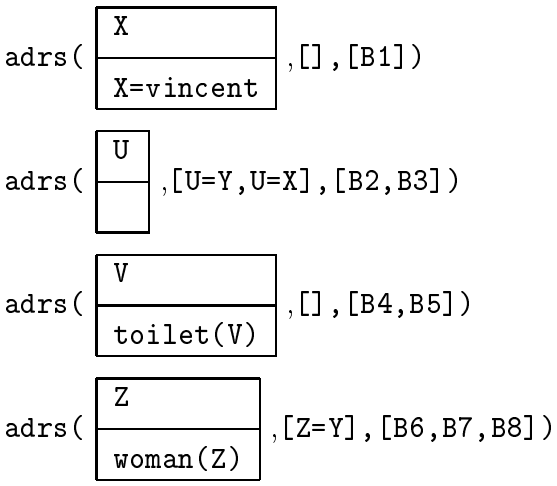
Let's see how this works for the simple discourse 'Vincent dates a woman. If he goes to the toilet, then the woman collapses'. We get



After applying the predicate `presupDrs` we get the following intermediate stage of the DRS (now in semi-Prolog notation):



And its list of presupposition with potential link and accommodation sites:



The last thing to do is take this DRS, and work through its presuppositions. And this is extremely easy: if a link can be made we just use Prolog unification to carry out that job (since we use Prolog variables for discourse referents), and unify all potential accommodation sites with the empty drs, with the help of `clearProjections/1`. For accommodation, things are equally pleasant. We pick a potential site (with `deleteFromList/3`), unify it with the presupposition, and then, again using `clearProjections/1`, clean all other projection sites.

```
linkOrAccommodate([]).
linkOrAccommodate([adrs(_,Links,Projections)|Others]):-
   memberOfList(X=Y,Links), X=Y, !,
   clearProjections(Projections),
   linkOrAccommodate(Others).
linkOrAccommodate([adrs(Presup,[],Projections)|Others]):-
```

```
        deleteFromList(Presup,Projections,OtherProjections),
        clearProjections(OtherProjections),
        linkOrAccommodate(Others).

    clearProjections([]).
    clearProjections([drs([],[])|Projections]):-
        clearProjections(Projections).
```

(Adding the cut the second clause prevents unwanted accommodation when linking is possible.) After doing `linkOrAccommodate/1`, there are still a number of merge operations to be carried out, which is done in the one but last clause of `projectDrs`.

Exercise 10.3.2 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).

Exercise 10.3.3 Add a free variable check on the generated solutions and test the modified programs on examples like 'every boxer likes his wife'.

Software Summary of Chapter 10

`presupDRT.pl` Implementation of Van der Sandt's presupposition projection algorithm for DRSs. (page 354)

`presupInterface.pl` Interface definitions for the presupposition program. (page 358)

Further Reading

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).

The only previous implementation of the Van der Sandt architecture we are aware of is due to Johan Bos (Bos 1993).

Chapter 12

Applications II

This interim version of Chapter 12 is devoted to implementing the acceptability constraints imposed by Van der Sandt on presupposition 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.

The material described here is the focus of ongoing development. In fact, the reader who is interested in the ideas explored here is urged not to stick with material presented here, but to keep an eye on

```
http://www/coli.uni-sb.de/~bos/comsem/
```

For a start, the reader will already find there the DORIS system, an improved implementation of these ideas. Later developments will be made available on this webpage.

12.1 Consistency and Informativity

Every contribution to a discourse should obey the maxims of conversation. Two very general maxims 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. It rules out the following discourses:

> 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 woman have husbands.

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, using a theorem prover to check which accommodation possibilities obey the maxims. We will also allow the user to ask what the DRS under consideration is, and the system will tell the user if one of the maxims is violated.

Which theorem prover are we are going to use? For a start, 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 12.1.1 [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, and think of examples that cannot be compiled out in this manner.

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.

Two predicates run the discourse manager: `ask` and `answer`. `ask/0` starts the session with an empty DRS. `ask/1` has as arguments a list of the current readings (represented as DRSs), and ask the user for new input. (If the number of current readings is zero, the system starts a new session.)

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

ask([]):- !,
   nl,write('Discourse not accepted. '),nl,
   nl,write('Starting again with empty discourse. '),nl,
   ask.

ask(Readings):-
   readLine(Input),
   answer(Input,Readings).
```

The predicate `answer/2` checks the input and gives an appropriate response. The input '`bye`' ends the session, '`new`' clears the discourse, and '`drs`' gives the current readings in DRS format. If the input is not one of these pre-defined commands the system tries to parse the input and create a DRS, in the way we are already familiar with.

