The 36 Chambers of SHRDLU

Randall A. Helzerman

October 13, 2014

```
*/

:-use_module(library(lists)).
:-use_module(library(ugraphs)).
%:-use_module(library(swi)).

:-discontiguous(clause/1).
:-discontiguous(reduce/3).
:-discontiguous(category/3).
:-discontiguous(combine_sem/6).
:-discontiguous(process_logical_form/2).

:-dynamic(clause/1).
:-dynamic(axiom/1).

/*
```

Contents

Chapter 1

Introduction

SHRDLU was an amazing program for natural language understanding, written by Dr. Terry Winograd. SHRDLU's expertise was limited to conversations about the "blocks world", which is a restricted domain consisting of a tabletop with a bunch of colored blocks on it. The blocks have various shapes, and can be stacked on top of each other or arrainged in various ways, subject to a rudimentary set of physical laws.

The user could have conversations with SHRDLU about this domain, asking it questions about the blocks on the table and the configurations that they were in, and instructing it to create towers by stacking blocks on top of each other.

This is an example conversation which the users could have with SHRDLU:

```
PERSON: Is there a large block behind a pyramid?
SHRDLU: Yes, three of them.  A large red one, a large green
        cube, and the blue one.
PERSON: Put a small one onto the green cube which supports
        a pyramid.
SHRDLU: OK
PERSON: Does the shortest thing the tallest pyramid's support
        supports support anythying green?
SHRDLU: Yes, the green pyramid.
```

In addition to text input-output, SHRDLU had three-dimensional graphic output which displayed the state of the blocks world and animated the moving of blocks according to the commands of the user.

But the most amazing thing about SHRDLU was that it was written over 30 years ago! The Computing power of children's playthings have literally millions of times more processing power than the PDP-6 which hosted SHRDLU. But today there are no examples of natural language processing programs remotely appraching SHRDLU's ability to understand complex sentences. What happened?

This book is an exploration of of what we've learned about natural language processing since SHRDLU. The method of exploration is to reimplement

SHRDLU from scratch, using the latest syntactic and semantic tricks we have learned in the intervening 36 years.

1.1 Why all these chambers??

The title of this book is an allusion to the Shaw Brothers film "The 36th Chamber of Shaolin," which is indubitably one of the greatest martial arts film of all time. A young man enters the Shaolin Temple to learn Kung-Fu. As he learns Kung-fu, he progresses through 36 chambers, each chamber containing increasingly difficult physical and mental hardships and skills he must master. The entire process bears more than a passing resemblance to grad school.

But the real inspiration behind this chambered approach is from a suggestion by Fred Brooks in his book "The Mythical Man Month" in which he suggests that we change the metaphor from "architecting" a program to "growing" a program. Just as a living organism starts small but is nevertheless a complete living organism and grows bigger, Similarly, we should try as soon as possible to form a working computer program, and "grow" it by making each of the subsystems incrementally more and more powerful. An excellent example of how effective this approach can be in the context of natural language processing is Patrick Blackburn and Johan Bos's book "Natural Language Understanding". The start with a small but complete system called "CURT" and then they grow it, and the sucession of ever cleverer CURTs gives the reader a smooth on-ramp which greatly eases the burden of understanding.

This book follows a similer pattern. We get a very small, but complete system working as quickly as possible, and then each suceeding chamber illustrates how to enhance the functionality of one of the subsystems, or how to integrate a new functionality into an already working system.

1.2 Why are your programs half-baked?

This book was written primarily for didactic purposes. The emphasis is on understandability. Some of the routines are simplistic, and don't handle all of the corner cases which a full, robust system would have to handle. Most of the time, this is as a set-up for the next chamber–experiencing the painful limitations of one program are the best motivation for the next chamber. But sometimes, I rest content with sub-optimal solution, or a solution which doesn't handle all of the corner cases. If you find a routine like this, congratulations! Pat yourself on the back, and write the correct version in your program.

