Prolog and Natural Language Semantics

Robin Cooper, Ian Lewin and Alan W Black

1992-93

Contents

Chapter 1

Introduction

We will explore how Prolog can be used to provide semantic interpretation in natural language processing. By semantics for natural language in this connection we understand not just the relating of a semantic representation language to natural language but the evaluation of natural language expressions with respect to databases encoded in Prolog. Evaluating a declarative sentence (on a given reading) with respect to a database involves determining whether the sentence is true with respect to the data base, whether the sentence appropriately describes the database. Evaluating a question with respect to a database might determine what information in the database would lead to appropriate answers to the question.

There are some central questions that we will discuss throughout. What kind of semantic representation language will we need, if any? How can this semantic representation language be related to a natural language in a regular way? What kind of databases are appropriate and how can the semantic representation language be evaluated with respect to the databases?

The first half of these notes deals with sentence semantics.

In order to get things started we will make some standard assumptions. We will choose first order predicate calculus as our semantic representation language. We will show how this language can be evaluated in the simplest kind of prolog data base and then show some standard logic programming techniques for relating natural language to predicate calculus.

We will then explore a more sophisticated logical language called the lambda calculus which has been widely used in computational semantics. This will lead us to the examination of some particularly difficult computational problems associated with quantified sentences like *most men have two children*.

We will complete the first half by considering the semantics of questions and looking at how questions and quantifiers are treated in CHAT-80, the best known natural language system developed in Prolog.

The second half of these notes deals with phenomena that go beyond the boundary of the sentence, i.e. discourse phenomena.

We will first look at cases of pronoun resolution in discourse and introduce the framework of discourse representation theory (DRT) which has recently gained considerable attention in computational semantics and is particularly interesting from a logic programming perspective.

We will then look at some more traditionally AI approaches to problems of pronoun resolution and discourse structure and see how they might be incorporated into the logic programming framework we have developed. In particular we will look at some uses of semantic networks for representing focus and attempt to create declarative versions of algorithms that have been proposed for determining pronoun resolution and discourse focus.

Finally, we will look at some of the AI work relating speech act theory to planning and language generation. A topic of particular importance here is the representation of belief. We will attempt to incorporate some of the classic AI work in this area into our general logic programming framework.

The purpose of this course is not just to understand some aspects of contemporary computational semantics but to have actual experience in writing Prolog programs that embody aspects

of semantics of natural language. As well as describing various semantic representation languages and how to translate simple sentences into these semantics forms, we will also want to know how write programs to achieve this.

1.1 Related texts

There is no one ideal text book to go with this course but there are a number of books which are useful as background reading. As well as books there are a number of papers which we will look at closely and should be read in order to understand that part of the course.

Some related books are

"Readings in Natural Language Processing" by Barbara Grosz, Karen Soarck Jones and Bonnie Lynn Webber (Morgan Kaufman 1986): available in AI SB library. This book is a collection of various "classic" papers in natural language processing, not just computational semantics. A useful book to have if you are intested in NLP.

"Mathematical Methods in Linguistics" by Barbara Partee, Alice ter Muelen and Robert Wall (Kluwer 1990). A very good background reference text. Covers many formal aspects of semantics and computation.

"Prolog and Natural Language Analysis" by Fernando Pereira and Stuart Shieber (CSLI lecture notes 1987). This book is closest to this course. The first few lectures are based on information in this book. Some of the early Prolog programs used come from this text.

"Introduction to Montague Semantics" by David Dowty, Robert Wall and Stanley Peters (Reidel 1981). A good introduction to Montague Grammar and the use of logic for semantic representation. Covers MG is far more detail than is required for this course but is well written and realtively readable.

Some specific papers are:

"An Algorithm for Generating Quantifier Scopings" by Jerry Hobbs and Stuart Shieber, *Computational Linguistics*, Vol. 13, Nos 1-2, pp 47-63 (1987).

"Discourse, anaphora and parsing" by Mark Johnson and Ewan Klein, COLING 86, pp. 669-75. A good introduction to Discourse Representation Theory.

"A theory of truth and semantic representation" by Hans Kamp (1981) in Formal Methods in the Study of Language, ed. by Jeroen Groenendijk, Theo Janssen and Martin Stokhof, Amsterdam: Mathematisch Centrum, pp. 277-322. This is the original article on DRT, good background but not necessary. Also see "From discourse to logic: Introduction to Model Theoretic Semantics of Natural Language, Formal logic and Discourse Representation Theory." by Hans Kamp and Uwe Reyle. This is now the definitive description of classical DRT. It is published by Kluwer Academic Publishers.

"Logic for Natural Language Analysis", by Fernando Pereira, Ph.D. thesis, University of Edinburgh. Reprinted as Technical Note 275, January 1983, AI Center, SRI International, Menlo Park, California. This describes the details of the CHAT-80 question answering system.

"An Efficient Easily Adpatable System for Interpreting Natural Language Queries" by David Warren and Fernando Pereira, *Computational Linguistics*, Vol. 8, Nos 3-4, pp 110-122 (1982). Again details of the CHAT-80 system.

"Focusing in the Comprehension of Definite Anaphora", by Candy Sidner, *Computational Models of Discourse*, eds. M. Brady and R. Berwick, pp 331-371. 1983. Also in Readings in Natural Language Processing. This is the original paper describing the focusing algorithm we dicuss in chapter 8.

Chapter 2

Predicate calculus and NL interpretation

2.1 Introduction

In this chapter we will introduce a syntax and semantics for predicate calculus, where by semantics we mean evaluation in a database. We will then show how to use a definite clause grammar to translate English into predicate calculus. Putting all this together we will have a program that will evaluate English sentences with respect to databases.

2.2 Syntax of predicate calculus

This discussion relates to the program `pc.pl` .

We will use prolog terms to represent formulae of predicate calculus. For example,

```
love(a,b)
~(love(a,b) & love(b,a))
```

We define prolog operators such as and `&` to be the connectives. The predicate `wff/1` will tell you whether an arbitrary prolog term is a well-formed formula of predicate logic:

```
?- wff(~(love(a,b) & love(b,a))).
    yes
```

We will use prolog variables to represent logical variables, although some purists will tell you that this is inadvisable. We will find it very useful, however, when we come to treat the semantics. Thus the following will be a well-formed formula

```
love(X,Y)
```

We use prolog functors to represent quantifiers:

```
every(X,man(X) ---> run(X))
```

This corresponds to a more usual predicate logic representation like

$\forall x[man(x) \rightarrow run(x)]$

Exercise 2.2.1

Use this program to determine whether the following are or are not well-formed formulae. Explain why they are accepted or not.

```
man(X) & run(X)
~love(a,b) ---> love(b,a)
~(love(a,b) ---> love(b,a))
run(a,b)
see(a,b)
every(X,man(X),run(X))
some(X,man(a)&run(a))
some(a,man(X)&run(Y))
some(X,man(X)&run(Y))
```

Do you think the program gets the right results in all cases?

Exercise 2.2.2

How would you modify the program so that it displays the structure of a wff that it accepts?

Exercise 2.2.3

Define a predicate `closed_wff` which accepts all and only those formulae which are sentences, i.e. contain no free variables. Or at least discuss what problems might arise in adding such a predicate to this program.

2.3 Programs:pc.pl

```
% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

% Definition of well-formed formulas

wff(For) :-
        For =..[P|Args],
        predicate(P,Args),
        terms(Args).
wff((For1 & For2)) :-
        wff(For1),
        wff(For2).
wff((For1 \/ For2)) :-
        wff(For1),
        wff(For2).
wff((For1 ---> For2)) :-
        wff(For1),
        wff(For2).
wff((~ For)) :-
        wff(For).

wff(For) :-
        functor(For,Q,2),
        quantifier(Q),
        For =.. [Q,V,For1],
        var(V),
        wff(For1).
```

```
% Terms are constants or variables

terms([]).
terms([X|L]) :-
        term(X),
        terms(L).

term(X) :- var(X),!.
term(X) :- constant(X).

% A sample lexicon.  Arity n represented by a list of
% length n.

predicate(run,[_]).
predicate(man,[_]).
predicate(woman,[_]).

predicate(love,[_,_]).

constant(a).
constant(b).
constant(c).


/* Quantifiers do not properly belong in the lexicon but
   we will want to add more quantifiers as we extend
   predicate calculus */

quantifier(every).
quantifier(some).


% Ancillary syntactic notion

closed_atomic_formula(For) :-
        For =.. [_|Args],
        constants(Args).

constants([]).
constants([Arg|Args]) :-
        \+ var(Arg),
        constant(Arg),
        constants(Args).
```

2.4 Semantics for predicate calculus

This discussion relates to the files `sits.pl` and `satisfy-pc.pl` .

We shall define a predicate `satisfy/2` which takes a database as its first argument and a formula of predicate calculus (as defined by the syntax in `pc.pl`) as its second argument. The first decision we have to make concerns the form that our databases will take. Here we will take the simplest possible form of database. We will take a database as being essentially a collection of closed atomic formulae of predicate calculus (alternatively, simple ground prolog terms with a

single functor and atoms as arguments). Thus we will allow our databases to contain clauses like

```
smile(a)
like(a,b)
```

but not

```
smile(X)
some(X,smile(X))
smile(a)&like(a,b)
~smile(a)
```

We will call our databases *situations* since a number of the ideas that we will develop as we progress are inspired by situation semantics—though they may be more conventionally be called *models*. We will assert clauses such as the following into the prolog database

```
sit1(woman(a)).
```

to indicate that `a` is a woman in database or situation `sit1` . Sometimes we will talk of `sit1` as supporting the fact `woman(a)` or of the fact `woman(a)` as holding in `sit1` . A sample situation can be found in the file `sits.pl` .

In the file `satisfy-pc.pl` we define a predicate `satisfy/2` which defines our semantics for predicate calculus. Its first argument is to be instantiated to a name of a situation, such as `sit1` , and its second argument is to be a formula of predicate calculus. Note that the program does not use the code we gave for defining the syntax of predicate calculus but that the formulae which can be interpreted by `satisfy/2` are precisely the well-formed formulae defined by the syntax with the exception that the program is not limited by any particular lexicon for predicate calculus.

We have chosen to give pretty much the most straightforward semantics for predicate calculus that one could think of, given our assumptions and the fact that we are using prolog. The simple case for atomic formulae is special in that it requires that the formula is a prolog term representing a fact that can be supported by a situation, i.e. that it is atomic and ground and checks to see whether it is supported by the situation. Thus we get the following behaviour:

```
?- satisfy(sit1, woman(a)).
    yes
?- satisfy(sit1, woman(X)).
    no
```

Note that this is different to what you might normally expect from a prolog system (where the free variables would correspond to existential quantification) and what you might normally expect from intepretations of predicate calculus (where free variables are often interpreted as universal quantification).

Conjunction and disjunction are interpreted as you might expect (e.g. a situation will satisfy `p&q` if it satisfies both `p` and `q`) and implication is interpreted equivalently to `~p\/q` (i.e. the material conditional). However, negation is not interpreted in the classical way since we use prolog's negation to interpret it. Since `sit1` does not contain a fact `blurg(w)` (and indeed no fact with either `blurg` or `w`) we get the following:

```
?- satisfy(sit1,~blurg(w)).
    yes
```

This means that the absence of information from a situation is taken to be negative information. We could change this by allowing negative facts to be included in our situations, but this would involve us in complications which are unnecessary for what we are going to do at the moment.

The treatment of quantification is perhaps one of the most useful aspects of this program, in that setting a spy point on `satisfy/2` enables us to see exactly how variables are getting bound and what is required for a quantified sentence to be satisfied. When we interpret quantified formulae we have to decide what domain we are quantifying over. This is not so important in the case of the existential quantifier since in order to interpret a formula like

```
some(X,run(X))
```

we just have to find some object which can be substituted for the `X` in `run(X)` so that the result is true in the relevant situation. In the case of the universal quantifier as in

```
every(X,run(X))
```

we have to determine that everything can be substituted for the `X` and that each of the resulting clauses are true in the relevant situation. But everything where? Everything in the whole universe? We shall take the domain to be everything in the situation which we are using to evaluate the formula. The predicate `domain/2` collects together everything which is an argument to a predicate in the situation and calls that the domain of the situation. It also makes a note in the database of what the domain of the situation is so that it does not have to compute it more than once. (Beware of this feature if you change a situation in the middle of a session. I have added a utility predicate `reset_domain/1` which will remove the information about the domain from the database.)

The rules for the quantifiers call the predicates `all/3` and `exists/3` . A call

```
all(X,[a,b], satisfy(sit1,run(X)))
```

will, for example, check that both of the following succeed:

```
satisfy(sit1,run(a))
satisfy(sit1,run(b))
```

Note the use of the prolog double negation in both `all/3` and `exists/3` . This is a trick which ensures that any unification within the scope of the negations will not go beyond the scope of those negations. Thus the variables will be able to unified with different atoms as is necessary in the case of `all/3` and they will remain as variables at the end of the computation so that the logical formula we started with will remain uninstantiated at the end of the computation.

Exercise 2.4.1

Set a spy point on `satisfy/2` . Try the following goals and explain what is going on in terms of the information provided by the traces.

```
satisfy(sit1,every(X,man(X)--->some(Y,woman(Y)&love(X,Y))))
satisfy(sit1,some(Y,woman(Y)&every(X,man(X)--->love(X,Y))))
```

Exercise 2.4.2

For each of the following formulae create a situation which satisfies it and another which does not. Use `satisfy/2` to verify your answer and explain what crucial characteristics each of the situations needs to have in order to allow or prevent satisfaction.

```
every(X,man(X)--->run(X))
every(X,~man(X)\/run(X))
~every(X,man(X)--->run(X))
every(X,man(X)&run(X))
some(X,man(X)&run(X))
some(X,man(X)--->run(X))
some(Y,woman(Y)&every(X,man(X)--->love(X,Y)))
```

Exercise 2.4.3

Suppose we were to add a quantifier two to our logic so that we would allow a sentence like

```
two(X,man(X))
```

How might you extend the definition of `satisfy/2` to accommodate this?

2.5 Programs:satisfy-pc.pl

```
% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).
:- dynamic domain/2.

% Satisfaction

satisfy(Sit, P&Q) :-
        satisfy(Sit,P),
        satisfy(Sit,Q).

satisfy(Sit, P\/Q) :-
        (satisfy(Sit,P); satisfy(Sit,Q)).

satisfy(Sit, P--->Q) :-
        (satisfy(Sit,~P); satisfy(Sit,Q)).

satisfy(Sit, ~P) :-
        \+ satisfy(Sit,P).

satisfy(Sit,some(X,P)) :-
        domain(Sit,Dom),
        exists(X,Dom,satisfy(Sit,P)).

satisfy(Sit,every(X,P)) :-
        domain(Sit,Dom),
        all(X,Dom,satisfy(Sit,P)).

satisfy(Sit,P) :-
        fact(P),
        support(Sit,P).

% Representation of situations

support(Sit,Fact) :-
        Clause =.. [Sit,Fact],
        Clause.

fact(Fact) :-
        Fact =.. [_|Args],
        atoms(Args).

atoms([]).
atoms([X|L]) :-
        atom(X),
        atoms(L).

% Quantifying over domains

domain(Sit,Dom) :-
```

```
        setof(X,
              Fact^P^Args^(support(Sit,Fact),
              Fact =..[P|Args],member(X,Args)),Dom),
        asserta((domain(Sit,Dom) :- !)).

all(_,[],_).
all(X,[Y|Dom],Clause) :-
        \+ \+ (X = Y,Clause),
        all(X,Dom,Clause).

exists(X,Dom,Clause) :-
        \+ \+ (member(X,Dom),Clause).

% Utility
reset_domain(Sit) :-
        retract((domain(Sit,_) :- !)).
```

An example situation

```
sit1(woman(a)).
sit1(run(a)).
sit1(man(b)).
sit1(love(b,a)).
sit1(man(j)).
sit1(love(j,a)).
sit1(man(c)).
sit1(woman(e)).
sit1(love(c,e)).
```

2.6 Translating English to predicate calculus

This section discusses the program `eng-pc.pl` . This is a dcg which parses English sentences into predicate calculus. It involves some rather subtle argument passing and is not the easiest of programs to understand. The complexity is caused by the mismatch between the syntax of quantified sentences in a natural language like English and their corresponding sentences in predicate calculus. Compare the structures

```
s(np(det(every),n(man)),vp(v(runs)))
every(X,man(X)--->run(X))
```

The English sentence breaks into two major constituents every man and runs whereas the logical sentence contains a constituent

```
man(X)--->run(X)
```

which does not correspond to any constituent in the English sentence. The complexity is even greater when we look at sentences with more than one quantified noun-phrase:

```
s(np(det(every),n(man)),vp(v(loves),np(det(a),n(woman))))
every(X,man(X)--->some(Y,woman(Y)&love(X,Y)))
```

Here what corresponds to the NP every man is the non-constituent of the logic

```
every(X,man(X)--->
```

and what corresponds to the NP a man is the non-constituent

```
some(Y,woman(Y)&
```

Thus we can think of the contribution of the NPs as being logical formulae with parts unspecified:

```
every(X,man(X)---> ...)
some(Y,woman(Y)& ...)
```

Actually, we know a bit more than is represented here. We know that the variable which is being quantified over (`X` and `Y` respectively) will occur somewhere in the part of the formula represented by '`...` '. Consider now what the contribution of the determiner without its following common noun would be:

```
every(X,___---> ...)
some(Y,___& ...)
```

Thus the contribution of a determiner is a template for a logical form which specifies a quantifier, a variable to be quantified and a connective with slots to be filled both to the left and right of the connective. This program uses prolog variables to represent the slots as well as the logical variables. You can see this by looking at the rules for determiners:

```
determiner(X,Range,Scope,every(X,Range ---> Scope)) -->
        [every].
determiner(X,Range,Scope,some(X,Range & Scope)) --> [a].
```

The template logical form is the last argument to the functor determiner. The slot to the left of the connective is labelled Range since intuitively the formula with which this will be unified will represent what is quantified over, the range of the quantifier, e.g. men, women, etc. The slot to the right of the connective is labelled Scope since this is intuitively what the NP will have scope over. The remaining variables are there to keep track of those parts of the logical form which other rules will need to specify without making reference to the internal structure of the logical form. Thus, for example, the rule that parses noun-phrases consisting of a determiner followed by a noun is:

```
noun_phrase(X, Scope, For) -->
    determiner(X, Range, Scope, For), noun(X, Range).
```

This will make sure that the variable in the logical form for the noun will be the same variable as that in the logical form for the determiner and that the logical form for the noun will become the range in the logical form for the determiner(i.e. the formula For). This logical form will furthermore be the logical form of the NP.

Exercise 2.6.1

Explain the treatment of quantified NPs (such as a man, every woman) in object position, e.g. the italicized NP in

every man loves *a woman*

Exercise 2.6.2

Explain the treatment of proper nouns such as *John* and *Mary*.

Exercise 2.6.3

The program provides a predicate `interpret/2` which allows calls of the form

```
interpret(Sentence,Logical_Form).
```

Which of the variables need to be instantiated? Can the program be used to generate English sentences from logical forms?

2.7 Combining the programs

It is straightforward to combine the programs `eng-pc.pl` and `satisfy.pl` so that we can evaluate English sentences in databases. For example, we might define:

```
satisfy_eng(Sit, Sent) :-
        interpret(Sent,Logic),
        satisfy(Sit,Logic).
```

Exercise 2.7.1

Show that the programs can be combined in this way, giving illustrative examples.