```
answer([],Readings):- !, ask(Readings).
answer([bye],_):- !.
answer([new],_):- !, ask.
answer([drs],Readings):- !, printReadings(Readings), ask(Readings).
answer(Input,Readings):-
   d(MergeDrs,Input,[]),!,
   betaConvert(MergeDrs,ReducedDrs),
   getReadings(ReducedDrs,Readings,PotentialReadings),
   selectReadings(PotentialReadings,SelectedReadings),
   ask(SelectedReadings).
answer(_,Readings):-
   nl,write('Unknown input!'),nl,
   ask(Readings).
```

The predicate `getReadings/3` computes all potential reading for the yet unresolved DRS, using `projectDrs/2` (the presupposition projection predicate from Chapter 10). Then the predicate `selectReadings/2` selects those readings that do not violate the maxims of conversation. Here is how `getReadings` is coded.

```
getReadings(ReducedDrs,Readings,PotentialReadings):-
   findall((DrsSoFar,Drs),
           (
            memberOfList(DrsSoFar,Readings),
            projectDrs(merge(DrsSoFar,ReducedDrs),Drs)
           ),
           PotentialReadings).
```

Our reading selection module is coded as the predicate `selectReadings/2`. The first argument, the input, is a list of pairs of DRSs, where the first DRS is the DRS computed so far, and the second DRS is the translation of the new contribution to the discourse. The second argument, the output, only contains those readings that do not violate the maxims, coded as `consistent` and `informative`.

```
selectReadings([],[]).
selectReadings([(DrsSoFar,Drs)|Potential],[Drs|Selected]):-
   consistent(Drs),
   informative(DrsSoFar,Drs),
   !,
   selectReadings(Potential,Selected).
selectReadings([_|Potential],Selected):-
   selectReadings(Potential,Selected).
```

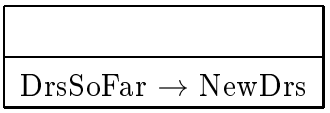
Checking for consistency boils down to translating the DRS to first-order logic — call this formula Φ — and then checking that Φ is consistent. This means we need to find out whether $\neg\Phi$ is a theorem: if it is, then Φ is inconsistent, and if it is not, then Φ is consistent. Now, Otter is a refutation 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.

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 `knowledge(Drs)`, an interface to certain facts of the world coded in DRS format (we give an example of such a knowledge base below) the following code takes such knowledge into account.

```
consistent(NewDrs):-
   knowledge(Drs),
   mergeDrs(merge(Drs,NewDrs),B),
   drs2fol(B,Phi),
   callTheoremProver(Phi,Proof),
```

```
    (
     Proof=yes,
     nl,write('One reading is inconsistent!'),nl,
     !, fail
    ;
     Proof=no
    ).
```

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 `NewDrs` and `DrsSoFar` we translate the DRS



to first-order logic, say to a formula Φ. Then we try to prove that Φ is a theorem, by giving $\neg\Phi$ to Otter. If Φ is a theorem, we have no new information. Here, too, we need to take background knowledge into account. So the actual coding is as follows:

```
informative(drs([],[]),_):-!.
informative(DrsSoFar,NewDrs):-
   knowledge(Drs),
   mergeDrs(drs([],[merge(Drs,DrsSoFar)>NewDrs]),B),
   drs2fol(B,Phi),
   callTheoremProver(~Phi,Proof),
   (
    Proof=yes,
    nl,write('One reading is non-informative!'),nl,
    !, fail
   ;
    Proof=no
   ).
```

Let us now code the interface to Otter. This is done as follows: with `fol2otter/1` the formula is 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. It returns the value 26386 if Otter found a proof, and 26624 when it didn't succeed in finding a proof.

```
callTheoremProver(Formula,Proof):-
   fol2otter(Formula),
   shell('./otter < temp.in > temp.out',X),
   (X=26368,Proof=yes;X=26624,Proof=no).
```

Finally here's an example of background knowledge coded as a DRS. For the particular example we have in mind, we represent the following knowledge: no man is a woman, no woman is a man, if someone is called Marsellus then he is a man (and similarly for Vincent and Butch), and if someone is called Mia then she is woman. In DRS format this is:

```
knowledge(drs([],
              [drs([X],[man(X)]) > drs([],[~drs([],[woman(X)])]),
               drs([X],[woman(X)]) > drs([],[~drs([],[man(X)])]),
               drs([X],[X=marsellus]) > drs([],[man(X)]),
               drs([X],[X=vincent]) > drs([],[man(X)]),
               drs([X],[X=butch]) > drs([],[man(X)]),
               drs([X],[X=mia]) > drs([],[woman(X)])
              ])).
```

Her is an example session. (The input of the user is preceded by a >, the rest is output of the system.)