1.3 Why are you using Prolog instead of (insert favorite language here)

Prolog is, these days, a very arcane, indeed archaic, language. Although it had good P.R. during the 80's, it never made the leap from niche to mainstream–why not? There are many reasons (see [**?**] for heroic attempts to overcome some of these drawbacks) but the can be summarized like this: It is said that a good programming language should have two virtues:

1. It should mke simple tasks easy.

2. It should make difficult tasks possible.

What really prevented prolog from ever being a mainstream programming language is that sorely lacks the first virtue. So why use it? Well, prolog fairs much better on the second virtue, and in addition, it has a third virtue uniquely its own, which most other programing language *don't* have:

1. It makes (otherwise) impossible tasks possible.

In this reguard, I think it would be interesting to consider the example of the book "Paradigms of AI Programming" by Peter Norvig, which is, in the opinion of this author, the best book on programming ever written. In his book, Norvig describes in detail several natural language processing programs, with varying capabilities. When looking at these programs, an interesting pattern emerges: any program which needs more than the most basic powers of analyzing the structure of sentences is written in prolog. Norvig's preferred programming languge is LISP, but on the way to natural language processing, Norvig finds it useful to *first* implement prolog in LISP and *then* implement his natural language processing programs on top of prolog.

This is by no means an isolated case. Carl Hewitt's invention of planner, for example, or (even more pertinently) Terry Winograd's SHRDLU. Because both of these authors emphasized the *doing* part of understanding language (e.g. to understand "the block" go and *find* a block in the database) it might seem strange to cite these as examples of using prolog as an abstraction layer. However, Hewitt has emphasized that prolog is really a version of planner, and therefore SHRDLU (which was also written in a version of planner called MICROPLANNER) very much exemplifies this strategy of using a prolog-esque layer of abstraction on top of LISP.

This of course, is obvious only in hindsight. The whole LISP/prolog dichotomy is a false dichotomy. No matter what programming language you use, you will find it useful to structure your programs such that they have many levels of abstraction on top of assembly language, and one of these levels of abstraction is gunna look a lot like prolog. Since this is inevitable anyways, we might as well start not from assembly language, or even LISP, but from prolog. We can thereby help ourselves to over 30 years worth of prolog research, which

has produced very fast and efficient prolog compilers, as well as libraries, programming techniques and methodologies, and a wealth of practical know-how in constructing and maintaining programs.

1.4 Why don't you make the programs downloadable?

Typing programs in from the printed page has a venerable history. This book is written to help you learn. If you are a novice prolog programmer or linguist, typing in programs is a great way to fix the concepts in your mind. If you are NOT a novice, you probably already have several NLP systems which you've already written lying around–in which case you won't want to use the programs in this book as-is; you'll want to adapt them to your own programing language/environment/infrastructure. In neither case will you benefit at all by just downloading and running the program.

Chapter 2

The First Chamber: Prolog

This chapter is a brief introduction to the prolog language and some techniques and methodologies for using it effectively. Although the introduction is failrly complete, it is beyond the scope of this book to give a full introduction to prolog; the reader will no doubt want to suppliment with other introductory texts. Rather, the principle aim of this chapter is to make it easier to read the prolog programs in this book, by familiarizing the reader with the prolog language, discussing how to model the often contradictory and ambiguous knowledge of the real world in prolog, and by showing how the author used some basic tecniques to make it easier to synthesize some tricky programming problems.

2.1 Variables

Sometimes, there is an object you you want to talk about. But you don't exactly know *which* object it is. But even though the exact identity of the object is unknown to you, you do know a few facts about it. A good example of this is the game of 20 questions:

```
Is it smaller than a breadbox?  yes
Is it red?                       no
Is it round?                    yes
```

The game is to use what you *do* know about the unknown object to find out *which* object it is. In algebra class, you learned another way to do this: use a letter in place of a number:

$$2 * X = 4$$

We don't know exactly *which* number X is, but we do know that if you add 2 to it you get 4. Prolog uses variables much the same way that algebra does. The general gameplan is to using a variable to find out as many facts as you can about the unknown object. If you know enough *other* facts about it, you can figure out which object you *are* talking about.

Syntactically, a variable in prolog is:

1. an upper-case letter
2. optionally followed by any number of upper or lower case letters, or the underscore character '_'.

for example:

```
X  Y  Distance  This_is_a_variable
```

In algebra, you could only talk about numbers using variables. Prolog is a bit more flexible. The kinds of things we can talk about in prolog are *atoms* and *terms*.

The most basic things we can talk about are atoms. Atoms can be numbers (as in algebra), but they can also be strings of characters of this form:

1. a lower case letter
2. optionally followed by any number of upper or lower case letters, or the underscore character '_'.

for example, here are some atoms:

```
abraham  issac jacob
my_left_foot
```

here are some items which are not atoms

```
f(x)
a+b
‘‘this is not an atom’’
```

those are all *terms*. A term is a way of creating a new object out of atoms (kind of like in chemestry how molecules are made out of atoms). For example, we want to represent a burrito, made out of a tortilla, beans, shredded chease, and salsa:

```
burrito(tortilla,beans,shreadded_cheese,salsa).
```

a term has a *functor* (in the above case, its "burrito") and *arguments* which appear in parenthesis after the functor.