Exercise 2.7.2

The grammar treats two kinds of relative clauses: restrictive relative clauses (defined by the predicate `rel_clause`) and non-restrictive relative clauses (defined by the predicate `non_restr_rel_clause`). Explain what the syntactic and semantic differences are between these types of relative clause according to this program. Can you think of any ways in which this treatment of relative clauses is incomplete or inadequate?

2.8 Programs:eng-pc.pl

```
                    %Grammar with semantic representation
                              %Fernando Pereira
                               %Modified by RC

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

sentence(For) --> noun_phrase(X,Scope,For), verb_phrase(X,Scope).

noun_phrase(X,Scope,For) -->
    determiner(X,Range,Scope,For), noun(X,Range_noun),
    rel_clause(X,Range_noun,Range).
noun_phrase(X, Scope, For) -->
    determiner(X, Range, Scope, For), noun(X, Range).
noun_phrase(X,Scope,Scope&For_rel) -->
    proper_noun(X),
    non_restr_rel_clause(X, Scope, Scope&For_rel).
noun_phrase(X,For,For) --> proper_noun(X).

verb_phrase(X,For) -->
    trans_verb(X,Y,For_verb), noun_phrase(Y,For_verb,For).
verb_phrase(X,For) --> intrans_verb(X,For).

rel_clause(X,For,For&For_rel) --> [that], verb_phrase(X,For_rel).
non_restr_rel_clause(X,Scope,Scope&For_rel) -->
        [who], verb_phrase(X,For_rel).

determiner(X,Range,Scope,every(X,Range ---> Scope)) --> [every].
determiner(X,Range,Scope,some(X,Range & Scope)) --> [a].
```

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

proper_noun(john) --> [john].
proper_noun(mary) --> [mary].

trans_verb(X,Y,loves(X,Y)) --> [loves].

intrans_verb(X,lives(X)) --> [lives].

interpret(S, Logic) :- sentence(Logic, S,[]).
```

Chapter 3

The λ-calculus and natural language interpretation

3.1 Compositionality and the λ-calculus

In the program `eng-pc.pl` we did not associate a logical expression with each constituent of the English syntax. What we associated with many constituents was a partially specified template for a logical expression where we used prolog variables not only as logical variables but also as meta-variables holding a place for some piece of logical syntax. This is not compositional in the strict theoretical sense since it does not associate a logical expression with each English constituent. Furthermore it requires us to use additional arguments in our rules to keep track of all the variables which need to be passed around. The program is quite hard to understand and it is hard to develop large grammars in this style.

For these kinds of reasons it can be useful to use a version of the λ- calculus rather than predicate calculus in computing the logical form of a sentence. The λ- calculus is a richer logic than predicate calculus and allows us expressions which refer to arbitrary functions by taking an expression of the logic and prefixing it with λ and a variable, often referred to as λ-abstraction. For example,

$\lambda x[\text{run}(x)]$

is the function that will yield true when applied to an individual which is running. Similarly,

$\lambda x[\text{love}(x, \text{m})]$
$\lambda x[\text{love}(\text{m}, x)]$

will be functions that yield true for any individual that loves m or is loved by m respectively.

There are expressions of the language which represent the application of these functions to arguments.

$\lambda x[\text{love}(x, \text{m})](\text{a})$
$\lambda x[\text{love}(\text{m}, x)](\text{a})$

These expressions are equivalent to

love(a,m)
love(m,a)

respectively. This equivalence is referred to as β-reduction or λ-conversion.

There are two important things to notice about the λ-calculus. Firstly, we cannot only place λ's in front of formulas of the logic but any arbitrary expressions. Thus we can create complex λ-expressions with more than one λ.

$\lambda x \lambda y[\text{love}(x, y)]$
$\lambda y \lambda x[\text{love}(x, y)]$

These can be thought of as representing functions which, when applied to an argument, return another function as result. Thus the following equivalences hold in virtue of β- reduction.

$\lambda x \lambda y[\text{love}(x, y)](\text{a}) = \lambda y[\text{love}(\text{a}, y)]$
$\lambda y \lambda x[\text{love}(x, y)](\text{a}) = \lambda x[\text{love}(x, \text{a})]$

Notice that the order of the λ-variables makes a difference. By convention the parentheses are left associative so we have:

$\lambda x \lambda y[\text{love}(x, y)](\text{a})(\text{b}) \leftrightarrow \text{love(a,b)}$
$\lambda y \lambda x[\text{love}(x, y)](\text{a})(\text{b}) \leftrightarrow \text{love(b,a)}$

The other important thing is that we allow variables to range over anything, not just individuals and thus λ-expressions may be appropriate arguments to other λ-expressions. So, for example, we might translate the NP 'every man' by

$\lambda P[\forall x[\text{man}(x) \rightarrow P(x)]]$

and provide this with the argument

$\lambda y[\text{love}(\text{m}, y)]$

thus

$\lambda P[\forall x[\text{man}(x) \rightarrow P(x)]](\lambda y[\text{love}(\text{m}, y)])$

By two applications of β-reduction we would obtain the predicate calculus expression

$\forall x[\text{man}(x) \rightarrow \text{love}(\text{m}, x)]$

Notice that the variable P has taken over the role of the prolog variable `Scope` that was used in the program `eng-pc.pl` . Similarly we could translate the determiner 'every' by

$\lambda Q \lambda P[\forall x[Q(x) \rightarrow P(x)]]$

Exercise 3.1.1

What would the translation of the determiner 'a' be?

Exercise 3.1.2

Show how β-reduction can be applied successively to the following to obtain an expression of predicate calculus

$\lambda Q[\lambda P[\forall x[Q(x) \rightarrow P(x)]]](\lambda y[\text{woman}(y)])(\lambda z[\text{smile}(z)])$
$\lambda P[\forall x[\text{man}(x) \rightarrow P(x)]](\lambda y[\lambda P[\forall w[\text{woman}(w) \rightarrow P(w)]](\lambda z[\text{love}(y, z)])])$

How do you think these expressions might arise in the translation of English sentences?

3.2 A prolog version of the λ-calculus

This section relates to the program `lambda.pl` .

This program consists of an extension of our prolog version of predicate calculus and corresponds to the syntax used by Pereira and Shieber although they do not give a complete definition of their syntax. We will use the prolog operator '`^`' to represent 'λ' and place it after the variable. Thus

$\lambda x \lambda y[\text{love}(x, y)]$

will be represented by

```
X^Y^love(X,Y)
```

We will use the prolog operator '*' to represent application of λ-expressions to arguments. (Here we differ from Pereira and Shieber, who do not introduce any prolog notation for functional application.) Thus

$\lambda x \lambda y[\text{love}(x, y)](\text{a})(\text{b})$

is represented by the prolog term

```
X^Y^love(X,Y)*a*b
```

and NOT by the non-term

```
X^Y^love(X,Y)(a)(b)
```

λ-expressions, called λ-terms in the program, are admitted by rules which keep track of their arity, i.e. how many arguments they need to make a formula. Thus

```
lambda_term(X^For,1) :-
        wff(For).
```

says that a formula with a single λ-prefix is a λ-term with arity 1. Adding another λ-prefix to a λ-term with arity n creates a λ-term with arity $n + 1$. Similarly, there is a rule that says that a λ-term with arity n applied to an argument is a λ-term of arity $n - 1$, except in the case where $n = 1$, where the result is a formula (given by an additional rule for the predicate `wff/1` .

Note that there are no restrictions on what the argument can be other that to say that it is some kind of term and that the notion of term has been extended to include not only variables and constants but also formulae and λ-terms. This lack of restriction means that we are dealing with a version of the untyped λ-calculus rather than the typed λ-calculus. It is actually a version of the typed λ-calculus which has enjoyed wide application in natural language semantics, but it would make things a little more complicated than they need to be for the purposes of our current implementation if we were to code the type λ-calculus in prolog.

Exercise 3.2.1

Use the program to determine which of the following are well- formed formulae. Indicate the structure of those which are well-formed and explain what is wrong with those which are not well-formed. (Add appropriate constants and predicates to the lexicon so that the failure is not because a constant or predicate is missing or has the wrong arity.) Suggest English sentences which might correspond to the well-formed formulae, where appropriate.

```
some(Y,X^run(X)*Y)
want(j,every(X,Y^man(Y)&snore(Y)*X--->wake(X)))
want(j,every(X,Y^(man(Y)&snore(Y))*X--->wake(X)))
want(j,X^walk(X))
want(j,X^walk(X)&Y^talk(Y))
P
P*X
P*a*b
P^every(X,man(X)--->P*X)*Y^run(Y)
Q^P^every(X,Q*X--->P*X)*Y^student(Y)*Z^clever(Z)
Q^(P^every(X,Q*X--->P*X)*Y^student(Y))*Z^clever(Z)
X^fun(X)*Y^swim(Y)
X^fun(X)*Y^fun(Y)
X^Y^love(X,Y)*a
```

Exercise 3.2.2

Write a pretty printer that will display these formulae in a more readable form.

3.3 Implementing β-reduction with unification

This concerns the program `beta.pl` .

The heart of this is the one line program presented by Pereira and Shieber, which, using our formalism, is

```
reduce(X^P*X,P).
```

It unifies the variable in the λ-prefix with the argument and gives the λ-term stripped of its prefix in the second argument. While this is a splendid example of concise prolog we shall see that it is not without its problems.

The rest of this program, the definition of the predicate `convert/2` , is recursive on the syntax of the λ-calculus and ensures that appropriate conversions take place inside the phrase that we are reducing.

Exercise 3.3.1

Use the program to generate reductions of the formulae in the first exercise of the previous section. Do the reductions confirm your suggestions for corresponding English sentences? How much of a syntax check does the conversion do, can non-well-formed formulae be converted?

Exercise 3.3.2

We have not given a semantics for the λ-calculus as we did for predicate calculus, because we can rely on reducing to an expression of predicate calculus and then testing whether that is satisfied by some situation. How might you go about providing a semantics for the λ-calculus which would satisfy the constraint that any formula would be satisfied by exactly the same situations as its reduction?

3.4 Using the l-calculus for computing logical form

This section refers to the program `eng-lambda.pl` .

Using the λ-calculus allows us to give a program which gives a truly compositional semantics. We can assign a logical expression to each constituent of English and we no longer have need of templates with non-logical prolog variables like Scope and Range as we had in the previous program. Furthermore we have no need of extra arguments to keep track of variables since the λ-expressions do it for us. The result is a program that is much easier to read and much closer to a formalism that a linguist working in the Montague Grammar framework would employ.

For example, the rule introducing 'every' is

```
determiner(Q^P^every(X,Q*X ---> P*X)) --> [every].
```

Although the logical form is complex, it is immediately clear what the proposed semantics is. The rule for NPs consisting of determiners followed by nouns involves functional application as very many of the rules do now.

```
noun_phrase(Det*Noun) -->
    determiner(Det), noun(Noun).
```

Compare this with the corresponding rule in `eng-pc.pl` !

Exercise 3.4.1

Use the convert program to satisfy yourself that the program produces equivalent results to `eng-pc.pl` . Illustrate with key examples. Note that a call like

```
interpret(Sent,Logic),convert(Logic,Reduction).
```

will make `Logic` look absolutely horrendous since `convert/2` will do a lot of unification. If you want to see what the output of `interpret` really looks like call `interpret` without `convert` .

Exercise 3.4.2

Change the program so that it does β-conversion in each rule as the translation is built up rather than at the end of the parse. What might the computational advantage of doing it one way or the other be?

Exercise 3.4.3

Explain how transitive verbs are combined with their objects.

Exercise 3.4.4

Compare the treatment of proper names in this program with their treatment in `eng-pc.pl` .

Exercise 3.4.5

Complete the program so that it is equivalent to `eng-pc.pl` .

Exercise 3.4.6

How do things look for generating English sentences from logical forms now?

3.5 A problem with the implementation of β-reduction

Consider the following expression:

```
P^(P*j&P*m)*X^run(X)
```

By three applications of β-reduction, we should be able to reduce this to:

```
run(j)&run(m)
```

However, the predicate `convert/2` will fail on this example. It fails at the stage where we work on the intermediate reduction

```
X^run(X)*j&X^run(X)*m
```

We are now required to reduce both conjuncts. In working on the first conjunct the program unifies `X` with `j` and we obtain

```
run(j)&j^run(j)*m
```

There are two unfortunate things about this. Firstly, it is not a wff of the λ-calculus since in contains a λ-prefix which is not a variable. Secondly, when we come to try to reduce the right-hand conjunct we will attempt to unify `j` with `m` which will, of course, fail. In the real λ-calculus we would at this stage get

```
run(j)&X^run(X)*m
```

The variable in the λ-term in the right-hand conjunct is entirely independent of that in the left-hand. While we happen to have used the same symbol for the variables they are bound by different operators. In exactly the same way two occurrences of the same variable within the scope of different quantifiers are unrelated as in

$$\exists x F(x) \& \exists x G(x)$$

Using prolog variables as logical variables has got us into trouble here because the prolog variables are not getting bound in the way that logical variables do.

The solution to this problems involves making a copy of the λ-term with new variables and then doing the substitution involved in β-conversion. One has to be careful when doing this to make sure that the variables which are not being substituted for remain the same. A solution to this problems which makes clever use of `bagof/3` is provided in the program `reduce.pl` . This provides an alternative definition of the predicate `reduce/2` in `lambda.pl` . It is nothing like as elegant as the original definition of `reduce/2` , but nothing that solves this problems using prolog variables as logical variables is going to be toatally clean.

Exercise 3.5.1

Check that the program works with the example discussed in this section

Exercise 3.5.2

Why is it not possible to use double negation to solve this problem in a similar way that we used it in doing the semantics of predicate calculus?

3.6 Programs

These programs are for Chapter 3.

3.6.1 lambda.pl

```
% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

% Operator for functional application
:- op(400, yfx, *).

% Definition of well-formed formulas

wff(For) :-
        var(For),!.
wff(For) :-
        For =..[P|Args],
        predicate(P,Args),
        terms(Args).
wff((For1 & For2)) :-
        wff(For1),
        wff(For2).
wff((For1 \/ For2)) :-
        wff(For1),
        wff(For2).
wff((For1 ---> For2)) :-
        wff(For1),
        wff(For2).
wff((~ For)) :-
        wff(For).
wff((Lambda_term*Arg)) :-
        lambda_term(Lambda_term,s(0)),
        term(Arg).

wff(For) :-
        functor(For,Q,2),
        quantifier(Q),
        For =.. [Q,V,For1],
        var(V),
        wff(For1).

% Terms are constants or variables

terms([]).
terms([X|L]) :-
        term(X),
        terms(L).

term(X) :- var(X),!.
```

```
term(X) :- constant(X).
term(X) :- wff(X).
term(X) :- lambda_term(X,_).

% Lambda terms.
lambda_term(P,_) :-
        var(P),!.
lambda_term(X^For,s(0)) :-
        var(X),
        wff(For).
lambda_term(X^Lambda_term,s(N)) :-
        var(X),
        lambda_term(Lambda_term,N).
lambda_term(Lambda_term*Arg,s(N)) :-
        lambda_term(Lambda_term,s(s(N))),
        term(Arg).

% A sample lexicon.  Arity n represented by a list
% of length n.

predicate(run,[_]).
predicate(man,[_]).
predicate(woman,[_]).

predicate(love,[_,_]).

constant(a).
constant(b).
constant(c).

/* Quantifiers do not properly belong in the lexicon but
   we will want to add more quantifiers as we extend
   predicate calculus */

quantifier(every).
quantifier(some).

% Ancillary syntactic notion

closed_atomic_formula(For) :-
        For =.. [_|Args],
        constants(Args).

constants([]).
constants([Arg|Args]) :-
        \+ var(Arg),
        constant(Arg),
        constants(Args).
```

3.6.2 beta.pl

```
% beta.pl
```

```
/* Requires a lexicon for the lambda calculus */

% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

% Operator for functional application
:- op(400, yfx, *).

convert(P,P) :-
        var(P),!.
convert(P*A,P*A) :-
        var(P),!.
convert(X^P*A,Q) :- !,
        reduce(X^P*A,P1),
        convert(P1,Q).
convert(P*A,Q) :-
        convert(P,P1),
        % Prevent left recursion when P and P1 are identical
        \+ P = P1,!,
        convert(P1*A,Q).
convert(P&Q,P1&Q1) :- !,
        convert(P,P1),
        convert(Q,Q1).
convert(P\/Q,P1\/Q1) :- !,
        convert(P,P1),
        convert(Q,Q1).
convert(P--->Q,P1--->Q1) :- !,
        convert(P,P1),
        convert(Q,Q1).
convert(~P,~P1) :- !,
        convert(P,P1).
convert(P,Q) :-
        P =..[Quant,Var,Scope],
        quantifier(Quant),!,
        convert(Scope,NewScope),
        Q =..[Quant,Var,NewScope].
convert(P,Q) :-
        P =..[Pred|Args],
        predicate(Pred,Args),!,
        convert_all(Args,NewArgs),
        Q =..[Pred|NewArgs].
convert(P,P).

convert_all([],[]).
convert_all([P|L],[Q|L1]) :-
        convert(P,Q),
        convert_all(L,L1).

reduce(X^P*X,P).
```

3.6.3 eng-lambda.pl

```
%  Grammar with semantic representation
%
%  An incomplete modification of eng-pc.pl using
%  of the lambda-calculus

% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

% Operator for functional application
:- op(400, yfx, *).

sentence(NP*VP) --> noun_phrase(NP), verb_phrase(VP).

noun_phrase(Det*Noun) -->
    determiner(Det), noun(Noun).

noun_phrase(P^(P*X)) --> proper_noun(X).

verb_phrase(X^(NP*(Y^(TV*Y*X)))) -->
    trans_verb(TV), noun_phrase(NP).
verb_phrase(IV) --> intrans_verb(IV).

determiner(Q^P^every(X,Q*X ---> P*X)) --> [every].
determiner(Q^P^some(X,Q*X & P*X)) --> [a].

noun(X^man(X)) --> [man].
noun(X^woman(X)) --> [woman].

proper_noun(john) --> [john].
proper_noun(mary) --> [mary].

trans_verb(Y^X^love(X,Y)) --> [loves].

intrans_verb(X^live(X)) --> [lives].

interpret(S, Logic) :- sentence(Logic, S,[]).
```

3.6.4 reduce.pl

```
% Title: reduce.pl
% Authors:  Guy Barry and David Beaver

reduce(X^P*Y,Q) :-
        sub(Y,X,P,Q).