```
?- ask.

> Marsellus loves Butch and Mia.

> Mia is a boxer.

> Butch is a boxer.

> drs

Readings:
1 drs([A,B,C],
      [love(A,C),love(A,B),A=marsellus,B=mia,C=butch,boxer(B),boxer(C)])

> Marsellus loves one boxer.

One reading is inconsistent!

Discourse not accepted.

Starting again with empty discourse.
```

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 12.1.2 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.

12.2 Presupposition Accommodation

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 the 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. First we set up appropriate background knowledge:

```
knowledge(drs([],
              [drs([Y,Z],[have(Y,Z)]) > drs([],[of(Z,Y)]),
               drs([Y,Z],[of(Y,Z)]) > drs([],[have(Z,Y)]),
               ~ drs([X],[~drs([],
                               [drs([],[~drs([Y],[husband(Y),have(X,Y)])])
                              v drs([],[married(X)])])]),
               drs([X],[married(X)]) > drs([Y],[husband(Y),have(X,Y)])
              ])).
```

The next step is to extend out reading selection predicate with `localInformative/1`, which succeeds if all sub-DRS of its argument DRS contain new information with respect to super-ordinated DRSs.

```
selectReadings([],[]).
selectReadings([(DrsSoFar,Drs)|Potential],[Drs|Selected]):-
   consistent(Drs),
   informative(DrsSoFar,Drs),
   localInformative(Drs),
   !,
   selectReadings(Potential,Selected).
selectReadings([_|Potential],Selected):-
   selectReadings(Potential,Selected).
```

This new predicate uses `superSubDrs` to traverse through the DRS and compute pairs of super- and subordinated DRSs.

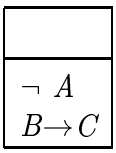
```
localInformative(Drs):-
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalInformative(List).
```

Then this list of pairs of DRSs is given to `allLocalInformative/1`, that calls the theorem prover in the way we're already familiar with. For each pairs of DRSs, two calls checks are performed: whether the sub-DRS follows from the superordinated DRS, and whether the negation from the sub-DRS follows from it.

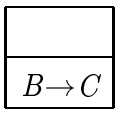
```
allLocalInformative([]).
allLocalInformative([(Super,Sub)|Others]):-
   knowledge(Drs),
   mergeDrs(drs([],[merge(Drs,Super)>Sub]),Drs1),
   mergeDrs(drs([],[merge(Drs,Super)>drs([],[~Sub])]),Drs2),
   drs2fol(Drs1,Psi),
   drs2fol(Drs2,Phi),
   (
    callTheoremProver(~Psi,Proof1),
    Proof1=yes,
    nl,write('One reading is (locally) non-informative!'),nl,
    !, fail
   ;
    callTheoremProver(~Phi,Proof2),
    Proof2=yes,
```

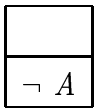
```
     nl,write('One reading is (locally) non-informative!'),nl,
     !, fail
    ;
     allLocalInformative(Others)
    ).
```

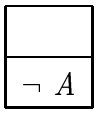
But what exactly these pairs of super-sub DRSs, and how are they computed (incidentally, Van der Sandt does not precisely define this). The best way to show the idea is giving an example. Assume we have the following DRS with sub-DRSs A, B, and C, structured as follows:



Then the pairs of super-sub DRSs are:

1. ( , A)

2. ( , B)

3. ( $\oplus$ B, C)

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).
```

There are a many important issues that this interim version of Chapter 12 has not so much as touched on. Once again, we advise interested readers to keep an eye on

`http://www/coli.uni-sb.de/~bos/comsem/`

for further developments.

Software Summary of Chapter 12

`discourseManager.pl` The main predicates that build the framework of our interactive discourse system. (page 360)

`discourse1.pl` The reader selector and knowledge base for the first experiment. (page 362)

`discourse2.pl` The reader selector and knowledge base for the second experiment. (page 364)

`fol2otter.pl` Translates a formula in otter syntax to standard output. (page 367)

Further Reading

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/`

Bibliography

Apt, K. (1995). *From Logic Programming to Prolog*. International series in computer science. Prentice Hall.