So, to reherse: prolog lets you name things with variables, and the things which are named by the variables in prolog are atoms and terms. To see how we can assign a name to a term, lets go to the command line.

2.2 Read-Eval-Print Loop

After you launch prolog, the command line prompt looks like this:

```
?-
```

At the prompt, you can give an atom a name like this:

```
?-  X = 5.
```

and prolog prints out:

```
X=5
?-
```

We could also do a term:

```
?- X = burrito(tortilla,beans,shreadded_cheese,salsa).

X = burrito(tortilla,beans,shreadded_cheese,salsa).
?-
```

2.3 Unification

Try typing in:

```
?-  X = 1+1.
```

You might be surprised that you get:

```
X = 1+1
?-
```

why didn't prolog do the addition? The answer is that "1+1" is a term. The "+" is a functor (much like the functor "burrito" above), but instead of being written as "+(1,1)" in front of its arguments, like the "burrito" functor was, it was written in between them. The "+" operator has what is called *infix* behavior. There several other operators which can be used in an infix way like this (the other arithmetic operators being obvious examples).

But for our present purposes, this means that "1+1" is not the same term as "2". In fact, they are different categories, "1+1" being a term, and "2" being an atom. This brings us to the "=" sign in prolog. In prolog, two things are "=" to each other when:

1. They are same term or atom, e.g. "1+1 = 1+1".
2. They can be made to *be* the same term by assigning some values to variables. e.g. "X = 1+1".

Given the above, this behaviour shouldn't be surprising to you:

```
?- 1+1=X.

X = 1+1
?-
```

"=" is completely symmetric: it doesn't matter what side of the = sign the variable is on. But "=" is even more flexible than this. The variables which we are assigning can be inside of a term:

```
?- 1+1=X.

X = 1+1
?-
```

What prolog has done here is find *what value* needs to be assigned to the variable "X" in order to make "1+X" the same term as "1+1". Obviously, the answer is "X=1" as prolog prints out.

The sort of atom-and-term equality which prolog usese is called *unification*. Two terms are said to *unify* if they are already the same term, or they can be made into the same term by instantiating any variables they contain.

2.4 Numerical Computations

No doubt you are wondering now how wec ould use prolog to actually add two numbers if we needed to. Prolog provides this with the "is" keyword. For example:

```
?-  X is 1 + 1.

X = 2.
```

which no doubt looks very odd. Using "=" for unification, and then using "is" for computing numerical results is counterintuitive, but it is one of the many charming eccedntricities which you'll get used to as you get familier with prolog.

2.5 Relations

To use prolog, you need to learn to use a new language, a kind of"Prolog-ese" (sometimes more derisively revered to as "Komputerdeutch"). The easiest way to learn it is to see some examples. We'll use a biblical example, from the King James Version, of course. We will translate a bit of the genology of Jesus into prolog.

> Mathew 1:14-16 Eleazar begat Matthan, and Matthan begat Jacob, and Jacob begat Joseph, the husband of Mary, of whome was born Jesus

To translate "Eleazar begat Matthan" into prolog-ese, we first ask the question: what is the relationship between Eleazar and Mathan? Typically, this relationship is expressed by the verb and associated words, in this case simply "begat". We then indicate that "Abraham" and "Jacob" stand in this relationship to each other by putting them in parenthesis behind the relationship, like this. Open up the editor and type in these names:

```
begat(eleazar,matthan).
```

Notice, we can't capitalize the names, because strings beginning with capital letters are variables. Continuing the rest of the mini-geneology:

```
begat(matthan,jacob).
begat(jacob,joseph).
```

now save and compile the file, and go to the command line. At the command line, we can now ask questions. First, lets ask some yes/no questions. "Is Eleazar the father of Matthan?" translates to:

```
?- begat(eleazar,matthan).

Yes
?-
```

We can also ask Wh-questions, like "Who is the father of matthan?" Note, this fits the pattern mentioned above. We know something about X, namely, that X is the father of Matthan. So we can use the fact that X is the father of Matthan to find out exactly who X is:

```
?- begat(X,matthan).

X = eleazar

Yes
```

Until now, we've only asked questions which have one answer. But most questions have many answers. Here is one such question: "Who is the father of whom?" There are many fathers and many sons, so this will question will have many answers. How to phrase it? We use two variables.

```
?- begat(X,Y).

X = eleazar,
Y = matthan
```

Prolog returns you *one* of the solutions. But it indicates to you that there may be more. Until now,we've just pressed return. But instead of pressing return, press ";":

```
?- begat(X,Y).

X = eleazar,
Y = matthan ;

X = matthan,
Y = jacob
```

and another solution appears. Pressing ";" again will cause the last solution to be displayed. This ability of prolog to explore *all possible* answers to a question is a big key to its power.