% sub(Val,Var,InTerm,OutTerm) :-
%        OutTerm is a copy of InTerm where the variable
%       Var has been  replaced by the term or variable
```

```
%       Val and all other variables remain unchanged
%       (given that Var does not occur in Val).
sub(Val,Var,InTerm,OutTerm) :-
        var(Var),
        bagof(  Term,
                Var^(Var=Val,Term=InTerm),
                [OutTerm] ).
```

Chapter 4

Quantification

4.1 Introduction

In this chapter we will look at two central problems having to do with quantifiers in natural language. One has to do with how we can extend our semantics to include quantifiers other than $\exists$ and $\forall$, the standard quantifiers of predicate calculus. The other has to do with the intepretation of scope ambiguities in natural language.

4.2 Generalized quantifiers

This discussion relates to the program `satisfy-gq.pl` . In addition to the determiners 'a' and 'every' English contains a number of determiners that represent different quantifiers. Those we treat here (at least in terms of logical representation if not in terms of English syntax) are: 'no', 'one', 'two', 'exactly n', 'at least n', 'at most n', 'many', 'few' and 'most'. These are called generalized or non-standard quantifiers because they go beyond the classical existential and universal quantifiers. The term "generalized quantifier" also refers to a particular semantic analysis of quantifiers which is necessary if we are to comprehend these quantifiers as well as the existential and universal quantifiers in a general framework.

One way to view the basic insight of the generalized quantifier analysis is that quantification involves the comparison of two sets. Consider the determiner 'most'. If we try to symbolize the sentence 'most men run' in the kind of logical form that we have been considering so far, we find that there is no connective which is appropriate:

```
most(X,man(X) ?  run(X))
```

It is simply not possible to think of most as quantifying over the whole domain: e.g. "most x are such that x is a man and x runs" or "most x are such that if x is a man then x runs" are simply not appropriate paraphrases of 'most men run' in the way that "for every x if x is a man then x runs" is appropriate for 'every man runs'. The sentence with 'most' has to be viewed as comparing two sets rather than making a statement about the whole domain. It compares the set of men with the set of runners and requires that some suitably large proportion of the members of the first set (say more than half) are also members of the second set. No logical connective will express this for us, so we will adopt a notation for quantified sentences which does not involve the use of a connective:

```
most(X,man(X),run(X))
```

As it happens this is a common notation in the logic programming literature (first suggested by Colmerauer) where it is given the name three-branch quantifier (3BQ).

The notion of set comparison in the interpretation of quantifiers is as good for standard quantifiers as it is for the non-standard ones. So, for example, we can think of the interpretation of

'every man runs' as requiring us to check that the set of men is a subset of the set of runners. So we can use the same notation for all quantifiers, e.g.

```
every(X,man(X),run(X))
```

In the program we include both versions of 'some' and 'every' for comparison. In these formulae we will call the second argument of the quantifier term the *range* and the third argument of the quantifier term the *scope*.

The program `satisfy-gq.pl` is not optimal from an efficiency point of view since it is supposed to emphasize that all natural language quantifiers submit to a general analysis. However, is does exploit one important concept from generalized quantifier theory which has a consequence for computation. This is the concept of conservativity. A quantifier q is conservative just in case the following holds:

`q(X,P,Q)` $\leftrightarrow$ `q(X,P,P&Q)`

It seems that all natural language quantifiers are conservative in this sense. Here are some English examples which illustrate this:

many men run $\leftrightarrow$ many men are men who run
few women sneeze $\leftrightarrow$ few women are women who sneeze
every child smiles $\leftrightarrow$ every child is a child who smiles

Computationally what this means is that in determining satisfaction we do not have to check all of the scope set but only its intersection with the range set. Furthermore nearly all of the quantifiers fall into the following two categories:

absolute: you have to check the intersection for some cardinality property

relative: you have to compare the intersection with the range set

For example, 'one' is absolute since you have to find the intersection of the range and scope sets and check the cardinality of the intersection. On the other hand, 'most' is relative since you have to find the intersection of the range and scope sets and check that a certain proportion of the members of the range set are in the intersection. This means that the computation of satisfaction of some formula will normally involve two steps

1. compute the intersection of the range and scope sets
2. either check the cardinality of the result or compare the intersection with the range set

All of the quantifiers treated in this program fall into this mould.

Some quantifiers are vague. That is, you cannot tell simply from the sentence how precisely to check the relevant sets. Consider 'many'. If I say

many men run
many men in Scotland run
many men in the class run

I probably have a different idea of what might constitute 'many' in each case. Even given some precise context I may not be able to say precisely what I mean by 'many'. Does 20,000 Scots count as many but not 19,999? Yet we have some idea that different "ball-park" figures are involved in the different sentences. A famous sentence from the linguistic literature (due to Barbara Partee) is

many men date many women

This shows that we may have different notions of what constitutes many for different uses of 'many' in the same sentence. It is difficult to know what to do for such quantifiers in computational applications. Consider what kind of world knowledge might have to built in in order for a practical

data-base query system to be able to sensibly answer a question like 'Do many men date many women?'. The solution taken in the present program is to build in an extra parameter in the quantifier expression which represents what counts as many, few or most. In the case of many and few it will be some absolute number, but in the case of most it will be some proportion like 0.7 or 9/10. We assume that this number is given us by some other (intelligent) module. Note, however, that such a module is unlikely to capture our intuitions about these sentences exactly since the quantifiers are truly vague - we cannot say precisely what number we have in mind for many even when given a precise context. Perhaps a useful alteration to this program would be one which uses something other than the arithmetic in standard Prolog and would allow us to leave the extra parameter as a variable when we call `satisfy` . (This will not work in the present program.) In this way it could place constraints on what the extra parameter could be. For example, if we have a situation where five men are running and we ask whether

```
many(Num,X,man(X),run(X))
```

is satisfied, it might instantiate `Num` to `5` and then leave the user to determine whether five counts as many or not.

An important property that quantifiers may have is *monotonicity*. Consider that the English VP 'arrived early' represents a subset of what is represented by 'arrived'. Now note that

every man arrived early → every man arrived

but NOT

no man arrived early → no man arrived

We say that 'every' is monotone increasing on its second argument, i.e.

if Q represents a subset of Q'
`every(X,P,Q)` → `every(X,P,Q')`

Notice also that

no man arrived → no man arrived early

but NOT

every man arrived → every man arrived early

The quantifier 'no' is monotone decreasing on its second argument, i.e.

if Q represents a subset of Q'
`no(X,P,Q')`→ `no(X,P,Q)`

The quantifier 'every' is monotone decreasing on its first argument, i.e.

if Q represents a subset of Q'
`every(X,Q',P)` → `every(X,Q,P)`

For example,

everybody who came drank a vodka martini → everybody who came early drank a vodka martini

Exercise 4.2.1

How must the program `pc.pl` be modified to accommodate the new structure of quantified formulae?

Exercise 4.2.2

Which of the quantifiers treated in `satisfy-gq.pl` are absolute and which are relative? Explain why.

Exercise 4.2.3

Check that the semantics given in satisfy-gq.pl respects the following properties of quantifiers, give example situations to illustrate these properties and explain why it is not possible to construct a situation which would satisfy a sentence that would violate the property.

monotone increasing on second argument - 'some', 'many', 'every'
monotone decreasing on second argument - 'no', 'few', 'at most 6'
monotone increasing on first argument - 'some', 'many'
monotone decreasing on first argument - 'every', 'no'

Exercise 4.2.4

Extend the program `eng-lambda.pl` (or your current extension of it) to include the quantifiers treated in `satisfy-gq.pl` . Discuss the choices you made with respect to numerals (like 'two') and vague quantifiers.

Exercise 4.2.5

Incorporate quantifiers corresponding to the and both into the program
`satisfy-gq.pl`

4.3 Programs

These programs are for Chapter 4

satisfy-gq.pl

satisfy-gq.pl

```
satisfy(Sit,P) :-
         % closed_atomic_formula(P),
        support(Sit,P).

satisfy(Sit, P&Q) :-
        satisfy(Sit,P),
        satisfy(Sit,Q).

satisfy(Sit, P\/Q) :-
        (satisfy(Sit,P); satisfy(Sit,Q)).

satisfy(Sit, P--->Q) :-
        (satisfy(Sit,~P); satisfy(Sit,Q)).

satisfy(Sit, ~P) :-
        \+ satisfy(Sit,P).

satisfy(Sit,some(X,P)) :-
        domain(Sit,Dom),
        \+ \+ (member(X,Dom),satisfy(Sit,P)).

satisfy(Sit,some(X,P,Q)) :-
        setof(X,(satisfy(Sit,P),satisfy(Sit,Q)),_).

satisfy(Sit,no(X,P,Q)) :-
        \+ setof(X,(satisfy(Sit,P),satisfy(Sit,Q)),_).
satisfy(Sit,one(X,P,Q)) :-
         setof(X,(satisfy(Sit,P),satisfy(Sit,Q)),[_]).
satisfy(Sit,two(X,P,Q)) :-
         setof(X,(satisfy(Sit,P),satisfy(Sit,Q)),[_,_]).
satisfy(Sit,exactly(Num,X,P,Q)) :-
         setof(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         length(Set,Num).
satisfy(Sit,at_least(Num,X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         length(Set,N),
         N >= Num.
satisfy(Sit,at_most(Num,X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         length(Set,N),
         N =< Num.

satisfy(Sit,every(X,P)) :-
        domain(Sit,Dom),
        all(Dom,X,satisfy(Sit,P)).
```

```
satisfy(Sit,every(X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         set_of(X,satisfy(Sit,P),Set).

satisfy(Sit,many(Num,X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         length(Set,N),
         N >= Num.
satisfy(Sit,few(Num,X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         length(Set,N),
         N =< Num.
satisfy(Sit,most(Prop,X,P,Q)) :-
         set_of(X,(satisfy(Sit,P),satisfy(Sit,Q)),Set),
         set_of(X,satisfy(Sit,P),Range),
         length(Set,N),
         length(Range,NR),
         N/NR >= Prop.

domain(Sit,Dom) :-
        setof(X,
              P^Fact^Args^(support(Sit,Fact),
              Fact =..[P|Args],member(X,Args)),Dom),
        asserta((domain(Sit,Dom) :- !)).

support(Sit,Fact) :-
        Clause =.. [Sit,Fact],
        Clause.


all([],_,_).
all([X|Dom],Y,Clause) :-
        \+ \+ (X = Y,Clause),
        all(Dom,Y,Clause).

set_of(X,Goal,Set) :-
        setof(X,Goal,L),!,
        L = Set.
set_of(_,_,[]).
```

Chapter 5

Quantifier Scope

5.1 The Problem(s)

The program **eng-lambda.pl** produces the following logical expression for *a woman loves every man*

some(X,woman(X) & every(Y,man(Y) —> love(X,Y)))

The sentence also appears to have the following (perhaps more natural) reading

every(Y,man(Y) & some(Y,man(Y) —> love(X,Y)))

The difference between these two expressions lies in the relative *scopes* of the two quantifiers **every** and **some**. In the first, **some** outscopes **every** and in the second **every** outscopes **some**. Ambiguities which arise in this fashion are called *quantifier scope ambiguities*.

Scope ambiguities arise with all sorts of other operators apart from quantifiers. *The prime-minister used to be a woman* might be taken to be true because it used to be the case that Margaret Thatcher was prime-minister. Alternatively the sentence might be making an outrageous claim about John Major.

Here we will concentrate largely on quantifier scope ambiguities.

The task of deciding which quantifiers should take scope over which others is a notoriously hard problem. Our example shows that simply giving the subject np wide scope (as in **eng-lambda.pl**) does not always produce the most plausible answer.

Clearly, contextual influences and general world knowledge play a large part in scope determination

1. A TV set blurted out that each senator was offended

2. A quick test confirmed that each drug was psychoactive

It is hard to imagine that for each senator a different TV set blurted out that he was offended but quite easy to imagine that each drug underwent a different test.

On the other hand, syntactic structures certainly have some influence. In some tests conducted by Vanlehn ('Determining the Scope of English Quantifiers' M.Sc. thesis at MIT 1978), small variations in sentences with a very similar content sometimes produced quite divergent reactions.

1. At the conference yesterday, I managed to talk to a guy who is representing each raw rubber producer in Brazil

2. At the conference yesterday, I managed to talk to a guy representing each raw rubber producer in Brazil

3. At the conference yesterday, I managed to talk to a representative from each raw rubber producer in Brazil

Nobody thought **each** outscoped **a** in 1, everyone thought that **each** outscoped **a** in 3 and people were evenly divided on 2.

The algorithm encoded in **qscope.pl** was devised by Hobbs and Shieber and is described in their paper 'An Algorithm for Generating Quantifier Scopings' (Computational Linguistics Volume 13 nos. 1-2). Some very low level routines which were not detailed in their paper have been added.

The program avoids the issue of trying to decide on the most plausible reading of a sentence - instead it is designed to generate all (and only) the *possible* readings of a sentence. Another program might be used to help decide which of the possible readings is actually the most likely one. Alternatively, one could try asking the user to decide amongst them but this is not a particularly attractive option. Ordinary speakers would have to be able to read and understand quite complex logical expressions. Furthermore, the number of readings generated can be huge. For the sentence

> *Some representative of every department in most companies saw a few samples of each product*

there are calculated to be 42 different readings! It would be somewhat impractical to ask a user to decide which of 42 different readings was intended by the sentence he or she just used.

Exercise 5.1.1

Sometimes different logical forms corresponding to different quantifier scopings are equivalent in the sense that any situation that satisfies one will satisfy the other and any situation which does not satisfy one will also not satisfy the other. For the following sentences, determine what the two logical forms associated with them would be and test whether they are equivalent or not. If they are not, show examples of situations that satisfy one and not the other and if possible the reverse too—does either quantifier scoping imply the other.

a) every man loves a woman
b) two women saw two samples
c) a child saw a dog
d) most students like a few professors
e) most professors like most students

5.2 Hobbs and Shieber's algorithm

In this section, we will describe the heart of Hobbs and Shieber's algorithm. This algorithm is implemented in the program **hs-one.pl**. In the next section, we will discuss an amendment which deals with some more complex cases of quantifier scoping. The amended program can be found in **hs-two.pl**. The full implementation, which deals with a wider range of issues than just quantifier scoping, can be found in **qscope.pl**.

hs-one.pl is designed to convert a formula which encodes predicate-argument relations, but *not* quantifier scopings, into a formula in the three-branch quantifiers notation of Generalized Quantifier Theory.

The program takes initial input of the following form:

A **wff** is of form *wff(predicate,* $[arg_1 \ \ldots \ arg_n]$*)*, where *predicate* is an atomic predicate symbol and the arg_i are constants, variables or complex terms. A complex term is of form *term(quant,variable,wff)*, where *quant* is one of 'every', 'a'. 'most' etc.

Our example *a woman loves every man* becomes

wff(loves,[term(a,x,wff(woman,[x])),term(every,y,wff(man,[y]))])

This representation does show predicate-argument relations (*e.g.* that the noun phrase *a woman* is the first argument to the predicate *loves*) but does not show which quantifier takes wider scope.

The H&S representation for three-branch quantifiers is this:

wff(quant,$[var, wff_1, wff_2]$*)*

The H&S algorithm contains at the top-level a simple recursion called *apply_terms*, which takes

an input form as argument and returns a scoped form as its result. The algorithm simply picks a complex term, *applies* it, picks another and so on —until there are no more complex terms in the formula.

apply_terms

```
if    there are no complex terms in the input wff,
then  terminate and return input wff
else  choose an applicable complex term (ct) in the wff
      apply ct to the wff generating a new wff
      recurse on the new wff
```

For the moment, we will say that an *applicable* complex term is just any complex term in the wff.

It is important that the choice of which complex term to apply next to a formula is *non-deterministic*. The different orders in which one chooses complex terms will correspond to different quantifier scopings for the formula.

The step of *applying* a complex term to a wff is equally straightforward. *apply* has two arguments: a complex term to be applied and a form to apply it to. *apply* also returns an output formula. *apply* has three steps

apply

1 let the complex term be *term(q_1,v_1,wff_1)*
2 let **tmp-wff** be the result of substituting v_1 for term(q_1,v_1,wff_1) in the input wff
3 return *wff(q_1,[v_1,wff_1,tmp-wff])*

That is, the result of applying a complex term to a wff is a new wff whose first 3 parts are just copied from the complex term and whose fourth part is the result of a simple substitution operation.

Exercise 5.2.1

Show that by choosing the complex terms in either order, the two readings of *some woman loves every man* are obtained.

5.3 Nested Noun Phrases

Sometimes noun phrases contain other noun phrases. Sentences containing these *nested* noun phrases may also be subject to quantifier scope ambiguities.

Every representative of a company arrived

Exercise 5.3.1

Write down the two readings of this sentence in three-branch quantifier notation

Suppose we represent this sentence in H&S notation as follows
wff(arrived,[term(every,x,wff(rep-of,[x,term(some,y,wff(cmp,[y]))]))])
If we call *apply_terms* on this example, then one of the readings generated will contain a *free* variable `y` which is not bound by the quantifier over `y`. The formula does not correspond to a reading of the original sentence. We will not be able to test whether a particular situation satisfies this formula or not - because we are not told anywhere what the value of `y` should be.

Exercise 5.3.2

Given the representation of *Every representative of a company arrived* shown above, which order of applying complex terms results in the output formula containing a free variable ?

The solution to the problem of nested noun phrases is contained in **hs-two.pl**. First, we alter the top-level recursion of *apply_terms* and add a flag which will cause the program to return *partially scoped* answers as well as *fully scoped* answers. The flag is called **Complete** and if its value is **false**, then partial answers will be returned.

The new definition of *apply_terms* is as follows

apply_terms
if there are no complex terms in the input wff,
then terminate & return input wff
else (**either if Complete = false**
then terminate & return input wff
else fail
or choose an *applicable* complex term (**ct**) in the wff
apply **ct** to the wff generating a new wff
recurse on the new wff with **Complete** as set on input)

The **either-or** clause is also *non-deterministic*. We can choose either disjunct to execute - though if the branch we choose fails, then we must try the other, as usual. The idea is that if the input formula does contain complex terms and **Complete** = **false**, then we can simply choose to stop the recursion immediately with whatever partial result we have found so far. Alternatively, we can choose to apply another complex term within the wff.

If **Complete** = **true** then even if we did choose the first disjunct that disjunct would fail so we would end up taking the second disjunct anyway.

Exercise 5.3.3

What are the results of calling *apply_terms* (with **Complete** = **false**) on a suitable input representation of *'some woman loves every man'* ?

There is also a new definition of an *applicable* complex term. Before, any complex term within a formula was applicable. Now we restrict the *applicable* complex terms to be only those complex terms which are *not* nested within other complex terms. So given a representation of *Every representative of a company arrived*, we could not now choose to apply *a company* first because it is nested within *every representative of a company*.

The last change we make is to the definition of *apply* itself.

apply
1 let the complex term be *term(q_1,v_1,wff_1)*
2 let **tmp-wff**$_1$ be the result of substituting v_1 for term(q_1,v_1,wff_1) in the input wff
3 call *apply_terms* on wff_1 (with **Complete** = **false**) returning **tmp-wff**$_2$
4 return *wff(q_1,[v_1,tmp-wff$_2$,tmp-wff$_1$])*

This is just the same as before except that instead of simply copying wff_1 into the output representation, we now recurse on *apply_terms* first, whilst also setting the flag **Complete** to **false**.

A Worked Example

Given the new definitions of *apply_terms* and *apply* and setting **Complete** = **true**, call *apply_terms* on
wff(arrived,[term(every,x,wff(rep-of,[x,term(some,y,wff(cmp,[y]))]))])
There are complex terms in the formula so we do not terminate. Furthermore, **Complete** = **true** so we must execute the second disjunct of the **either-or** clause.

Now we must choose an applicable complex term but there is only one– *every representative of a company* – (because the other is *nested* and therefore not applicable).

Now we must apply *every representative of a company* to our input wff. First we substitute `x` for *every representative of a company* thereby generating
wff(arrived,[x])

Secondly, we call apply_terms (this time with **Complete** = **false**) on the wff within *every representative of a company*, *i.e.*

wff(rep-of,[x,term(some,y,wff(cmp,[y]))])

There are complex terms in this wff and furthermore **Complete** = **false** so we could choose the first disjunct of the **either-or** clause. If we did this we would terminate this recursion and simply return

wff(rep-of,[x,term(some,y,wff(cmp,[y]))])

unchanged. However, we will choose not to do this, and instead we will take the other branch of the **either-or** clause.

First, we choose a complex term in the wff (there is only one): *a company*. Next, we substitute `y` for *a company* in our current wff, generating

wff(rep-of,[x,y])

Then we call *apply_terms* on 'wff(cmp,[y])' - but since there are no complex terms in this wff, the result is just 'wff(cmp,[y])'. We can now return the result of recursing on *representative of a company*, *i.e.*

wff(some,[y,wff(cmp,[y]),wff(rep-of,[x,y])])

Finally, we can now return the value of our original call to *apply* on *every representative of a company*

wff(every,[x,wff(some,[y,wff(cmp,[y]),wff(rep-of,[x,y])]),wff(arrived,[x])])

Having called *apply* from within *apply_terms* we must now recurse once more on *apply_terms* itself - but since there are no more complex terms within the wff the whole recursion terminates with the above result.

The result is that 'every' has wide scope and there are no free variables.

Exercise 5.3.4

Prove to yourself that, had we chosen the *second* disjunct of the **either-or** clause at a point in the above example, the other reading of the sentence would have been obtained. (Note: don't forget the final recursive call to *apply_terms* after you've finished applying *every representative of a company*)

5.4 Other Scope Ambiguities

The full algorithm of Hobbs and Shieber is in **qscope.pl**. That algorithm not only deals with quantifier scope ambiguties but also with some other scope ambiguities. For example,

Five students aren't here

is ambiguous between the following two readings:

$5x$ (students(x), $\sim$ here(x))

$\sim 5x$ (students(x), here(x))

By adding the clause `opaque(not,1)` to the program, the predicate `not` is declared *opaque* in its first argument. That is, quantifiers can take scope inside the scope of negation as well as outside.

Sentences involving verbs such as 'hopes' and 'believes' (often called *propositional attitudes*) are also ambiguous in interesting ways.

Jack believes every famous impressionist painter is French

is ambiguous between the following two readings

believes(jack,$\forall x$ (imp-painter(x),french(x)))

$\forall x$ (imp-painter(x),believes(jack,french(x)))

On the first reading, Jack has the general belief that any famous impressionist painters will in fact be French - and he need not have any beliefs concerning particular painters at all. For example, he need not believe that Alfred Sisley is French, because he isn't aware that Sisley *is* a famous impressionist painter. On the second reading, Jack believes of each individual who happens to be a famous impressionist painter that that individual is French. Therefore he believes Sisley is French. However, he need not have the *general* belief – he might not believe anything at all about impressionist painters in general.

Exercise 5.4.1

Write a suitable clause declaring that 'believes' is opaque. Test the program on some sentences containing the verb 'believes'.

Exercise 5.4.2

Find some other predicates that show interesting scoping behaviour. Experiment with **qscope.pl** using the examples you can find.

Exercise 5.4.3

Test **qscope.pl** on examples of sentences that include different *combinations* of scope-bearing words (for example, 'every','not','believes'). Comment on the plausibility of the results.

5.5 Programs: hs-one.pl