Asher, N. (1993). *Reference to Abstract Objects in Discourse*. Dordrecht: Kluwer.

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics*. North Holland.

Barendregt, H. (1991). Lambda calculi with types. In D. G. S. Abramsky and T. Maibaum (Eds.), *Handbook of Logic in Computer Science*, pp. 118–279. Oxford University Press.

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.

Bell, J. and M. Machover (1977). *A Course in Mathematical Logic*. North Holland.

Bos, J. (1993). VP Ellipsis and Presupposition in an Implementation of Discourse Representation Theory. Master's thesis, University of Groningen.

Bos, J., B. Buschbeck-Wolf, M. Dorna, and C. Rupp (1998). Managing information at linguistic interfaces. In *The 17th International Conference on Computational Linguistics*, Montreal, pp. ?–?

Bos, J., B. Gambäck, C. Lieske, Y. Mori, M. Pinkal, and K. Worm (1996). Compositional Semantics in Verbmobil. In *The 16th International Conference on Computational Linguistics*, Copenhagen, Denmark, pp. 131–136.

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.

Clocksin, W. and C. Mellish (1987). *Programming in Prolog* (Third, Revised and Extended Edition ed.). Springer Verlag.

Cooper, R. (1975). *Montague's Semantic Theory and Transformational Syntax*. Ph. D. thesis, University of Massachusetts.

Cooper, R. (1983). *Quantification and Syntactic Theory*. Dordrecht: Reidel.

Cooper, R., I. Lewin, and A. W. Black (1993). Prolog and Natural Language Semantics. Notes for AI3/4 Computational Semantics, University of Edinburgh.

D'Agostino, M. (1992). Are tabeleaux an improvement on truth tables? *Journal of Logic, Language and Information 1*, 235–252.

Dalrymple, M., J. Lamping, F. Pereira, and V. Saraswat (1997). Quantifiers, anaphora, and intentionality. *Journal of Logic, Language, and Information 6*, 219–273.

Deransart, P., A. Ed-Dbali, and L. Cervoni (1996). *Prolog: The Standard. Reference Manual*. Springer.

Dowty, D., R. Wall, and S. Peters (1981). *Introduction to Montague Semantics*. Dordrecht: Reidel.

Enderton, H. (1972). *A Mathematical Introduction to Logic*. New York and London: Academic Press.

Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving* (second ed.). Springer-Verlag.

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.

Gallier, J. (1986). *Logic for Computer Science*. New York: Harper Row.

Gallin, D. (1975). *Intentional and Higher-Order Modal Logic*. North Holland.

Gamut, L. (1991a). *Logic, Language, and Meaning. Volume I. Introduction to Logic*. Chicago and London: The University of Chicago Press.

Gamut, L. (1991b). *Logic, Language, and Meaning. Volume II. Intensional Logic and Logical Grammar*. Chicago and London: The University of Chicago Press.

Gazdar, G. and C. Mellish (1989). *Natural Language Processing in Prolog*. Addison-Wesley Publishing Company.

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.

Hähnle, R. and P. Schmitt (1994). The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning 13*, 211–221.

Heim, I. (1982). *The Semantics of Definite and Indefinite Noun Phrases*. Ph. D. thesis, University of Massachusetts.

Hindley, J. and J. Seldin (1986). *Introduction to Combinators and the Lambda Calculus*. London Mathematical Society Student Texts vol. 1. Cambridge University Press.

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.

Hobbs, J. R. and S. M. Shieber (1987). An algorithm for generating quantifier scopings. *Computational Linguistics 13*(1-2), 47–55.

Hodges, W. (1983). Elementary predicate logic. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosphical Logic. Volume I.*, pp. 1–131. Reidel.

Janssen, T. (1986a). *Foundations and Applications of Montague Grammar, Part 1: Philosophy, Framework, Computer Science*. CWI Tracts no. 19. Amsterdam: Centre for Mathematics and Computer Science.

Janssen, T. (1986b). *Foundations and Applications of Montague Grammar, Part 2: Applications to Natural Language*. CWI Tracts no. 28. Amsterdam: Centre for Mathematics and Computer Science.

Janssen, T. (1997). Compositionality. In J. van Benthem and A. ter Meulen (Eds.), *Handbook of Logic and Language*, pp. 417–473. Elsevier.

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.

Keller, W. R. (1988). Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer (Eds.), *Natural Language Parsing and Linguistic Theories*. Dordrecht: Reidel.

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.