2.5.1 Some cautions

To illustrate a pitfall when extracting relations from English prose, lets go over to the *other* geneology of Jesus which is found in the New Testament.

> Luke 3:23-24 Joseph, which was the son of Heli, which was the son of Matthat, which was the son of Levi . . .

So first of all, we notice that English has several different ways to express the same concept. Matthew uses "begat" and Luke uses "was the son of". When you are translating English sentences into prolog, there is one thing you have to watch out for. Typically, what we do is pick one, and then stick with it.

But there's another pitfall here. Suppose we want to add the information found in Luke here to the information found in Matthew. So far we have:

```
% from Matthew
begat(eleazar,matthan).
begat(matthan,jacob).
begat(jacob,joseph).

% from Luke
begat(heli,joseph).
begat(matthat,heli).
begat(levi,matthat).
```

Compile and consult the file, and now lets ask who the father of Joseph is?

```
?- begat(X,joseph).

X = jacob ;

X = heli
```

We get two different fathers for Joseph! Now notice carefully, because this is probably the biggest cause of prolog bugs: In our minds, we have all kinds of knowledge about the "begat" relationship between father and son. One of those bits of knowledge is that a son is only begat from one father. But, of course, prolog doesn't know that–prolog only knows what we've told it. And we've told prolog here that Joseph was begat of two fathers, Jacob and Heli.

So what to do? Well, lets take our clue from Joseph's grandfather. Matthew says that Joseph's grandfather was Matthan, Luke says that he was Matthat. These look like just a minor spelling variation, so we'll assume that they both are the same name of the same person, just spelled wrong. But of course, to prolog, matthan $\neq$ matthat. So what we do is kind of the same thing we did for the "father of" relation–we just pick a name and stick with it. It doesn't matter which name we pick, but here's the thing–in prolog, everythign you talk about has to have one and only name. This is called the *unique names assumption*. Prolog will *always* assume that Matthan and Matthat are two different things.

So don't confuse prolog–just pick one name for each thing you want to talk about and stick with it.

But there's another problem here. "Heli" doesn't really look like an alternate spelling for "Jacob". Probably if you saw those two different names, you'd do what prolog does, and assume they were talking about two different people. So this illustrates another difficulty when translating English into Prolog. Most of the knowledge we have is self-contradictory. But how to handle it? We have several choices here.

1. We can either say that Jacob actally had two names (he was known as "Heli" to his poker buddies).

2. We can say that there was a scribal error here–somebody just goofed while copying. "Jacob" is the correct name. (Or we could say that "Heli" is the correct name and "Jacob" is the mistake.)

3. We can say that we were wrong about the "father" relation–scripture actually teaches us that a son can have two fathers, or ocassionally, 0 fathers due to a virgin birth.

The key here is that there's really nothing inherent in the translation process which is going to help us choose which way to go. Which one we choose depends upon our purposes and desires. The more different alternatives you can think of, the more likely it will be that you can fulfill your purposes and desires. This is one of the key differences between a mediocre and a top-flight programmer: a topflight programmer will always think of more possilbe ways to reconcile contradictory information and requirements.

For the present, we'll assume that there has been a scribal error, Luke goofed on the name of father of Jacob. Once we assume this, we can confirm our theory by noticing that Luke also misspeled "Matthan", and so we'll just stick with the Geneology in Matthew for the present. (Later whenw we extend prolog to disjunction, we'll consider a different method).

The key points to remember:

1. Each relation must have one unique name.

2. Each thing we talk about must have one unique name.

3. The more possible ways you can think to resolve contrdictory information, the better programmer (or theologen) you will be.