```
% gen(Form, ScopedForm)
% ======================
%  Form        ==> a wff with in-place complex terms
%  ScopedForm  <== a full scoping of Form

gen(Form, ScopedForm) :-
        apply_terms(Form, ScopedForm).

% apply_terms(Form, ScopedForm)
% =====================================
%  Form        ==> a wff with in-place complex terms
%  ScopedForm  <== a full scoping of Form
%
%  Applies one or more terms to the Form alone

apply_terms(Form, Form) :-
        not(term(Form, Term)), !.

apply_terms(Form, ScopedForm) :-
        applicable_term(Form, Term),
        apply(Term, Form, AppliedForm),
        apply_terms(AppliedForm, ScopedForm).

% apply(Term, Form, NewForm)
% ==========================
%  Term    ==> a complex term
%  Form    ==> the wff to apply Term to
%  NewForm <== Form with the quantifier wrapped around it

apply(term(Quant, Var, Restrict),
                Body,
                wff(Quant, [Var, Restrict, OutBody])) :-
        subst(Var, term(Quant, Var, Restrict), Body, OutBody).

% applicable_term(Form, Term)
% ===========================
% Form  ==> an expression in the logical form language
```

```
% Term  <== any term in Form
%
% In this version of the program, we simply allow any
% Term contained within Form to be applicable - so we
% can just define it by means of the term/2 already
% used in apply_terms/2

applicable_term(Form, Term) :-
        term(Form,Term).

/************************************************************

My implementation of the low-level predicates not included
in the appendix of the paper. (all 3 programs identical
below this line)
************************************************************/

% quantifer(Quant)
% ================
% Quant ==> a valid quantifier

quantifier(every).
quantifier(most).
quantifier(some).
quantifier(each).
quantifier(a_few).

% subst(New, Old, OldForm, NewForm)
% ================================
% New         ==> A pattern to substitute for Old
% Old         ==> A pattern to be replaced by New
% OldForm     ==> a wff with in-place complex terms
% NewForm     <== OldForm with each occurrence of Old
%                     replaced by New

subst(New, Old, Old, New) :- !.
subst(New, Old, wff(Quant, ArgList),
      wff(Quant, NewArgList)) :- !,
        subst(New, Old, ArgList, NewArgList).
subst(New, Old, term(Quant, Var, Restrict),
                term(Quant, Var, NewRestrict)) :- !,
        subst(New, Old, Restrict, NewRestrict).

subst(B, A, [A|T], [B|NT]) :- !,
        subst(B, A, T, NT).
subst(B, A, [H|T], [NH|NT]) :- !,
        subst(B, A, H, NH),
        subst(B, A, T, NT).
subst(B, A, Form, Form).

% term(Form, Term)
% ================
% Form       ==> a wff or complex term
% Term       <== a complex term contained in Form
```

```
%
% Extracts a term from Form.

% If Form is a wff, a term of form is a term of
% its argument list.
term(wff(Pred, ArgList), Term) :-
        term(ArgList, Term).

% If Form is a term, then it is a term.
term(term(Quant, Var, Restrict),
     term(Quant, Var, Restrict)).

% If Form is a term, then a term is a term of its
% restriction.
term(term(Quant, Var, Restrict), Term) :-
        term(Restrict, Term).

% If Form is an argument list, a term is a term of
% its head or of its tail.
term([H|T], Term) :-
        term(H, Term);
        term(T, Term).

% free_in(VarList, Restriction)
% =============================
% VarList       ==> a list of variables which should
%                   be free in Restriction
% Restriction   ==> a wff or term
%
% Succeeds if each variable in VarList is free
% in Restriction

free_in([], R) :- !,
        fail.
free_in([H], R) :- !,
        free_in(H, R).
free_in([H|T], R) :- !,
        free_in(H, R),
        free_in(T, R).

% Var is a single variable. If Restriction is a wff,
% Var is free in it if it is free in its argument list.
free_in(Var, wff(Pred, ArgList)) :-
        free_in(Var, ArgList).

% Var is a single variable. If Restriction is an argument
% list, Var is free in it if it is free in the head or
% the tail of that argument list.
free_in(Var, [H|T]) :-
        free_in(Var, H);
        free_in(Var, T).

% Var is a single variable. If Restriction is a complex
% term, Var is free in % it if Var is not the variable
```

```
% quantified over and if Var is free in the restriction
% of the complex term.
free_in(Var, term(Quant, Var1, Restriction)) :-
        not Var = Var1,
        free_in(Var, Restriction).

% Var is a single variable. If Restriction is just this
% variable, then Var is free in Restriction
free_in(Var, Var).
```

5.6 Programs: hs-two.pl

```
% gen(Form, ScopedForm)
% =====================
% Form        ==> a wff with in-place complex terms
% ScopedForm  <== a full scoping of Form

gen(Form, ScopedForm) :-
        apply_terms(Form, true, ScopedForm).

% apply_terms(From, Complete, ScopedForm)
% ======================================
% Form       ==> a wff with in-place complex terms
% Complete   ==> true iff only full scopings are allowed
% ScopedForm <== a full or partial scoping of Form
%
% Applies one or more terms to the Form alone (not to any
% embedded forms).

apply_terms(Form, Complete, Form) :-
        not(term(Form, Term)), !.

apply_terms(Form, false, Form).

apply_terms(Form, Complete, ScopedForm) :-
        applicable_term(Form, Term),
        apply(Term, Form, AppliedForm),
        apply_terms(AppliedForm, Complete, ScopedForm).

% apply(Term, Form, NewForm)
% ==========================
% Term    ==> a complex term
% Form    ==> the wff to apply Term to
% NewForm <== Form with the quantifier wrapped around it

apply(term(Quant, Var, Restrict),
      Body,
      wff(Quant, [Var, PulledRestrict, OutBody])) :-
        apply_terms(Restrict, false, PulledRestrict),
        subst(Var, term(Quant, Var, Restrict), Body, OutBody).

% applicable_term(Form, Term)
```

```
% ============================
% Form ==> an expression in the logical form language
% Term <== a top-level term in Form (that is, a term
%          embedded in no other term) which is not free
%          in any variable bound along the path from Form
%          to Term.

applicable_term(Form, Term) :-
        applicable_term(Form, Term, []).


% applicable_term(Form, Term, BlockingVars)
% ========================================
% Form ==> an expression in the logical form language
% Term <== a top-level term in Form (that is, a term
%          embedded in no other term) which is not free
%          in any variable bound along the path from
%          Form to Term.
% BlockingVars ==> a list of variables bound along the
%                  path so far

% A term is an applicable top-level term...
applicable_term(term(Q, V, R), term(Q, V, R), BVs) :-
        % if it meets the definition.
        not(free_in(BVs, R)).

% An applicable term of the restriction or body of a
% quantifier is applicable only if the variable bound
% by the quantifier is not free in the term.
applicable_term(wff(Quant, [Var, Restrict, Body]),
                Term, BVs) :-
        quantifier(Quant), !,
        (applicable_term(Restrict, Term, [Var|BVs]);
         applicable_term(Body, Term, [Var|BVs])).

% An applicable term of any argument is an applicable
% term of the wff.
applicable_term(wff(Pred, Args), Term, BVs) :-
        applicable_term(Args, Term, BVs).

% An applicable term of any argument is an applicable
% term of the whole list.
applicable_term([F|R], Term, BVs) :-
        applicable_term(F, Term, BVs);
        applicable_term(R, Term, BVs).

% Note the absence of a rule looking for applicable
% terms inside of complex terms. This limits the applicable
% terms to be top-level.


/*****************************************************
Low-level predicates as in hs-one.pl
*****************************************************/
```

5.7 Programs:qscope.pl

```
% gen(Form, ScopedForm)
% ======================
% Form        ==> a wff with in-place complex terms
% ScopedForm  <== a full scoping of Form

gen(Form, ScopedForm) :-
        pull(Form, true, ScopedForm).

% pull(Form, Complete, ScopedForm)
% ================================
% Form       ==> a wff with in-place complex terms
% Complete   ==> true iff  only full scopings are allowed
% ScopedForm <== a full or partial scoping of Form
%
% Applies terms at various level of embedding in Form,
% including applying to the entire Form, and to opaque
% argument positions inside Form.

pull(Form, Complete, ScopedForm) :-
        pull_opaque_args(Form, PulledOpaque),
        apply_terms(PulledOpaque, Complete, ScopedForm).

% pull_opaque_args(Form, ScopedForm)
% ==================================
% Form       ==> a term or a wff with in-place
%                complex terms
% ScopedForm <== Form with opaque argument positions
%                recursively scoped
%
% Scopes arguments of the given Form recursively.

pull_opaque_args(wff(Pred, Args),
                 wff(Pred, ScopedArgs)) :- !,
        pull_opaque_args(Pred, 1, Args, ScopedArgs).

pull_opaque_args(Term, Term).

% pull_opaque_args(Pred, ArgIndex, Args, ScopedArgs)
% ==================================================
% Pred       ==> the predicate of the wff whose
%                arguments are being scoped
% ArgIndex   ==> the index of the argument currently
%                being scoped
% Args       ==> list of args from ArgIndex on
% ScopedArgs <== Args with opaque argument positions
%                recursively scoped
%
% Scopes a given argument if opaque; otherwise, scopes
% its subparts recursively.
```

```
% No more arguments.
pull_opaque_args(Pred, ArgIndex, [], []) :- !.

% Current argument position is opaque; scope it.
pull_opaque_args(Pred, ArgIndex,
                 [FirstArg|RestArgs],
                 [ScopedFirstArg|ScopedRestArgs]) :-
        opaque(Pred, ArgIndex),
        pull(FirstArg, false, ScopedFirstArg),
        NextIndex is ArgIndex + 1,
        pull_opaque_args(Pred, NextIndex, RestArgs,
                         ScopedRestArgs).

% Current argument position is not opaque; don't scope it.
pull_opaque_args(Pred, ArgIndex,
                 [FirstArg|RestArgs],
                 [ScopedFirstArg|ScopedRestArgs]) :-
        pull_opaque_args(FirstArg, ScopedFirstArg),
        NextIndex is ArgIndex + 1,
        pull_opaque_args(Pred, NextIndex, RestArgs,
                         ScopedRestArgs).

% apply_terms(From, Complete, ScopedForm)
% =======================================
% Form       ==> a wff with in-place complex terms
% Complete   ==> true iff only full scopings are allowed
% ScopedForm <== a full or partial scoping of Form
%
% Applies one or more terms to the Form alone (not to any
% embedded forms).

apply_terms(Form, Complete, Form) :-
        not(term(Form, Term)), !.

apply_terms(Form, false, Form).

apply_terms(Form, Complete, ScopedForm) :-
        applicable_term(Form, Term),
        apply(Term, Form, AppliedForm),
        apply_terms(AppliedForm, Complete, ScopedForm).

% apply(Term, Form, NewForm)
% ==========================
% Term    ==> a complex term
% Form    ==> the wff to apply Term to
% NewForm <== Form with the quantifier wrapped around it

apply(term(Quant, Var, Restrict),
                Body,
                wff(Quant, [Var, PulledRestrict, OutBody])) :-
        pull(Restrict, false, PulledRestrict),
        subst(Var, term(Quant, Var, Restrict), Body, OutBody).
/************************************************************
```

```
   applicable_term(Form,Term) & all Low level predicates
   as in hs-two.pl
**************************************************************/
```

Chapter 6

Questions and database query

6.1 Gap threading and semantic representation of questions and relative clauses

This discussion relates to the program **questions.pl**. Following Pereira and Shieber we will use a λ-term of arity one as the representation of a wh-question. Thus we will use `X^write(terry,X)` as the logical form for 'What did Terry write?'. The idea is that the answer would be the set of things to which this term truthfully applies. For yes-no questions we will use a mock λ-term with a λ-prefix "`yes^`", e.g. `yes^write(terry,shrdlu)`. The idea here is that the answer is "yes" if the formula without the prefix is satisfied and "no" otherwise.

Wh-questions such as "what did Terry write?", "what did a student say that Terry wrote?" (and also relative clauses) introduce long-distance (unbounded) dependencies between the wh-phrase and a gap. This program uses the method of gap-threading to treat them. Gaps are introduced by the rules

$$q(X\hat{}S) \; --> \; \text{whpron}, \; \text{sinv}(S, [\text{gap}(np, X)] - []).$$
$$\text{optrel}(P\hat{}(X\hat{}(P * X\&S))) \; --> \; \text{relpron}, \; \text{s}(S, [\text{gap}(np, X)|\text{Out}] - \text{Out}).$$

and resolved by the rule

$$np(P\hat{}(P * X), [\text{gap}(np, X)|\text{Out}] - \text{Out}) \; --> \; [].$$

the gap terms such as `gap(np,X)` encode both syntactic information about the category of the gap and semantic information indicating which variable will occur in the λ-prefix in the logical form of the question. We can think of the pairs of lists of gap terms as being associated with the constituents being parsed by the rule of which the pair of lists is an argument. A pair of lists `In-Out` indicate that the gaps listed in `In` are being sought within the constituent and that the gaps listed in `Out` are being sought after the constituent has been parsed. Thus for example if the pair `[gap(np,X)]-[]` is associated with a constituent then an NP-gap associated with the variable `X` in the logical form must have been found within the constituent. If the in- and out-lists are identical then no gap can have been found in the constituent. The program implements a couple of syntactic constraints. No extraction is allowed from subject noun-phrases and no extraction is allowed from relative clauses (the complex NP constraint). These are implemented by requiring the in- and out-lists to be identical for subject noun-phrases and for noun-phrases containing relative clauses. Gap information is not passed down into relative clauses.

Exercise 6.1.1

For what kind of wh-questions may a logical form which is a λ-term of arity one be inadequate? How could you extend the program to accommodate them?

Exercise 6.1.2

Add to the program questions.pl so that it will treat questions with "which", such as "which programs did Terry write?"

Exercise 6.1.3

Add to the program so that it will treat questions where the wh-phrase is a PP, e.g. "where did Terry work?", "in which room did Terry work?"

6.2 Answering questions

This concerns the program **answer.pl.** Giving the semantics for questions here involves specifying what the answer to the questions would be with respect to a given situation. We define a predicate `answer(Sit,X^For,Ans)`. In the case where `X` is a variable (corresponding to a wh- question in English) `Ans` will be instantiated to a list. This will be the list of things in the domain of `Sit` which, when substituted for `X` in `For`, will make it be the case that `Sit` satisfies `For`. The program makes explicit reference to the domain of `Sit` requiring it to be computed and requiring that any instantiation of `X` which allows `For` to be satisfied be a member of the domain. This is redundant in the case of positive questions. However, it is important in the case of negative questions like

```
X^ run(X)
```

so that we obtain some reasonable answer.

The answers for yes/no-questions (where `X` is instantiated to yes) are straightforward. The answer is "yes" if `For` is satisfied and "no" otherwise.

Exercise 6.2.1

Suppose we wanted to treat λ-terms of arbitrary arity as questions, i.e. not only `X^For` but also `X^Y^For, X^Y^Z^For` etc. How could the program be modified to find suitable answers for these? Are there any English questions that might correspond to these "λ-questions"?

Exercise 6.2.2

How appropriate is the treatment of negative questions here? Would we still have to compute the domain of the situation if we allowed negative formulae in our situation databases?

6.3 Questions in CHAT-80

This relates to the files in the directory **chat**. CHAT-80 is a program that was written in Edinburgh by Fernando Pereira and David Warren around 1980. The natural language part of the program was described in Pereira's PhD thesis written in the AI department. It still serves as a paradigm example of the logic programming approach to NLP because of its large grammatical coverage, its impressive efficiency and the clarity with which it is coded. In particular it serves as a paradigm example of an application of the kind of approach to computational semantics we have been pursuing here. While the semantics in the system is in some ways less theoretically sophisticated and general than what we have been studying here, there are a number of features which make it more efficient.

To load chat you will need to consult the file **load.pl** in the chat directory. Type

```
?- hi.
```

to get the program started. It will now accept and answer questions concerning a geographical database. You will find examples of questions it will handle in the file **demox**. To leave the program type