Lewin, I. (1990). A Quantifier Scoping Algorithm without A Free Variable Constraint. In *Coling-90. Papers presented to the 13th International Conference on Computational Linguistics*, University of Helsinki, pp. 190–193.

McCune, W. and R. Padmanabhan (1996). *Automated Deduction in Equational Logic and Cubic Curves*. Lecture Notes in Computer Science (AI subseries). Springer-Verlag.

Moore, R. (1989). Unification-Based Semantic Interpretation. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver.

Muskens, R. (1996). Combining montague semantics and discourse representation. *Linguistics and Philosophy 19*, 143–186.

Partee, B. (1997). Montague grammar. In J. van Benthem and A. ter Meulen (Eds.), *Handbook of Logic and Language*, pp. 5–91. Elsevier.

Pereira, F. and S. Shieber (1987). *Prolog and Natural Language Analysis*. CSLI Lecture Notes 10. Stanford: Chicago University Press.

Ranta, A. (1994). *Type-Theoretical Grammar*. Oxford: Clarendon Press.

Reyle, U. (1993). Dealing with Ambiguities by Underspecification: Construction, Representation and Deduction. *Journal of Semantics 10*, 123–179.

Robinson, J. (1965). A machine oriented logic based on the resolution principle. *Journal of the ACM 12*, 23–61.

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.

Sundholm, G. (1983). Systems of deduction. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosphical Logic. Volume I.*, pp. 133–188. Reidel.

Sundholm, G. (1986). Proof theory and meaning. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosphical Logic. Volume III.*, pp. 471–506. Reidel.

Thomason, R. (1974). *Formal Philosophy. Selected Papers of Richard Montague*. New Haven: Yale University Press.

Turner, R. (1997). Types. In J. van Benthem and A. ter Meulen (Eds.), *Handbook of Logic and Language*, pp. 535–586. Elsevier.

Urquhart, A. (1995). The complexity of propositional proofs. *Bulletin of Symbolic Logic 1*, 425–467.

Van Deemter, K. and S. Peters (Eds.) (1996). *Semantic Ambiguity and Underspecification*. CSLI Lecture Notes 55. Stanford: Chicago University 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.

Part III

Appendices

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

$$\text{CUSTOMER(y)} \rightarrow \text{LOVES(VINCENT,y)}$$

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

$$\forall \text{y}[\text{CUSTOMER(y)} \rightarrow \text{LOVES(VINCENT,y)}]$$

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 $g(\text{x})$. 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 LOVES(VINCENT,MIA), we lose nothing if we replace them by simpler symbols such as p and q. Following this line of thought leads us to define *propositional languages*. To specify a propositional language, we first say which symbols we are going to start with. (A fairly standard choice is p, q, r, s, t, and so on, often decorated with superscripts and subscripts: for example, p'', r''', or q_2.) The chosen symbols are called proposition symbols, or atomic sentences. Complex sentences are built up using the standard Boolean connectives (for example, $\neg$, $\wedge$, $\vee$ and $\rightarrow$) in the obvious way, and the truth values of complex sentences are calculated using the familiar truth table rules.

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

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.(xx)x

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

(λx.(xx)x)(λx.(xx)x)

If we now apply β-conversion we get:

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

But this is just the functional application followed by an additional occurrence of $\lambda x.(xx)x$! 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?

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

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

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

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

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

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

order variables, which range over individuals, would now be thought of as the set of type e variables) together with the λ operator.

In the typed lambda 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 fairly straightforward way, the interpretation can be extended to cover all lambda expressions. For example, the expression λP.$\exists$x(WOMAN(x)$\wedge$P(x)) would be interpreted as a function which when given the type of function that P denotes as argument (and P denotes type $\langle e, t \rangle$ functions, like the characteristic function of MAN)

returns a type t value (that is, either TRUE or FALSE). Admittedly, thinking in terms of functions that take other functions as arguments and return functions as values can get pretty tedious, but the basic idea is straightforward, and for applications in natural language semantics, only a very small part of the function hierarchy tends to be used.

In short, throughout this chapter we have been talking about the lambda calculus as a mechanism for marking 'missing information', and we have exploited the mechanisms of functional application and 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.

Appendix C

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
```

By making use of the *cut* (written `!`) we can control the backtracking mechanism in a far more drastic way. The cut removes certain branches from the proof tree. If a cut is put in a clause, and Prolog encounters it during a proof, it removes all the clauses of the same predicate that are listed further down in the knowledge base, and moreover, removes all alternative solutions to conjuncts to the left of the cut in its clause.

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).