2.6 The Grandfather Clause

In our program, we've only talked about the relationship of fatherhood. What if we wanted to know about grandparents? In fact, there is enough information implicitly in the database to find out who the grandparent of, say, Joseph is. On the command line, with a little thought, we can formulate a query which will produce it:

```
?- begat(Parent,joseph), begat(Grandparent,Parent).

Parent = jacob,
Grandparent = matthan
```

Notice how we've strategically used a shared variable to link the two "begat" queries together. Both queries share the variable "Father", so when it is instantiated in the first query, it is used in the secong query to select the father of the father. Also notice, suppose we weren't interested in just Joseph's grandfather– suppose we wanted to be able to find anybody's grandfather. We could just replace the hard-coded "joseph" with a variable, and it would work like this:

```
?- begat(Parent,Kid), begat(GrandParent,Parent).
Parent = matthan,
Kid = jacob,
GrandParent = eleazar ;

Parent = jacob,
Kid = joseph,
GrandParent = matthan ;

No
?-
```

2.7 Rules

Now we know how to find grandfathers. There is a problem, however. Its tedious to have to keep typing two begats all the time, so suppose we wanted to modify our original program to have a clause "grandparent(Kid,Grandparent)" in it, which was true if "Grandparent" were bound to the grandparent of whatever "Kid" was bound to. There are two ways we could do it. We could, by hand, type in all the relations. But this is tedious, error prone, hard to update, and worst of all, redundant. Our database already contains the information we need, we just need to tell it how to derive grandparent information from begat information.

To do this, we need what is called a *rule*. Rules have two parts, a *head* and a *body*. Lets talk about the body first. The body of the rule is just a query, the same kind that you could type in on the command line. For our grandfather example, the body would be:

```
begat(Parent,Kid), begat(GrandParent,Parent).
```

The *head* of the rule is what you would like to name this query. For our example, we've said we wanted to name it:

```
grandparent(Kid,GrandParent)
```

Now, the head and the body of the rule are connected with what is called (surprise surprise) the *neck*. Its just the symbol ":-". So the whole rule will look like:

```
grandparent(Kid,GrandParent) :-
     begat(Parent,Kid), begat(GrandParent,Parent).
```

If you add that rule to your biblical geneology program and consult the file, then you will be able to use it to find the grandparents of people just the same way you were able to use "begat" to find the parents of people.

Notice that the "," which seperates begat(Parent,Kid) from begat(GrandParent,Parent) is acting like a conjunction, or the "and" logical constant. And is not the only logical constant we can use in rule bodies; There are three more constructs, disjunction, negation, and if-then-else.

The first is disjunction. For example, if we had a relation called "mother(X,Y)" which was true if X was the mother of Y, and a relation called "father(X,Y)" which was true if X was the father of Y, then we could write a predicate which was true if Y is a child, as follows:

```
child(Y) :- father(X,Y).
child(Y) :- mother(X,Y).
```

It is convenient and more perspicuous to have a shorthand for this. We can express the same thing using the disjunction operator, which in prolog is ";":

```
child(Y) :-
     father(X,Y)
;
     mother(X,Y).
```

we could have written it like this:

```
child(Y) :-
      father(X,Y) ;
      mother(X,Y).
```

but it aids in reading the program if we make it more obvious that we are using disjunction instead of conjunction. So the convention is to seperate disjuncts by a non-indented ";".

The next construct which can appear in rule bodies is negation. We will introduce how negation works in rules by considering how we might represent gender information in our genological database. We could add both male(X) and female(X) facts. But this requires more storage space than we actually need: for if somebody is male they are not female, and vice versa. Suppose we've only stored the information for the males:

```
mail(matthat).
mail(levi).
male(Jesus).
```

etc. If we add the following rule:

```
female(X) :- \+ male(X).
```

we could then conclude that mary was female, even though we've not explicitly stored mary's gender:

```
?- female(mary).

true.
```

A caveat about using this kind of default method to store knowledge: suppose we were to query whether my_left_foot were female:

```
?- female(my_left_foot).

true.
```

Why does this happen? Well, we have no fact "male(my_left_foot)" stored in in the databsae, so since my_left_foot is not explicitly male, we conclude that it is female. The "
+" construct is powerful, but needs to be used with caution.

The final construct resembles an if-then-else in traditional programming languages. Unlike disjunction, this has no straightforward translation into rules using only conjunction.

Suppose we had a predicate "male(X)" which was true if X was male. Then we could write child the following way:

```
child(Y) :-
     ( male(X) ->
         father(X,Y)
     ;
         mother(X,Y)
     ).
```

This first tests to see if X is male. If X is male, then it ususes the father relation to test if Y is a child of X. If X is not male, then it uses the mother relation to test if Y is a child of X.