```
Question:  bye.
```

To see something of what it is doing give the command:

```
Question:  trace.
```

There are four main components:

1. parsing the sentence, producing a parse tree
2. translating the parse tree to a logical form
3. planning, rearranging the logical form to make it more efficient to query the database with
4. evaluation of the logical form in the database, i.e. answering the question.

Two important things that it does not do are

1. generate an answer to the question in English
2. handle any kind of discourse phenomena, relating a question to a previous question.

The method of parsing long distance dependencies uses the gap-threading technique, though in a somewhat different form than we have presented it. Pereira's thesis is the source of this technique. We are most interested here in the fourth component, evaluation and the relevant code is to be found in the file **talkr.pl**

The logical form produced for questions seems more procedural than what we have looked at since it appears that the logical form is a prolog procedure for answering the question. This difference is, however, only apparent since the logical form is not called as a prolog procedure but rather interpreted by a predicate answer/1 quite similar to the predicate answer/3 that we have defined. There is a predicate satisfy/1 quite similar to the related predicates we have been working with. One major difference is that there is only one database so that there is no need for the argument Sit which we have been using. Apart from the definite article *the*, about which I will not say anything here, the natural language determiners treated are *a*, *some*, *no*, *every*, *all*, *any*, *each* and the numerals. Existential quantification (represented by `X^For` - don't confuse this with λ-abstraction but think of the notation used for existential quantification in bagof etc.) is treated in the standard prolog way:

```
satisfy(X^P) :- !, satisfy(P).
```

i.e. no real quantification at all. This accounts for *a* and *some*. The determiner *no* is accounted for by the prolog negation of this. The treatment of the universal determiners *every*, *all* , *any* and *each* exploits the logical equivalence between $\forall$ and $\sim \exists \sim$. Thus at the level of logical form these determiners will result in something of the form

```
\+ exists ... \+ ...
```

Hence the evaluation of all of these quantifiers boils down to prolog pattern matching.

An exception to this treatment is cases where *each* has scope over a whole question. The general approach to scope phenomena is a rather pragmatic one. Only one scoping is produced. In general the order of the quantifiers in the logical form is the same as the corresponding determiners in the English sentence. The exception to this is that *each* and *any* are marked as strong and will take scope over other quantifiers, except that they do not take scope outside a relative clause. When *each* takes scope over a whole question this is a special case. Consider: 'which country borders each European country?' The reading taken for this could be paraphrased as: for each European country answer which country borders it. The answer is a list of pairs consisting of a European country and a country which borders it. The logical form is

```
answer([X,Y]) :-
        country(X),
        & european(X),
        & country(Y),
        & borders(Y,X).
```

This question is to be compared with "which country borders every European country?" Here the answer sought is a single country which borders all European countries, i.e. none.

Exercise 6.3.1

Find examples handled by chat that illustrates the fact that each and any take widest scope but do not take scope beyond relative clauses. Explain what is going on and discuss whether the readings are intuitively correct? Are there other possible readings for these sentences? Is the reading assigned by chat the most preferred reading (according to your intuitions)?

6.4 Programs:questions.pl

```
/* This is a version of Program 4.5 on p. 124 of Pereira
   and    Shieber. It differs from the original program
   in that it employs the method of gap threading discussed
   in the following section (4.2.7). It also uses the
   lambda calculus in the manner of eng-lambda.pl.  I
   have also added some extra rules and fixed a bug.
   rhc */

/* Needs beta.pl and logic-lexicon.pl */

% Operators for connectives

:- op(500,xfy,[&,\/]).
:- op(550,xfy,--->).
:- op(450,fx,~).

% Operator for functional application
:- op(400, yfx, *).

q(VP) --> whpron, vp(VP, []-[]).
q(X^S) --> whpron, sinv(S, [gap(np,X)]-[]).
q(yes^S) --> sinv(S,[]-[]).

s(S) --> s(S,[]-[]).
s(NP*VP,In-Out) --> np(NP, In-In), vp(VP, In-Out).

sinv(NP*VP,In-Out) -->
        aux, np(NP, In-In), vp(VP, In-Out).

sbar(S,In-Out) --> [that], s(S,In-Out).

np(Det*(Rel*N), In-In) -->
        det(Det), n(N), optrel(Rel).
np(P^(P*X), In-In) --> pn(X).
np(P^(P*X), [gap(np,X)|Out]-Out) --> [].
```

```
vp(X^(NP*(Y^(TV*Y*X))), In-Out) -->
        tv(TV), np(NP, In-Out).
vp(VP, In-In) --> iv(VP).
vp(STV*Sbar, In-Out) -->
        stv(STV), sbar(Sbar,In-Out).

optrel(N^N) --> [].
optrel(P^(X^(P*X&VP*X))) -->
        relpron, vp(VP, []-[]).
optrel(P^(X^(P*X&S))) -->
        relpron, s(S, [gap(np, X)|Out]-Out).

det(LF) --> [D], {det(D,LF)}.
det(every, Q^P^every(X,Q*X ---> P*X) ).
det(a, Q^P^some(X,Q*X & P*X) ).

n(LF) --> [N], {n(N,LF)}.
n(program, X^program(X) ).
n(student, X^student(X) ).

pn(E) --> [PN], {pn(PN, E)}.
pn(terry, terry).
pn(shrdlu, shrdlu).

tv(LF) --> [TV], {tv(TV,LF)}.
tv(wrote, Y^X^write(X,Y) ).
tv(write, Y^X^write(X,Y) ).
tv(writing, Y^X^write(X,Y) ).

stv(LF) --> [STV], {stv(STV,LF)}.
stv(said, Y^X^say(X,Y) ).
stv(say, Y^X^say(X,Y) ).
stv(saying, Y^X^say(X,Y) ).

iv(LF) --> [IV], {iv(IV,LF)}.
iv(halts, X^halt(X) ).

relpron --> [RelPron], {relpron(RelPron)}.
relpron(that).
relpron(who).
relpron(whom).

whpron --> [WhPron], {whpron(WhPron)}.
whpron(what).
whpron(who).
whpron(whom).

aux --> [Aux], {aux(Aux)}.
aux(is).
aux(did).

parse(S,LF) :-
        (q(L,S,[]);
```

```
        s(L,S,[])),
        convert_question(L,LF).

convert_question(X^P,X^Q) :-
        !,convert(P,Q).
convert_question(P,Q) :-
        convert(P,Q).
```

6.5 Programs:answer.pl

```
/* Requires satisfy-pc.pl */

answer(Sit,X^For,Ans) :-
        var(X),!,
        domain(Sit,Dom),
        setof(X,(member(X,Dom),satisfy(Sit,For)),Ans).
answer(Sit,yes^For,yes) :-
        satisfy(Sit,For),!.
answer(Sit,yes^For,no).
```

Chapter 7

Discourse anaphora

7.1 Introduction

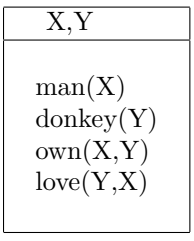
In this chapter we move from the discussion of the semantics of sentences to the discussion of simple discourses, that is, the semantic processing of several sentences one following after the other. We will focus on the phenomenon of anaphora, that is, when pronouns are linked to previous noun-phrases, either within the same sentence or elsewhere in the preceding discourse. Here are some of the kinds of examples we want to handle. The subscripts on the noun-phrases indicate what the antecedents of the pronouns are. The stars indicate that the discourse cannot be read with the anaphora resolution as indicated.

John$_1$ owns a donkey$_2$. It$_2$ loves him$_1$.
A man$_1$ owns a donkey$_2$. It$_2$ loves him$_1$.
A man$_1$ owns a donkey that loves him$_1$.
Every man$_1$ owns a donkey$_2$. *It$_2$ loves him$_1$.
Every man$_1$ owns a donkey$_2$. *Mary loves him$_1$.
Every man$_1$ owns a donkey$_2$. *It$_2$ loves Mary.
Every man$_1$ owns a donkey that loves him$_1$.
Every man that owns a donkey$_1$ beats it$_1$.

We will use the framework of discourse representation theory (DRT) developed by Hans Kamp to treat these phenomena. The implementation of it we will discuss is based on the ideas developed in the implementation of DRT by Mark Johnson and Ewan Klein. The basic idea is that we will translate English discourses into discourse representation structures (DRSs) rather than logical form and we will define satisfaction for DRSs.

7.2 Simple discourse representation structures

This discussion relates to the program `satisfy-simple-drs.pl` At an abstract level simple DRSs are like our situations. A DRS consists of two components: a domain and a set of constraints on that domain. Here is an example in diagrammatic notation:



This differs from our situations in two ways. Firstly the domain is made explicit. Secondly, the domain is parametric, i.e. it consists of objects which are to be matched against the domain of a situation when we do semantics. In the theory these are often called discourse referents or discourse markers. We shall use prolog variables to implement them. As with logical variables this is a design choice. The Johnson and Klein implementation uses prolog atoms for discourse referents. We shall implement DRS with a pair of lists:

```
[[X,Y],[man(X),donkey(Y),own(X,Y),love(Y,X)]]
```

The semantics for simple DRSs is quite straightforward. A situation satisfies a DRS if there is some binding for the discourse referents such that all the constraints are supported by the situation under that binding. Recall that `support/2` is the predicate which checks that a basic fact is included in a situation. Prolog looks after finding the binding for you so that does not have to be implemented explicitly.

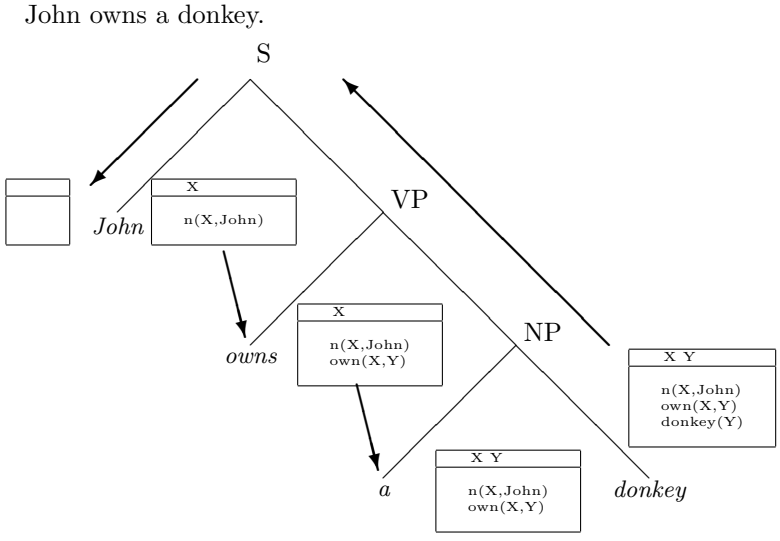
Exercise 7.2.1

It is possible for a situation to satisfy a DRS more than one way, giving different instantiations for the variables in the DRS. Give examples and explain what is going on.

7.3 Translating simple discourses

This discussion relates to the program `eng-simple-drs.pl` .

The technique that we will use for deriving DRSs for English sentences is a similar kind of threading to that we used for gaps in the previous chapter. (We have taken the idea of using threading from Johnson and Klein.) With each English constituent we associate an ingoing and an outgoing discourse representation. Here is a diagrammatic representation of the flow of discourse information in the sentence.



The output DRS of this sentence would then be the input DRS for a following sentence in the discourse. Note that there has been a change in how we treat proper names. We are no longer thinking of them as representing constants. Rather they function to introduce new discourse referents and add constraints about the name and the gender of the thing associated with the discourse referent. If a discourse referent with these constraints already exists in the DRS (e.g. if we have used the name 'John' twice in the same discourse) a new discourse referent will not be added.

The indefinite NP functions in the same way as a proper name except that it will always introduce a new discourse referent. If we use the NP 'a donkey' twice. There will be two discourse referents for donkeys in the DRS. It is an important aspect of the discourse representation approach that the indefinite NP does not introduce a representation of existential quantification into the DRS. The effect of existential quantification comes in the semantics for the DRS where you try to find some way of assigning discourse referents to individuals in the situation that satisfies the DRS.

Pronouns on the other hand never introduce new discourse referents. They will always require that there is a discourse referent previously introduced into the DRS which has the appropriate gender constraints.

In addition to assigning input and output DRSs to constituents this program also assigns something like a logical form. This is because we need to keep track of the relationship between constituents and various parts of constraints. For example, the two arguments in the constraint introduced by a transitive verb must be associated with the appropriate NPs.

Exercise 7.3.1

Explain how this program treats the discourses:

John$_1$ owns a donkey$_2$. It$_2$ loves him$_1$.
A man$_1$ owns a donkey$_2$. It$_2$ loves him$_1$.
A man$_1$ owns a donkey that loves him$_1$.

Indicate what the input and output DRSs are at important points and how the anaphora resolution is determined.

Exercise 7.3.2

Discuss the treatment of gender here. Are there cases in English where it would be problematic? Are there cases in other languages which would cause problems? Do you have any ideas how it might be done better?

Exercise 7.3.3

Is the sentence

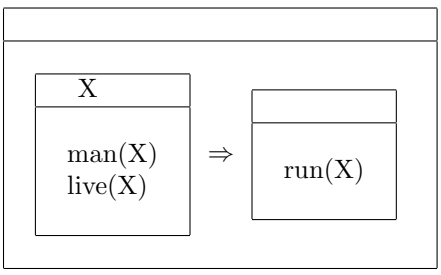
John likes him

treated correctly here? Explain.

7.4 Complex discourse representation structures

This discussion relates to the program `satisfy-complex-drs.pl` .

It is possible to introduce all of the connectives (negation, conjunction, disjunction and implication) and generalized quantifiers into DRSs. Here we will be concerned only with implication and universal quantification which are introduced by the same addition to DRSs. We allow DRSs of the form:



This discourse represent corresponds to the two English sentences

every man who lives runs
if a man lives he runs

Here the constraint consists of two embedded DRSs joined by an arrow. The effect of universal quantification comes in the interpretation of this constraint. In order for a situation to support such a constraint it must be the case that for each of the bindings of discourse referents which make the situation satisfy the left-hand DRS, the right-hand DRS is also satisfied on that binding. Thus here the semantics for the DRS requires universal quantification. Note, however, that again the quantification is not represented directly within the DRS where the constraint corresponding to 'man' occurs. Thus it can be possible for an indefinite NP within an *if*-clause to have the effect of a universal quantifier.

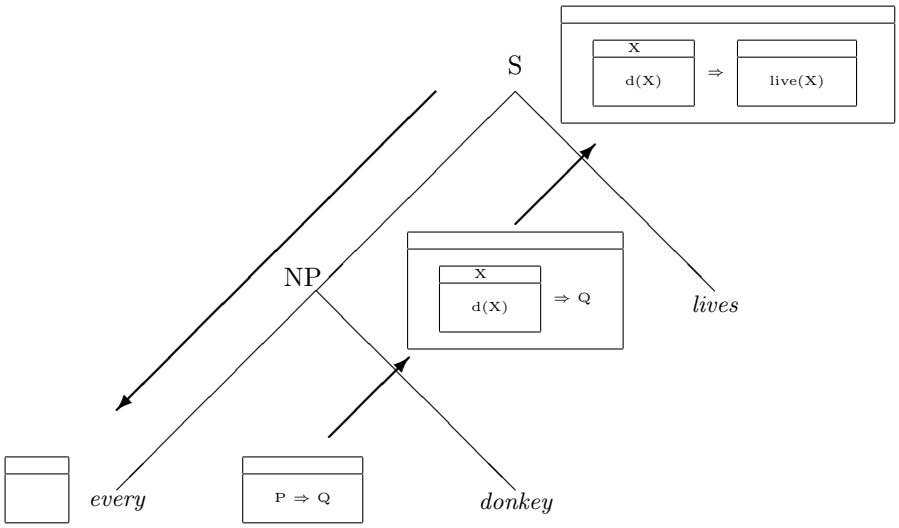
Exercise 7.4.1

Suppose we were to introduce a different arrow into our discourse representation to represent the quantifier 'two'. How might you change `satisfy-complex-drs.pl` to accommodate it? How about 'exactly n'?

7.5 Translating discourses with quantified sentences

This discussion relates to the programs `eng-complex-drs.pl` and `eng-global-drs.pl` .

In the program `eng-complex-drs.pl` a major distinction is made between proper names and indefinite NPs on the one hand and quantified NPs (using the category qnp) on the other. It is possible to have a single category of NP in the syntax but this would make the program somewhat harder to understand. The only addition to this program over the program `eng-simple-drs.pl` is that of universally quantified NPs. It is the determiner 'every' which requires the introduction of the complex constraint as described in the previous section. Here is a diagrammatic representation of the flow of information for the sentence 'every donkey lives'.



In this implementation information from the antecedent DRS is copied into the consequent DRS. It is important to do this to make sure that the information is accessible to pronouns. The general

case is that any discourse referent which is in an enclosing DRS is available for anaphora resolution to an pronoun being processed in relation to a DRS within it. In addition any information is accessible in an antecedent DRS is accessible to the consequent DRS. This implementation fails in an important way in that it treats proper names just like indefinites, adding them to the current DRS rather than the largest (global) DRS. Thus according to `eng-complex-drs.pl` both of the following discourses will not parse:

Every donkey likes a man. He lives.
Every donkey likes John. He lives.

Since we are only dealing with one quantifier scope here, where there could be a different man for each donkey, the first discourse is correctly rejected. But the second is incorrectly rejected since the discourse referent for John is added to an embedded consequent DRS and thus is not accessible from the global DRS to which the constraint about living is to be added. This is corrected in the program `eng-global-drs.pl` . In this program proper names are distinguished from indefinite NPs in that the former will add a discourse referent to the global DRS and the latter will add a discourse referent to the local DRS. Pronoun resolution checks first the local DRS for an antecedent and then the global DRS. Both of these programs successfully treat the so-called donkey sentences:

Every man that owns a donkey$_1$ beats it$_1$.

The problem with such sentences is that, given the assumptions that 'every' corresponds to a universal quantifier and 'a' corresponds to an existential there is no way to get a predicate calculus representation of the apparent reading where the universal quantifier has wider scope than the existential and the pronoun gets bound by the existential. In the discourse representation framework such sentences are successfully treated because the discourse referent introduced by 'a donkey' is in effect universally quantified over since it occurs in the antecedent DRS introduced by 'every'.

Exercise 7.5.1

Explain how the following discourses are treated by the program `eng-complex-drs.pl` .

Every man$_1$ owns a donkey$_2$. *It$_2$ loves him$_1$.
Every man$_1$ owns a donkey$_2$. *Mary loves him$_1$.
Every man$_1$ owns a donkey$_2$. *It$_2$ loves Mary.
Every man$_1$ owns a donkey that loves him$_1$.
Every man that owns a donkey$_1$ beats it$_1$.

Give enough detail to show how the threading of DRS information allows the crucial differences to be made.

Exercise 7.5.2

Explain how the treatment of global DRSs is implemented in `eng-global-drs.pl` discussing the examples

*Every donkey likes a man. He lives. Every donkey likes John. He lives.

7.6 Programs

These are programs for chapter 6.

7.6.1 satisfy-simple-drs.pl

```
satisfy(Sit,[Dom,Constr]) :-
        support_all(Sit,Constr).

support_all(_,[]).
support_all(Sit,[X|L]) :-
        support(Sit,X),
        support_all(Sit,L).

support(Sit,Fact) :-
        Clause =.. [Sit,Fact],
        Clause.
```

7.6.2 eng-simple-drs.pl

```
/*  A simple implementation of discourse representation
    using simple drs's */

/*  You may need to define member and append, depending
    on the prolog you are using */

s(DRSIn,DRSOut) --> np(DRSIn,DRS1,X),
                    vp(DRS1,DRSOut,X^Cond).

np(DRSIn,DRSOut,X) -->
        det(DRSIn,DRS1),
        noun(DRS1,DRS2,X^Cond),
        optrel(DRS2,DRSOut,X^Cond1).

np(DRSIn,DRSOut,X) -->
        proper_noun(DRSIn,DRSOut,X).

np(DRSIn,DRSOut,X) -->
        pronoun(DRSIn,DRSOut,X).

optrel(DRS,DRS,X^_) --> [].
optrel(DRSIn,DRSOut,X^Cond) -->
        [that],vp(DRSIn,DRSOut,X^Cond).

vp(DRSIn,DRSOut,X^Cond) -->
        trans_verb(DRSIn,DRS1,Y^X^Cond),
        np(DRS1,DRSOut,Y).
vp(DRSIn,DRSOut,X^Cond) -->
        intrans_verb(DRSIn,DRSOut,X^Cond).

det(DRS,DRS) -->
        [a].

noun([Dom,Constr],[[X|Dom],
```

```
     [man(X),male(X)|Constr]],X^man(X)) -->
        [man].
noun([Dom,Constr],[[X|Dom],
     [woman(X),female(X)|Constr]],X^woman(X)) -->
        [woman].
noun([Dom,Constr],[[X|Dom],
     [donkey(X),neuter(X)|Constr]],X^donkey(X)) -->
        [donkey].

proper_noun(DRSIn,DRSOut,X) -->
        [john],{check_add(X,[named(X,john),male(X)],
                          DRSIn,DRSOut)}.
proper_noun(DRSIn,DRSOut,X) -->
        [mary],{check_add(X,[named(X,mary),female(X)],
                          DRSIn,DRSOut)}.


pronoun(DRS,DRS,X) -->
        [he], {check(X,[male(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [she], {check(X,[female(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [it], {check(X,[neuter(X)],DRS)}.

trans_verb([Dom,Constr],
           [Dom,[love(X,Y)|Constr]],
           Y^X^love(X,Y)) -->
        [loves].
trans_verb([Dom,Constr],
           [Dom,[beat(X,Y)|Constr]],
           Y^X^beat(X,Y)) -->
        [beats].
trans_verb([Dom,Constr],
           [Dom,[own(X,Y)|Constr]],
           Y^X^own(X,Y)) -->
        [owns].

intrans_verb([Dom,Constr],
             [Dom,[live(X)|Constr]],
             X^live(X)) -->
        [lives].


drs_sent(Sent,DRSIn,DRSOut) :-
        s(DRSIn,DRSOut,Sent,[]).

drs([],DRS,DRS).
drs([Sent|Disc],DRSIn,DRSOut) :-
        drs_sent(Sent,DRSIn,DRS1),
        drs(Disc,DRS1,DRSOut).

drs(Disc,DRS) :-
        drs(Disc,[[],[]],DRS).
```

```
check(X,Cs,[Dom,Constr]) :-
        member(X,Dom),
        strict_members(Cs,Constr).

check_add(X,Cs,DRS,DRS) :-
        check(X,Cs,DRS),!.
check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).

strict_member(X,[Y|L]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).

strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).
```

7.6.3 satisfy-complex-drs.pl

```
:- op(150,xfy,=>).

satisfy(Sit,[Dom,Constr]) :-
        support_all(Sit,Constr).

support_all(_,[]).
support_all(Sit,[X|L]) :-
        support(Sit,X),
        support_all(Sit,L).

support(Sit, [Dom,Constr]=>DRS) :-
        !,bagof(Dom,support_all(Sit,Constr),Bindings),
        satisfy_all(Sit,Bindings,Dom,DRS).
support(Sit,Fact) :-
        Clause =.. [Sit,Fact],
        Clause.

satisfy_all(_,[],_,_).
satisfy_all(Sit,[B|Bindings],Dom,DRS) :-
        \+ \+ (B=Dom,satisfy(Sit,DRS)),
        satisfy_all(Sit,Bindings,Dom,DRS).
```

7.6.4 eng-complex-drs.pl

```
/*  A simple implementation of discourse representation
    using complex drs's for universal quantification*/

/*  You may need to define member and append, depending
    on the prolog you are using */

:- op(150,xfy,=>).
```

```
s(DRSIn,DRSOut) --> np(DRSIn,DRS1,X),
                    vp(DRS1,DRSOut,X^Cond).
s(DRSIn,DRSOut) --> qnp(DRSIn,DRSOut,X^P^(Q=>P)),
                    vp(Q,P,X^Cond).

np(DRSIn,DRSOut,X) -->
        det(DRSIn,DRS1),
        noun(DRS1,DRS2,X^Cond),
        optrel(DRS2,DRSOut,X^Cond1).

qnp(DRSIn,DRSOut,X^P^DRS) -->
        det(DRSIn,DRSOut,Q^P^DRS),
        noun(DRSIn,DRS1,X^Cond),
        optrel(DRS1,Q,X^Cond1).

np(DRSIn,DRSOut,X) -->
        proper_noun(DRSIn,DRSOut,X).

np(DRSIn,DRSOut,X) -->
        pronoun(DRSIn,DRSOut,X).

optrel(DRS,DRS,X^_) --> [].
optrel(DRSIn,DRSOut,X^Cond) -->
        [that],vp(DRSIn,DRSOut,X^Cond).

vp(DRSIn,DRSOut,X^Cond) -->
        trans_verb(DRSIn,DRS1,Y^X^Cond),
        np(DRS1,DRSOut,Y).
vp(DRSIn,DRSOut,X^Cond) -->
        trans_verb(Q,P,Y^X^Cond),
        qnp(DRSIn,DRSOut,Y^P^(Q=>P)).
vp(DRSIn,DRSOut,X^Cond) -->
        intrans_verb(DRSIn,DRSOut,X^Cond).

det(DRS,DRS) -->
        [a].

det([Dom,Constr],[Dom,[Q=>P|Constr]],Q^P^(Q=>P)) -->
        [every].

noun([Dom,Constr],[[X|Dom],
     [man(X),male(X)|Constr]],X^man(X)) -->
        [man].
noun([Dom,Constr],[[X|Dom],
      [woman(X),female(X)|Constr]],X^woman(X)) -->
        [woman].
noun([Dom,Constr],[[X|Dom],
     [donkey(X),neuter(X)|Constr]],X^donkey(X)) -->
        [donkey].

proper_noun(DRSIn,DRSOut,X) -->
        [john],{check_add(X,[named(X,john),male(X)],
                          DRSIn,DRSOut)}.
```

```
proper_noun(DRSIn,DRSOut,X) -->
        [mary],{check_add(X,[named(X,mary),female(X)],
                          DRSIn,DRSOut)}.


pronoun(DRS,DRS,X) -->
        [he], {check(X,[male(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [him], {check(X,[male(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [she], {check(X,[female(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [her], {check(X,[female(X)],DRS)}.
pronoun(DRS,DRS,X) -->
        [it], {check(X,[neuter(X)],DRS)}.

trans_verb([Dom,Constr],
           [Dom,[love(X,Y)|Constr]],
           Y^X^love(X,Y)) -->
        [loves].
trans_verb([Dom,Constr],
           [Dom,[beat(X,Y)|Constr]],
           Y^X^beat(X,Y)) -->
        [beats].
trans_verb([Dom,Constr],
           [Dom,[own(X,Y)|Constr]],
           Y^X^own(X,Y)) -->
        [owns].

intrans_verb([Dom,Constr],
             [Dom,[live(X)|Constr]],
             X^live(X)) -->
        [lives].


drs_sent(Sent,DRSIn,DRSOut) :-
        s(DRSIn,DRSOut,Sent,[]).

drs([],DRS,DRS).
drs([Sent|Disc],DRSIn,DRSOut) :-
        drs_sent(Sent,DRSIn,DRS1),
        drs(Disc,DRS1,DRSOut).

drs(Disc,DRS) :-
        drs(Disc,[[],[]],DRS).

check(X,Cs,[Dom,Constr]) :-
        member(X,Dom),
        strict_members(Cs,Constr).

check_add(X,Cs,DRS,DRS) :-
        check(X,Cs,DRS),!.
check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).
```

```
strict_member(X,[Y|L]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).

strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).
```

7.6.5 eng-global-drs.pl

```
/*  A simple implementation of discourse representation
    using complex drs's for universal quantification*/

/*  You may need to define member and append, depending
    on the prolog you are using */

:- op(150,xfy,=>).

s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut) -->
        np(Gl_DRSIn,DRSIn,DRS1,X,Gl_DRS1),
        vp(Gl_DRS1,DRS1,DRSOut,X^Cond,Gl_DRSOut).
s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut) -->
        qnp(Gl_DRSIn, DRSIn,DRSOut,X^P^(Q=>P),Gl_DRS1),
        vp(Gl_DRS1,Q,P,X^Cond,Gl_DRSOut).

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut) -->
        det(Gl_DRSIn,DRSIn,DRS1,Gl_DRS1),
        noun(Gl_DRS1,DRS1,DRS2,X^Cond,Gl_DRS2),
        optrel(Gl_DRS2,DRS2,DRSOut,X^Cond1,Gl_DRSOut).

qnp(Gl_DRSIn,DRSIn,DRSOut,X^P^DRS,Gl_DRSOut) -->
        det(Gl_DRSIn,DRSIn,DRSOut,Q^P^DRS,Gl_DRS1),
        noun(Gl_DRS1,DRSIn,DRS1,X^Cond,Gl_DRS2),
        optrel(Gl_DRS2,DRS1,Q,X^Cond1,Gl_DRSOut).

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut) -->
        proper_noun(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut).

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut) -->
        pronoun(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut).

optrel(Gl_DRS,DRS,DRS,X^_,Gl_DRS) --> [].
optrel(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut) -->
        [that],vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut).

vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut) -->
        trans_verb(Gl_DRSIn,DRSIn,DRS1,Y^X^Cond,Gl_DRS1),
        np(Gl_DRS1,DRS1,DRSOut,Y,Gl_DRSOut).
vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut) -->
        trans_verb(Gl_DRSIn,Q,P,Y^X^Cond,Gl_DRS1),
```

```
        qnp(Gl_DRS1,DRSIn,DRSOut,Y^P^(Q=>P),Gl_DRSOut).
vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut) -->
        intrans_verb(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut).

det(Gl_DRS,DRS,DRS,Gl_DRS) -->
        [a].

det(Gl_DRS,[Dom,Constr],
    [Dom,[Q=>P|Constr]],Q^P^(Q=>P),Gl_DRS) -->
        [every].

noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [man(X),male(X)|Constr]],X^man(X),Gl_DRS) -->
        [man].
noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [woman(X),female(X)|Constr]],X^woman(X),Gl_DRS) -->
        [woman].
noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [donkey(X),neuter(X)|Constr]],X^donkey(X),Gl_DRS) -->
        [donkey].

proper_noun(Gl_DRSIn,DRS,DRS,X,Gl_DRSOut) -->
        [john],{check_add(X,[named(X,john),male(X)],
                          Gl_DRSIn,Gl_DRSOut)}.
proper_noun(Gl_DRSIn,DRS,DRS,X,Gl_DRSOut) -->
        [mary],{check_add(X,[named(X,mary),female(X)],
                          Gl_DRSIn,Gl_DRSOut)}.


pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS) -->
        [he], {check(X,[male(X)],DRS);
                check(X,[male(X)],Gl_DRS)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS) -->
        [him], {check(X,[male(X)],DRS);
                  check(X,[male(X)],Gl_DRS)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS) -->
        [she], {check(X,[female(X)],DRS);
                  check(X,[female(X)],Gl_DRS)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS) -->
        [her], {check(X,[female(X)],DRS);
                  check(X,[female(X)],Gl_DRS)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS) -->
        [it], {check(X,[neuter(X)],DRS);
                check(X,[neuter(X)],Gl_DRS)}.

trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[love(X,Y)|Constr]],
           Y^X^love(X,Y),Gl_DRS) -->
        [loves].
trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[beat(X,Y)|Constr]],
           Y^X^beat(X,Y),Gl_DRS) -->
        [beats].
trans_verb(Gl_DRS,[Dom,Constr],
```

```
           [Dom,[own(X,Y)|Constr]],
           Y^X^own(X,Y),Gl_DRS) -->
        [owns].

intrans_verb(Gl_DRS,[Dom,Constr],
             [Dom,[live(X)|Constr]],
             X^live(X),Gl_DRS) -->
        [lives].


drs_sent(Sent,Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut) :-
        s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut,Sent,[]).

drs([],Gl_DRS,DRS,DRS,Gl_DRS).
drs([Sent|Disc],Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut) :-
        drs_sent(Sent,Gl_DRSIn,DRSIn,DRS1,Gl_DRS1),
        drs(Disc,Gl_DRS1,DRS1,DRSOut,Gl_DRSOut).

drs(Disc,DRS) :-
        drs(Disc,[[],[]],[[],[]],DRSOut,Gl_DRSOut),
        merge_drs(Gl_DRSOut,DRSOut,DRS).

merge_drs([Dom1,Constr1],[Dom2,Constr2],[Dom,Constr]) :-
        append(Dom1,Dom2,Dom),
        append(Constr1,Constr2,Constr).

check(X,Cs,[Dom,Constr]) :-
        member(X,Dom),
        strict_members(Cs,Constr).

check_add(X,Cs,DRS,DRS) :-
        check(X,Cs,DRS),!.
check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).

strict_member(X,[Y|L]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).

strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).
```

Chapter 8

Anaphora resolution algorithms and DRS threading

8.1 Introduction

In this chapter we will look at Hobbs' "naive" algorithm for pronoun resolution and see how we might encode it logically using the kind of DRS threading that we used in the previous chapter. We will then look at Hobbs' proposals for a semantic algorithm and evaluate them.

8.2 The Hobbs syntactic algorithm

This algorithm is in some ways more sophisticated than what we did in the previous chapter. In other ways it is less sophisticated. The algorithm tells you how to traverse a parse tree that has been previously built starting with a pronoun that you want to resolve and searching the tree to find an appropriate antecedent. If the pronoun does not get resolved to an antecedent in the tree in which the pronoun is located then trees for previous sentences in the discourse are searched. Thus the algorithm assumes a rather different kind of program to the one we developed. Firstly, it assumes that parse trees are being built for sentences in the discourse and are stored so that we may retrieve them. Secondly, it is an algorithm that searches back in the syntactic representation for possible antecedents after the parse is completed. It does not carry a list of possible antecedents around during the parse in the way that the DRS-threading programs do. Here is Hobbs' informal characterization of the algorithm.

> (1) Begin at the NP node immediately dominating the pronoun.
>
> (2) Go up the tree to the first NP or S node encountered. Call this node X, and call the path used to reach it p.
>
> (3) Traverse all branches below node X to the left of path p in a left-to-right, breadth-first fashion. Propose as the antecedent any NP node that is encountered which has an NP or S node between it and X.
>
> (4) If node X is the highest S node in the sentence, traverse the surface parse trees of previous sentences in the text in order of recency, the most recent first; each tree is traversed in a left-to-right, breadth-first manner, and when an NP node is encountered, it is proposed as antecedent. If X is not the highest S node in the sentence, continue to step 5.
>
> (5) From node X, go up the tree to the first NP or S node encountered. Call this new node X, and call the path traversed to reach it p.
>
> (6) If X is an NP node and if the path p to X did not pass through the Nbar node that X immediately dominates, propose X aa the antecedent.

(7) Traverse all branches below node X to the left of path p in a left-to-right, breadth-first manner. Propose any NP node encountered as the antecedent.

(8) If X is an S node, traverse all branches of node X to the right of path p in a left-to-right, breadth-first manner, but do not go below any NP or S node encountered. Propose any NP node encountered as the antecedent.

(9) Go to step 4.

This algorithm builds in precede and command conditions on antecedents for pronouns within sentences. Clause (3) will not allow

John$_1$ likes him$_1$

since an NP or S node does not intervene between the NP John and the first S dominating the pronoun *him*. However, it does allow

John's$_1$ mother likes him$_1$
That John$_1$ is always sick bothers him$_1$

Clause (4) will allow any NP in a preceding sentence to be an antecedent. This is less sophisticated than our previous treatment. Clause (6) is to distinguish between examples like

Mr Smith saw a driver$_1$ in his$_1$ truck

where according to Hobbs' analysis *in his truck* is a sister of the Nbar *driver* directly dominated by NP, and

Mr Smith$_1$ saw a driver of his$_1$ truck

which, according to Hobbs' analysis has an Nbar node dominating both the Nbar *driver* and the PP *of his truck*.

If the pronoun is in an embedded sentence, clause (7) will allow any NP to the left of the sentence to be an antecedent for the pronoun.

John$_1$ thinks that he$_1$ is sick
The fact that John$_1$ has spots makes me think that he$_1$ is sick

Clause (8) allows antecedents to the right of the pronoun. It allows

The fact that he$_1$ has spots makes John$_1$ feel uneasy

but disallows

The fact that he$_1$ has spots makes me think that John$_1$ is sick

It appears that Hobbs made an error here since this sentence should be allowed.

Exercise 8.2.1

Explain how Hobbs' algorithm would work in the following discourse:

A man that knows Mary loves her. He owns a donkey that loves her.

8.3 A declarative version of the algorithm

This discussion relates to the program `Hobbs-drs.pl` . This program is not an implementation of the complete Hobbs algorithm but shows in principle how it can be encoded in a logic program using threading. The program is designed to treat the following examples:

*John$_1$ loves him$_1$
John$_1$ loves a woman that loves him$_1$

The technique is to thread yet another argument through the syntactic parse. This time the argument is a list of discourse referents that are local to the NP or sentence being processed. The idea is that a pronoun can only be resolved to a discourse referent that is available in the same way as the previous programs and is not on the local list. Any discourse referent that is added to the DRS is also added to the local list. The ingoing local list to an S or NP is not passed down to the constituents of the S or NP since they are no longer local there. They are, however, passed along in the outlist of the S or NP as they are local to the clause in which the S or NP occurs. The discourse referent contributed by the head noun of a relative clause (e.g. *man* in *a man that likes Mary*) is local both to the relative clause and to the clause in which the NP *a man that likes Mary* occurs.

This program allows pronouns to be resolved during a single pass parse and does not require that we search a syntactic tree for antecedents after the parse has been completed.

Exercise 8.3.1

How could you modify this program to give a simple treatment of reflexive pronouns (like *himself*)?

Exercise 8.3.2

A treatment of the local lists with respect to NPs whereby discourse referents in the ingoing local list to an NP are not local for constituents within the NP will not quite work for the examples *a driver in his truck* and *a driver of his truck*. Explain what the problem is and suggest what the solution might be.

Exercise 8.3.3

Hobbs' algorithm expresses a preference for certain antecedents in that it finds some before others. How far does the order of discourse referents in the domain of the constructed DRS correspond to this?

Exercise 8.3.4

Explain how this program works on the following discourse:

> A man that knows Mary loves her. He owns a donkey that loves her.

8.4 Hobbs' semantic approach

I will not discuss this in detail. It shows how a heuristic approach involving world knowledge can be inovked to resolve pronouns that do not get the right antecedent immediately assigned by the "naive" algorithm. Another way to view this is as a filter which chooses among the several possibilities presented by the structural approach. While such heuristics are necessary and important in any practical system it is difficult to make any general sense of them that will give us principles that can be transported from one domain to another. Hobbs' approach represents an early attempt to find some kind of general patterns but it has its limitations. Let us consider one example:

> The FBI said they had tentative identifications on the fugitives, but didnt know where *they* were.

We are concerned with the problem of finding an antecedent for *they*. Hobbs' proposal rests on the fact that the conjunction *but* suggests contrast. He says we can make the following inferences from the first conjunct:

> the FBI had tentative identifications on the fugitives
> → the FBI had tentatively identified the fugitives
> → the FBI (tentatively) knows the names of the fugitives

This contrasts with

the FBI does not know the location of X

which can be inferred from the second conjunct. X therefore gets resolved to *the fugitives* because of the contrasting pattern.

This all seems quite intuitive and in some sense to correspond to the kind of reasoning that a human being might go through. On the other hand, it involves forward chaining reasoning on both conjuncts. One can imagine that in a very restricted system it might work but if there is any richness to the inferences we can draw it seems that it would be very difficult to know in advance which chains of inference from the two conjuncts would lead us to the appropriate contrast.

Hobbs suggests an alternative treatment of this example based on searching the lexicon for inferences we can draw about previous entities mentioned in the discourse to see if we can make it match what is asserted about the referent of the pronoun. In this example he suggests forward chaining on *fugitive*.

fugitive(X)
→ hide-from(X, police)
→ cause(X, not(know(police,location(X))))
→ not(know(police,location(X)))

This last is exactly what we know from the second conjunct about the referent of *they*. Therefore *they* should be resolved to *the fugitives*. This, like the previous solution, has the problem of doing forward chaining in a potentially large space of possible inferences. Furthermore we have to choose the appropriate word in the preceding discourse to do the inferencing on. But there is not just a search problem here. If we could really make this inference from *fugitive* then it would appear that the second conjunct of the sentence is redundant and it is difficult to see why we should use *but* to indicate contrast if what is expressed by the second conjunct follows from the first. Furthermore, this analysis of *fugitive* might have the unfortunate consequence that a sentence like

the police found the fugitives

is contradictory, because if the police found them they presumably know where they are and yet according to this being a fugitive means that the police do not know where you are.

8.5 Programs

Hobbs-drs.pl

```
/*  A simple implementation of discourse representation
    using complex drs's for universal quantification*/

/*  You may need to define member and append, depending
    on the prolog you are using */

:- op(150,xfy,=>).

s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut,
  LocList,LocList) -->
        np(Gl_DRSIn,DRSIn,DRS1,X,Gl_DRS1,
           [],LocList1),
        vp(Gl_DRS1,DRS1,DRSOut,X^Cond,
           Gl_DRSOut,LocList1,_).
s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut,
  LocList,LocList) -->
        qnp(Gl_DRSIn, DRSIn,DRSOut,X^P^(Q=>P),
            Gl_DRS1,[],LocList1),
        vp(Gl_DRS1,Q,P,X^Cond,Gl_DRSOut,LocList1,_).

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut,
   LocListIn,LocListOut) -->
        det(Gl_DRSIn,DRSIn,DRS1,Gl_DRS1),
        noun(Gl_DRS1,DRS1,DRS2,X^Cond,
             Gl_DRS2,[],LocList1),
        optrel(Gl_DRS2,DRS2,DRSOut,X^Cond1,
               Gl_DRSOut,LocList1,_),
        {append(LocListIn,LocList1,LocListOut)}.

qnp(Gl_DRSIn,DRSIn,DRSOut,X^P^DRS,Gl_DRSOut,
    LocListIn,LocListOut) -->
        det(Gl_DRSIn,DRSIn,DRSOut,Q^P^DRS,Gl_DRS1),
        noun(Gl_DRS1,DRSIn,DRS1,X^Cond,
             Gl_DRS2,[],LocList1),
        optrel(Gl_DRS2,DRS1,Q,X^Cond1,
               Gl_DRSOut,LocList1,_),
        {append(LocListIn,LocList1,LocListOut)}.

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut,
   LocListIn,LocListOut) -->
        proper_noun(Gl_DRSIn,DRSIn,DRSOut,X,
                    Gl_DRSOut,LocListIn,LocListOut).

np(Gl_DRSIn,DRSIn,DRSOut,X,Gl_DRSOut,
   LocListIn,LocListOut) -->
        pronoun(Gl_DRSIn,DRSIn,DRSOut,X,
                Gl_DRSOut,LocListIn,LocListOut).

optrel(Gl_DRS,DRS,DRS,X^_,Gl_DRS,LocList,LocList) --> [].
```

```
optrel(Gl_DRSIn,DRSIn,DRSOut,X^Cond,
       Gl_DRSOut,LocListIn,LocListIn) -->
        [that],vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,
                  Gl_DRSOut,LocListIn,_).

vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut,
   LocListIn,LocListOut) -->
        trans_verb(Gl_DRSIn,DRSIn,DRS1,Y^X^Cond,Gl_DRS1),
        np(Gl_DRS1,DRS1,DRSOut,Y,Gl_DRSOut,
           LocListIn,LocListOut).
vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut,
   LocListIn,LocListOut) -->
        trans_verb(Gl_DRSIn,Q,P,Y^X^Cond,Gl_DRS1),
        qnp(Gl_DRS1,DRSIn,DRSOut,Y^P^(Q=>P),
            Gl_DRSOut,LocListIn,LocListOut).
vp(Gl_DRSIn,DRSIn,DRSOut,X^Cond,Gl_DRSOut,
   LocList,LocList) -->
        intrans_verb(Gl_DRSIn,DRSIn,DRSOut,
                     X^Cond,Gl_DRSOut).

det(Gl_DRS,DRS,DRS,Gl_DRS) -->
        [a].

det(Gl_DRS,[Dom,Constr],[Dom,[Q=>P|Constr]],
    Q^P^(Q=>P),Gl_DRS) -->
        [every].

noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [man(X),male(X)|Constr]],X^man(X),Gl_DRS,
     LocListIn,[X|LocListIn]) -->
        [man].
noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [woman(X),female(X)|Constr]],X^woman(X),Gl_DRS,
     LocListIn,[X|LocListIn]) -->
        [woman].
noun(Gl_DRS,[Dom,Constr],[[X|Dom],
     [donkey(X),neuter(X)|Constr]],X^donkey(X),Gl_DRS,
     LocListIn,[X|LocListIn]) -->
        [donkey].

proper_noun(Gl_DRSIn,DRS,DRS,X,Gl_DRSOut,
            LocListIn,[X|LocListIn]) -->
        [john],{check_add(X,[named(X,john),male(X)],
                          Gl_DRSIn,Gl_DRSOut),
        \+ strict_member(X,LocListIn)}.
proper_noun(Gl_DRSIn,DRS,DRS,X,Gl_DRSOut,
            LocListIn,[X|LocListIn]) -->
        [mary],{check_add(X,[named(X,mary),female(X)],
                          Gl_DRSIn,Gl_DRSOut),
        \+ strict_member(X,LocListIn)}.


pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS,
        LocListIn,[X|LocListIn]) -->
```

```
        [he], {(check(X,[male(X)],DRS);
                 check(X,[male(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS,
        LocListIn,[X|LocListIn]) -->
        [him], {(check(X,[male(X)],DRS);
                  check(X,[male(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS,
        LocListIn,[X|LocListIn]) -->
        [she], {(check(X,[female(X)],DRS);
                  check(X,[female(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS,
        LocListIn,[X|LocListIn]) -->
        [her], {(check(X,[female(X)],DRS);
                  check(X,[female(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS,DRS,DRS,X,Gl_DRS,
        LocListIn,[X|LocListIn]) -->
        [it], {(check(X,[neuter(X)],DRS);
                 check(X,[neuter(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.

trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[love(X,Y)|Constr]],
           Y^X^love(X,Y),Gl_DRS) -->
        [loves].
trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[beat(X,Y)|Constr]],
           Y^X^beat(X,Y),Gl_DRS) -->
        [beats].
trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[own(X,Y)|Constr]],
           Y^X^own(X,Y),Gl_DRS) -->
        [owns].
trans_verb(Gl_DRS,[Dom,Constr],
           [Dom,[know(X,Y)|Constr]],
           Y^X^knows(X,Y),Gl_DRS) -->
        [knows].

intrans_verb(Gl_DRS,[Dom,Constr],
             [Dom,[live(X)|Constr]],
              X^live(X),Gl_DRS) -->
        [lives].


drs_sent(Sent,Gl_DRSIn,DRSIn,DRSOut,
         Gl_DRSOut,LocListIn,LocListOut) :-
        s(Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut,
          LocListIn,LocListOut,Sent,[]).

drs([],Gl_DRS,DRS,DRS,Gl_DRS).
drs([Sent|Disc],Gl_DRSIn,DRSIn,DRSOut,Gl_DRSOut) :-
```

```
        drs_sent(Sent,Gl_DRSIn,DRSIn,DRS1,Gl_DRS1,[],_),
        drs(Disc,Gl_DRS1,DRS1,DRSOut,Gl_DRSOut).

drs(Disc,DRS) :-
        drs(Disc,[[],[]],[[],[]],DRSOut,Gl_DRSOut),
        merge_drs(Gl_DRSOut,DRSOut,DRS).

merge_drs([Dom1,Constr1],[Dom2,Constr2],[Dom,Constr]) :-
        append(Dom1,Dom2,Dom),
        append(Constr1,Constr2,Constr).

check(X,Cs,[Dom,Constr]) :-
        member(X,Dom),
        strict_members(Cs,Constr).

check_add(X,Cs,DRS,DRS) :-
        check(X,Cs,DRS),!.
check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).

strict_member(X,[Y|L]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).

strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).
```

Chapter 9

Discourse structure and anaphora resolution

9.1 Introduction

In this chapter we will look at the paper on Focusing in the Comprehension of Definite Anaphora by Candy Sidner. We will first examine one of her example dialogues and introduce her algorithms by hand-running them on the discourse. We will then consider what it would mean to get the effect of her algorithms in the discourse representation framework we have been working with.

9.2 Sidner's focusing algorithms

The discourse we will examine is similar to the one labelled D29 in Sidner's paper:

1. Alfred and Zohar liked to play baseball.
2. They played it every day after school before dinner.
3. After their game, Alfred and Zohar had ice cream cones.
4. They tasted really good.
5. They were Italian and they often had chocolate sprinkles on
6. One day they met a man at the ice-cream parlour.
7. He told them that he had seen them playing.
8. He wanted them to play for his team.

Having parsed the first sentence, we set up some possibilities for what might be in focus, since it is things in focus that pronouns will get resolved to. To do this we use the Expected Focus Algorithm which is used only on the first sentence of the discourse:

Expected Focus Algorithm

Choose an expected discourse focus as:

1. The subject of a sentence if the sentence is an is-a or a there-insertion sentence.

2. The first member of the default expected focus list (DEF list), computed from the thematic relations of the verb, as follows:

Order the set of phrases in the sentence using the following preference schema:

theme unless the theme is a verb complement in which case the theme from the complement is used
all other thematic positions with the agent last
the verb phrase

There is a similar algorithm to choose an expected actor focus though this is not made explicit in the paper.

The result of applying these two algorithms to the first sentence is:

Default Expected Focus List - [baseball, Alfred and Zohar, liked to play baseball]

Discourse Focus - baseball

Actor Focus - Alfred and Zohar

Now we come to the second sentence. After parsing the first job is to resolve the pronouns given the information about focus we have from the first sentence. This is done by the rule R1 whose basic form is given by:

R1

If the pronoun under interpretation appears in a sentence thematic relation other than agent, choose the discourse focus as the co-specifier unless any of the syntactic, semantic and inferential knowledge constraints rule out the choice. If the pronoun appears in agent position, choose the actor focus as co-specifier in the same way.

This assigns the actor focus to *they* and the discourse focus to *it*. Now we apply the *Focusing algorithm* which is used on every sentence other than the first in the discourse [1]:

Focusing algorithm

Preliminaries: If the Focus Stack has not been set by previous applications of this algorithm, Focus Stack := [].

If the Discourse Focus has not been set by previous applications of this algorithm, Current Focus := Expected Focus computed by Expected Focus algorithm. Otherwise Current Focus := Discourse Focus.

Alternate Focus List := Potential Focus List, if there is one. Otherwise, Alternate Focus List := Default Expected Focus List.

0. Initialization: Make note of the existence of do-anaphora, anaphora co- specifying the Current Focus and the Alternate Focus List, implicit specifications, anaphors which specify elements not in the discourse or the lack of an anaphor use.

1. Do-anaphora: If the sentence contains do-anaphora, take the last member of the Alternate Focus List as the focus (Discourse Focus?). Stack the Current Focus in the Focus Stack and halt.

2. Focus set collection: If you are on the third sentence of the discourse or later and if focus sets are being collected and no anaphora occur in the current sentence, continue the collection. If some anaphor appears in the current sentence, use its co-specifier as the focus. Halt.

3. Choosing between Current Focus and the Alternate Focus List:

If there is an anaphor co-specifying the Current Focus and another co-specifying some member,F, of the Alternate Focus List,

then

> **if** neither anaphors corresponding to the Current Focus and F are in agent position
> **then**
>
> > **if** only F is represented by a pronoun
> > **then** (Discourse) Focus := F, push Current Focus onto Focus Stack and Current Focus := F
> > **else** retain Current Focus as Discourse Focus
> > **endif**

[1] I have slightly paraphrased Sidner's original definition.

else

if the anaphor corresponding to the Current Focus is not in agent position
then retain the Current Focus as (Discourse) Focus.
else (Discourse) Focus := F.
endif

endif

endif

Halt.

4. Retaining the Current Focus as (Discourse) Focus: If only the Current Focus has anaphors which co-specify it, retain the Current Focus as (Discourse) Focus. Halt.

5. Alternate Focus List as Focus: If the anaphors only co-specify a member of the Alternate Focus List, move the focus to it. If several members of the Alternate Focus List are co-specified, choose the focus in the manner suggested by the Expected Focus Algorithm. Halt.

6. Focus stack use: If the anaphors only co-specify a member of the focus stack, move the focus to the stack member by popping the stack. Halt.

7. Implicit specification: If a definite noun phrase implicitly specifies an element associated with the focus, retain the Current Focus and flag the definite noun phrase as implicit specification. If specification is associated with a member of the Alternate Focus List, move the (Discourse) Focus to that member and flag the definite noun phrase as implicit specification. Halt.

8. Lack of anaphora: If there are no anaphors co-specifying any of the Current Focus, Alternate Focus List or Focus Stack but the Current Focus can fill a non-obligatory case-role (argument-role) in the sentence or if the verb- phrase is related to the Current Focus by nominalization, retain the Current Focus and halt.

9. Focus set initialization: If there are no foci mentioned and the sentence is discourse initial, collect focus sets.

10. No focus used: Otherwise if there are no foci mentioned, retain the Current Focus as (Discourse) Focus.

Looking at the second sentence of our discourse this algorithm applies in the following way: The preliminaries will give us:

Focus Stack - []
Current Focus - baseball
Alternate Focus List - [(baseball?), Alfred and Zohar, liked to play baseball]

0. Initialization:

   We note that *they* co-specifies *Alfred* and *Zohar* on the Alternative Focus List and that *it* co-specifies the Current Focus, *baseball*.

1. Do-anaphora:

   This step does not apply.

2. Focus set collection:

   This step does not apply.

3. Choosing between Current Focus and the Alternate Focus List:

This step confirms *baseball* as the Current Focus. There are anaphors co- specifying the Current Focus (*it*) and another co-specifying some member of the Alternate Focus List (*they*). It is not the case that they are both in agent position, but it is the case that the anaphor corresponding to the current focus (*it*) is not in agent position. Thus we retain the Current Focus as the Discourse Focus. And halt.

After applying the focus algorithm, we compute the Potential Focus List in preparation for the next sentence.

Potential Focus List Algorithm

1. If a cleft of pseudocleft sentence is used, the potential focus is the cleft item if and only if the element is non-clefting position co-specifies the focus. When it does not, the sentence is incoherent.

2. Otherwise order a potential focus list of all the noun-phrases filling a thematic relation in the sentence, excluding a noun phrase in agent position and the noun phrase which co-specifies the focus if one exists. The last member of the Potential Focus List is the verb phrase of the sentence.

Clause (1) of this algorithm does not apply. By clause (2) we obtain

Potential Focus List - [every day, school, dinner, played it every day after school before dinner]

Now we move on to sentence (3) of the discourse. First the pronoun interpretation rule, R1. Sidner says that it is not detailed enough to resolve *their*. However, perhaps we can take their to be an agent, in which case R1 will require the Actor Focus to be co- specified, as desired. Sidner suggests that *their game* should count as a mention of *baseball*, and therefore counts as an anaphor co-specifying the focus and keeping it in focus. We enter the Focusing Algorithm with:

Focus Stack - []
Current Focus - baseball
Alternate Focus List - [every day, school, dinner, played it every day after school before dinner]

If *their* is regarded as the only anaphor, then the only clause of the Focus Algorithm that applies is (10) which maintains the current discourse focus. If *their game* is counted as an anaphor then this also will maintain the current discourse focus, by clause (4) since *Alfred* and *Zohar* are no longer maintained on the Alternate Focus List. Thus either way, *baseball* continues as the Discourse Focus. Computing the Potential Focus List yields:

Potential Focus List - [ice-cream cones, had ice-cream cones]

We now come to the fourth sentence. Pronoun Interpretation goes as follows: *they* is not in agent position, thus we try to choose the discourse focus as the thing co-specified. However, *baseball* does not match *they*, firstly for syntactic reasons (there is a clash between singular and plural) and secondly for semantic reasons (baseball doesn't taste of anything). So Sidner suggests that the rule R1 needs to be made more sophisticated. In case of such a failure the Potential Focus List is to be used to resolve the pronouns. This provides us with *ice-cream cones* as desired.

We now move on to the Focusing Algorithm. The preliminaries yield:

Focus Stack - []
Current Focus - baseball
Alternate Focus List - [ice-cream cones, had ice-cream cones]

During Initialization we notice that there is one anaphor which refers to something on the Alternate Focus List. Clause (5) is the one that is relevant. We thus move the focus to *ice-cream cones*. This means that we set Current Focus := ice-cream cones and push *baseball* onto the Focus Stack, i.e. Focus Stack := [ice-cream cones].

Finally, computing the Potential Focus List for this sentence yields [tasted really good].

Exercise 9.2.1

Analyze the rest of the discourse in a similar fashion.

Exercise 9.2.2

Explain how the program `focus-drs.pl` treats the discourse

> A man owns a donkey. It loves him.

9.3 Programmes for Chapter 9

9.3.1 focus-drs.pl

```
/*  A simple implementation of discourse representation using
    complex drs's for universal quantification*/

/*  You may need to define member and append, depending on
    the prolog you are using */

:- op(150,xfy,=>).
:- op(100,xfy,:).

s(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,
  LocList-LocList,FocIn-FocOut) -->
        np(Gl_DRSIn-Gl_DRS1,DRSIn-DRS1,X,
           []-LocList1,FocIn-Foc1),
        vp(Gl_DRS1-Gl_DRSOut,DRS1-DRSOut,X^Cond,
           LocList1-_,Foc1-FocOut).
s(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,
  LocList-LocList,FocIn-FocOut) -->
        qnp(Gl_DRSIn-Gl_DRS1, DRSIn-DRSOut,X^P^(Q=>P),
            []-LocList1,FocIn-Foc1),
        vp(Gl_DRS1-Gl_DRSOut,Q-P,X^Cond,
           LocList1-_,Foc1-FocOut).

np(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,_:X,
   LocListIn-LocListOut,FocIn-FocOut) -->
        det(Gl_DRSIn-Gl_DRS1,DRSIn-DRS1,FocIn-Foc1),
        noun(Gl_DRS1-Gl_DRS2,DRS1-DRS2,X^Cond,
             []-LocList1,Foc1-Foc2),
        optrel(Gl_DRS2-Gl_DRSOut,DRS2-DRSOut,X^Cond1,
               LocList1-_,Foc2-FocOut),
        {append(LocListIn,LocList1,LocListOut)}.

qnp(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,_:X^P^DRS,
    LocListIn-LocListOut,FocIn-FocOut) -->
        det(Gl_DRSIn-Gl_DRS1,DRSIn-DRSOut,Q^P^DRS,
            FocIn-Foc1),
        noun(Gl_DRS1-Gl_DRS2,DRSIn-DRS1,X^Cond,
             []-LocList1,Foc1-Foc2),
        optrel(Gl_DRS2-Gl_DRSOut,DRS1-Q,X^Cond1,
               LocList1-_,Foc2-FocOut),
        {append(LocListIn,LocList1,LocListOut)}.

np(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X,
   LocListIn-LocListOut,FocIn-FocOut) -->
        proper_noun(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X,
                    LocListIn-LocListOut,FocIn-FocOut).

np(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X,
   LocListIn-LocListOut,FocIn-FocOut) -->
        pronoun(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X,
```

```
                LocListIn-LocListOut,FocIn-FocOut).

optrel(Gl_DRS-Gl_DRS,DRS-DRS,X^_,LocList-LocList,Foc-Foc) --> [].
optrel(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X^Cond,
       LocListIn-LocListIn,FocIn-FocOut) -->
        [that],vp(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X^Cond,
                  LocListIn-_,FocIn-FocOut).

vp(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X^Cond,
   LocListIn-LocListOut,FocIn-FocOut) -->
        trans_verb(Gl_DRSIn-Gl_DRS1,DRSIn-DRS1,Y^X^Cond,
                   FocIn-Foc1),
        np(Gl_DRS1-Gl_DRSOut,DRS1-DRSOut,Y,
           LocListIn-LocListOut,Foc1-FocOut).

vp(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X^Cond,
   LocListIn-LocListOut,FocIn-FocOut) -->
        trans_verb(Gl_DRSIn-Gl_DRS1,Q-P,Y^X^Cond,
                   FocIn-Foc1),
        qnp(Gl_DRS1-Gl_DRSOut,DRSIn-DRSOut,Y^P^(Q=>P),
            LocListIn-LocListOut,Foc1-FocOut).
vp(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,X^Cond,
   LocList-LocList,FocIn-FocOut) -->
        intrans_verb(Gl_DRSIn-Gl_DRSOut,
                     DRSIn-DRSOut,X^Cond,FocIn-FocOut).

det(Gl_DRS-Gl_DRS,DRS-DRS,Foc-Foc) -->
        [a].

det(Gl_DRS-Gl_DRS,[Dom,Constr]-[Dom,[Q=>P|Constr]],
    Q^P^(Q=>P),Foc-Foc) -->
        [every].

noun(Gl_DRS-Gl_DRS,[Dom,Constr]-[[X|Dom],
     [man(X),male(X)|Constr]],X^man(X),
     LocListIn-[X|LocListIn],Foc-Foc) -->
        [man].
noun(Gl_DRS-Gl_DRS,[Dom,Constr]-[[X|Dom],
     [woman(X),female(X)|Constr]],X^woman(X),
     LocListIn-[X|LocListIn],Foc-Foc) -->
        [woman].
noun(Gl_DRS-Gl_DRS,[Dom,Constr]-[[X|Dom],
     [donkey(X),neuter(X)|Constr]],X^donkey(X),
     LocListIn-[X|LocListIn],Foc-Foc) -->
        [donkey].

proper_noun(Gl_DRSIn-Gl_DRSOut,DRS-DRS,_:X,
            LocListIn-[X|LocListIn],Foc-Foc) -->
        [john],{check_add(X,[named(X,john),male(X)],
                          Gl_DRSIn,Gl_DRSOut),
        \+ strict_member(X,LocListIn)}.
proper_noun(Gl_DRSIn-Gl_DRSOut,DRS-DRS,_:X,
            LocListIn-[X|LocListIn],Foc-Foc) -->
        [mary],{check_add(X,[named(X,mary),female(X)],
```

```
                          Gl_DRSIn,Gl_DRSOut),
        \+ strict_member(X,LocListIn)}.


pronoun(Gl_DRS-Gl_DRS,DRS-DRS,R:X,
        LocListIn-[X|LocListIn],
        (Foci,Anaphors,NewInfo)-
          (Foci,[X|Anaphors],NewInfo)) -->
        [he],
        {r1(Foci,R:X),
        (strict_check(X,[male(X)],DRS);
          strict_check(X,[male(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS-Gl_DRS,DRS-DRS,R:X,LocListIn-[X|LocListIn],
        (Foci,Anaphors,NewInfo)-
          (Foci,[X|Anaphors],NewInfo)) -->
        [him],
        {r1(Foci,R:X),
        (strict_check(X,[male(X)],DRS);
          strict_check(X,[male(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS-Gl_DRS,DRS-DRS,R:X,LocListIn-[X|LocListIn],
        (Foci,Anaphors,NewInfo)-
          (Foci,[X|Anaphors],NewInfo)) -->
        [she],
        {r1(Foci,R:X),
        (strict_check(X,[female(X)],DRS);
          strict_check(X,[female(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS-Gl_DRS,DRS-DRS,R:X,LocListIn-[X|LocListIn],
        (Foci,Anaphors,NewInfo)-
          (Foci,[X|Anaphors],NewInfo)) -->
        [her],
        {r1(Foci,R:X),
        (strict_check(X,[female(X)],DRS);
          strict_check(X,[female(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.
pronoun(Gl_DRS-Gl_DRS,DRS-DRS,R:X,LocListIn-[X|LocListIn],
        (Foci,Anaphors,NewInfo)-
          (Foci,[X|Anaphors],NewInfo)) -->
        [it],
        {r1(Foci,R:X),
        (strict_check(X,[neuter(X)],DRS);
         strict_check(X,[neuter(X)],Gl_DRS)),
        \+ strict_member(X,LocListIn)}.

trans_verb(Gl_DRS-Gl_DRS,[Dom,Constr]-
                          [Dom,[love(exp:X,th:Y)|Constr]],
        th:Y^exp:X^love(exp:X,th:Y),
        (Foci,Anaphors,NewInfo)-
         (Foci,Anaphors,[love(exp:X,th:Y)|NewInfo])) -->
        [loves].
trans_verb(Gl_DRS-Gl_DRS,[Dom,Constr]-
                          [Dom,[beat(ag:X,th:Y)|Constr]],
```

```
        th:Y^ag:X^beat(ag:X,th:Y),
        (Foci,Anaphors,NewInfo)-
          (Foci,Anaphors,[beat(ag:X,th:Y)|NewInfo])) -->
        [beats].
trans_verb(Gl_DRS-Gl_DRS,[Dom,Constr]-
                          [Dom,[own(ag:X,th:Y)|Constr]],
        th:Y^ag:X^own(ag:X,th:Y),
        (Foci,Anaphors,NewInfo)-
          (Foci,Anaphors,[own(ag:X,th:Y)|NewInfo])) -->
        [owns].
trans_verb(Gl_DRS-Gl_DRS,[Dom,Constr]-
                          [Dom,[know(exp:X,th:Y)|Constr]],
        th:Y^exp:X^knows(exp:X,th:Y),
        (Foci,Anaphors,NewInfo)-
          (Foci,Anaphors,[know(exp:X,th:Y)|NewInfo])) -->
        [knows].

intrans_verb(Gl_DRS-Gl_DRS,[Dom,Constr]-
                            [Dom,[live(th:X)|Constr]],
        th:X^live(th:X),
        (Foci,Anaphors,NewInfo)-
         (Foci,Anaphors,[live(th:X)|NewInfo])) -->
        [lives].


drs_sent(Sent,Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,
         LocListIn-LocListOut,FocIn-FocOut) :-
        s(Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,
          LocListIn-LocListOut,FocIn-Foc1,Sent,[]),
        update_foc(Foc1,FocOut).

drs([],Gl_DRS-Gl_DRS,DRS-DRS,Foc-Foc).
drs([Sent|Disc],Gl_DRSIn-Gl_DRSOut,DRSIn-DRSOut,FocIn-FocOut) :-
        drs_sent(Sent,Gl_DRSIn-Gl_DRS1,
                 DRSIn-DRS1,[]-_,FocIn-Foc1),
        drs(Disc,Gl_DRS1-Gl_DRSOut,DRS1-DRSOut,Foc1-FocOut).

drs(Disc,DRS) :-
        drs(Disc,[[],[]]-Gl_DRSOut,[[],[]]-DRSOut,
            ((nil,nil,[],[]),[],[])-FocOut),
        merge_drs(Gl_DRSOut,DRSOut,DRS).

drs(Disc,DRS,FocOut) :-
        drs(Disc,[[],[]]-Gl_DRSOut,[[],[]]-DRSOut,
            ((nil,nil,[],[]),[],[])-FocOut),
        merge_drs(Gl_DRSOut,DRSOut,DRS).

merge_drs([Dom1,Constr1],[Dom2,Constr2],[Dom,Constr]) :-
        append(Dom1,Dom2,Dom),
        append(Constr1,Constr2,Constr).

r1((_,ActorFocus,_,_),ag:ActorFocus).
r1((DiscourseFocus,_,_,_),th:DiscourseFocus).
r1((DiscourseFocus,_,_,_),exp:DiscourseFocus).
```

```
r1((_,_,AltFocusList,_), _:X) :-
        member(X,AltFocusList).
r1((_,_,_,[X|FocusStack]), _:X).

update_foc((FociIn,Anaphors,NewInfo),(FociOut,[],[]) ) :-
        expected_focus(FociIn,FociOut,Anaphors,NewInfo),!.
update_foc((FociIn,Anaphors,NewInfo),(FociOut,[],[]) ) :-
        focus(FociIn,FociOut,Anaphors,NewInfo).

expected_focus((X,Y,[],[]),
               (DiscourseFocus,ActorFocus,AltFocusList,[]),
        _,[Cond|NewInfo]) :-
        X==nil,Y==nil,
        role(th,Cond,DiscourseFocus),
        role(ag,Cond,ActorFocus),
        other_roles(Cond,AltFocusList1),
        (\+ ActorFocus ==
            nil -> append(AltFocusList1,
                          [ActorFocus],AltFocusList);
        AltFocusList1=AltFocusList).

role(R,Cond,X) :-
        Cond =.. [Pred|Args],
        member(R:X,Args),!.
role(_,_,nil).

other_roles(Cond,L) :-
        Cond =.. [Pred|Args],
        bagof(X,R^(member(R:X,Args),\+(R=ag;R=th)),L),!.
other_roles(_,[]).

focus((DFocusIn,AFocusIn,AltFLIn,FStackIn),
      (DFocusOut,AFocusIn,AltFLOut,FStackIn),
        Anaphors,[Cond|NewInfo]) :-
        strict_member(DFocusIn,Anaphors),
        strict_shared_member(Anaphors,AltFLIn,X),
        role(ag,Cond,A),
        (\+A==DFocusIn -> DFocusIn = DFocusOut; X = DFocusOut),
        potential_focus(DFocusIn,Cond,AltFLOut).

focus((DFocusIn,AFocusIn,AltFLIn,FStackIn),
       (DFocusIn,AFocusIn,AltFLOut,FStackIn),
        Anaphors,[Cond|NewInfo]) :-
        strict_member(DFocusIn,Anaphors),!,
        potential_focus(DFocusIn,Cond,AltFLOut).

focus((DFocusIn,AFocusIn,AltFLIn,FStackIn),
      (X,AFocusIn,AltFLOut,[DFocusIn|FStackIn]),
        [X],[Cond|NewInfo]) :-
        strict_member(X,AltFLIn),!,
        potential_focus(DFocusIn,Cond,AltFLOut).

focus((DFocusIn,AFocusIn,AltFLIn,[Y|FStackIn]),
      (X,AFocusIn,AltFLOut,FStackIn),
```

```
        [X],[Cond|NewInfo]) :-
        X==Y,!,
        potential_focus(DFocusIn,Cond,AltFLOut).

focus((DFocusIn,AFocusIn,AltFLIn,FStackIn),
      (DFocusIn,AFocusIn,AltFLOut,FStackIn),
        _,[Cond|NewInfo]) :-
        potential_focus(DFocusIn,Cond,AltFLOut).

potential_focus(DFocusIn,Cond,AltFLOut) :-
        Cond =..[_|Args],
        bagof(X,R^(member(R:X,Args),\+R=ag,
              \+X==DFocusIn),AltFLOut),!.

potential_focus(_,_,[]).

check(X,Cs,[Dom,Constr]) :-
        member(X,Dom),
        strict_members(Cs,Constr).

strict_check(X,Cs,[Dom,Constr]) :-
        strict_member(X,Dom),
        strict_members(Cs,Constr).

check_add(X,Cs,DRS,DRS) :-
        check(X,Cs,DRS),!.
check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).

strict_member(X,[Y|L]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).


strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).

strict_shared_member([X|L1],L2,X) :-
        strict_member(X,L2).
strict_shared_member([_|L1],L2,X) :-
        strict_shared_member(L1,L2,X).
```

9.3.2 Examples

```
?- drs([[john,lives]],DRS,Foc).
    DRS = [[_64],
           [named(_64,john),male(_64),live(th : _64)]],
    Foc =  (_64, nil, [], []), [], [] ;
    no
```

```
?- drs([[john,loves,mary]],DRS,Foc).
    DRS = [[_110,_64],
           [named(_110,mary),female(_110),named(_64,john),
            male(_64),love(exp : _64,th : _110)]],
    Foc =  (_110, nil, [_64], []), [], [] ;
    no
?- drs([[a,man,owns,a,donkey]],DRS,Foc).
    DRS = [[_123,_54],
           [donkey(_123),neuter(_123),
            own(ag : _54,th : _123),man(_54),male(_54)]],
    Foc =  (_123, _54, [_54], []), [], [] ;
    no
?- drs([[a,man,owns,a,donkey],[it,lives]],DRS,Foc).
    DRS = [[_123,_54],
           [live(th : _123),donkey(_123),neuter(_123),
            own(ag : _54,th : _123),man(_54),male(_54)]],
    Foc =  (_123, _54, [], []), [], [] ;
    no
?- drs([[a,man,owns,a,donkey],[he,lives]],DRS,Foc).
    DRS = [[_123,_54],
           [live(th : _54),donkey(_123),neuter(_123),
            own(ag : _54,th : _123),man(_54),male(_54)]],
    Foc =  (_54, _54, [_54], [_123]), [], [] ;
    no
?- drs([[a,man,owns,a,donkey],[he,loves,it]],DRS,Foc).
    DRS = [[_123,_54],
           [love(exp : _54,th : _123),donkey(_123),
            neuter(_123),own(ag : _54,th : _123),man(_54),
            male(_54)]],
    Foc =  (_123, _54, [_54], []), [], [] ;
    no
?- drs([[a,man,owns,a,donkey],[it,loves,him]],DRS,Foc).
    DRS = [[_123,_54],
           [love(exp : _123,th : _54),donkey(_123),
            neuter(_123),own(ag : _54,th : _123),man(_54),
            male(_54)]],
    Foc =  (_123, _54, [_54], []), [], [] ;
    no
?- drs([[a,man,owns,a,donkey],[it,owns,him]],DRS,Foc).
    no
?- drs([[every,man,owns,a,donkey]],DRS,Foc).
    DRS = [[],
           [[[_55],[man(_55),male(_55)]] =>
             [[_129,_55],
              [donkey(_129),neuter(_129),
               own(ag : _55,th : _129),man(_55),male(_55)]]]],
    Foc =  (_129, _55, [_55], []), [], [] ;
    no
?-
```

Chapter 10

Summary

This course concerned Prolog and Natural Language Semantics. We took semantics in this context to mean the evaluation of expressions (natural language or logic) in databases - in particular specifying whether sentences are true with respect to a database. We used the predicate satisfy to compute this. There were seven main topics dealt with in the course.

10.1 Semantics of first order predicate calculus (FOPC).

Notice that in defining the predicate `satisfy` we prepared for things coming later in the course by giving it two arguments `satisfy(Situation,Formula)` rather than one `satisfy(Formula)` . We took a situation or database to be a collection of atomic facts in prolog, i.e. ground expressions like `like(john,mary)` - no variables and no negation.

10.2 Translation of English to FOPC

We gave a little program which translated a fragment of English into FOPC. This could be combined with the satisfy program to define satisfaction for English sentences.

10.3 Lambda calculus

The preceding program involved a good deal of passing around of extra arguments in order to compensate for the mismatch between the syntactic structures of English and FOPC. The lambda-calculus provides a logical expression corresponding to each subconstituent of the English syntax and thus, in principal at least, makes the programming of the mapping between natural language syntax and logical form more straightforward. We did not do semantics directly for the lambda-calculus. Rather we relied on beta- reduction to reduce the lambda-calculus expressions to expressions of FOPC which could then be fed to `satisfy` .

10.4 Quantifiers

We then looked at some semantic problems with FOPC as a logical form language for natural language. In particular we looked at generalized quantifiers (including *most*, *few*, *many*), how the generalized quantifier analysis from the theoretical literature might be exploited in prolog and relations to the three branch quantifier (3BQ) representation.

We also looked at problems in the computation of different quantifier scopings and the Hobbs and Shieber algorithm for doing this.]

10.5 Questions and Quantifiers in CHAT

We looked at the CHAT-80 system as an example of a system which employs something like this logic programming approach to sentence semantics and compared CHAT's satisfy-predicate with the variants we had developed. We looked in some detail at the treatment of questions in CHAT (semantic representation, and a little about the optimization used when querying the database). We went through the technique of gap-threading and the treatment of questions given in sample programs in Pereira and Shieber's book.

10.6 Discourse anaphora

We now went beyond the level of the sentence and looked in some detail at the implementation of Discourse Representation Theory (DRT). There are two main aspects to this: the way in which we derive DRSs during the parsing of the sentence and the semantics we give for the DRSs. Having seen that it is possible to give a declarative account of the construction of DRSs on the basis of natural language syntax we showed how it might be possible to give a declarative version of Hobbs' algorithm for assigning discourse antecedents using the same kind of techniques. We went through Hobbs' algorithm in some detail and showed how part of it can be encoded in a logic grammar format using threading of local discourse referent lists.

10.7 Focus and discourse anaphora

We went in some detail through certain aspects of Sidner's algorithms concerning focus and showed how a declarative version of her algorithms might be grafted onto the treatment of DRT we had given previously.