In addition to the logical constants, we can use "is" in the body of a rule to perform numerical computation. This predicate is true if X is twice what Y is:

```
twice(X,Y) :-
    Y is 2*X.
```

We've now specified pretty much everything which can appear in prolog rules.

2.8 Counting and Recursion

Suppose there were a father who had three sons:

```
begat(father, son1). begat(father, son2). begat(father, son3).
```

We can enumerate these three on the command line:

```
?- begat(father, Son).
Son = son1 ;
Son = son2 ;
Son = son3 ;
```

but how could we count them? Right now, we can deal with multiple sons, but we're dealing them at seperate instances in time. Instead of dealing with one object at three different times, we want to be able to deal with three objects at the same time. Prolog provides a way to do this. It is called a *list*. Syntactically, lists are easy–just seperate what you want to put in the list by commans, and enclose the whole thing with square brackets, like this:

```
[1,2,3]
```

But how do we create one of these lists? Prolog provides a nice built-in function to do just this. Its called *findall/3* and it works like this:

```
?- findall(Son, begat(father,Son), Sons).
Sons = [son1,son2,son3]
```

The first argument is a variable. The second argument is a query, (which we could do on the command line), which contains that variable. The third argument is a list of all of the values for the variable which the query succeds on.

Now that we can get a list of the sons a father has, lets write a prolog rule which will count the number of items in the list. The rule will have two arguments. The first argument will be the list to be counted. The second argument will be the resulting count. For example:

```
?- count([a,b,c], Number).

Number = 3
```

If we were content to have this rule work only for a finite size of list (say it works only up until lists of 3 items long), we could write it purely by using unification, like this:

```
count([X],1).
count([X,Y],2).
count([X,Y,Z],3).
```

but of course we want this to work on arbitraritly large lists. In order to make that happen, we have to write a recursive proceedure.

The trick is to observe that the length of the list $[H|T]$ is 1+ the length of the list T. So suppose we had a magic rule which would give us the length of the list T. Call that routine "count". Then we could compute the count of the list $[H|T]$ like this:

```
count([H|T], N1) :-
     count(T,N),
     N1 is N + 1.
```

The only thing, of course, is that not all lists are of the form $[H|T]$. There is another list, the empty list, which we also have to take care of. The empty list has no elements, so we can just compute its length via the pattern matching method:

```
count([],0).
```

which completes the rule we were looking for. We know we are done because:

1. Ee have a rule which handles every kind of list, so we know that no list has been left out.

2. Each rule correctly calculates the length of the list it is given.

2.9 Skeletons and Techniques

At this point, you have been introduced to pretty much all of prolog's syntax. You might still very well be wondering right now about how to actually use this language to write the programs you want to write to get done what you want to get done. Does creating each new prolog procedure require a fresh burst of creative insight (as well as substantial intellectual effort)?

Fortunately no. There is a technique for constructing prolog predicates which can be used even on your non-creative days to synthesize the predicates you wish to write. The technique is very powerful and general; in fact most of the predicates in this book were generated via this very method. It is called "Skeletons and Techniques" and is what we will be talking about in this secition.

Skeltons and Techniques evolved out of earlier setpwise refinement techniques developed for imperative languages, and which apparently were first to adapted to prolog by Lakhotia [**?**]. Lee Naish [**?**] appears to be the first to notice the similarity to this and higher-order programming techniques. Naish and Sterling give the classical statement of Skeletons and Techniques in [**?**].

Larger programs are built out of elaborations of smaller programs. The "skeletons" of Skeletons and Techniquest are very very short programs. Frequently they are too short to be much use in and of themselves, but can serve as the seed to grow bigger and more useful programs.

As an example of a skeleton, consider this small predicate, which is true if its argument is a list: o

```
is_list([]).
is_list([H|T]) :- is_list(T).
```

To understand how it works, consider that there are two different kinds of lists:

1. The empty list.

2. A list of the form [H—T], where H is any prolog term and T is a list.

To this skeleton, we can apply one or more *techniques*. A technique is an elaboration of the basic skeleton, which adds some additional functionality. Consider again the "count" predicate which we programmed above, which counts the number of items in a list. We will reorder the clauses to make the commonality with the "is_list" skeleton more apparent:

```
count([],0).
count([H|T], N1) :-
     count(T,N),
     N1 is N + 1.
```

. The technique which was applied to the skelton here was:

1. Change name of predicate from "is_list" to count.

2. Add an additional argument, to hold/return the count of the items in the list.

3. Add an additional line after the recursive call to compute the partial sum of the count.

Notice, that with a slight variation of the technique, we can easily create another predicate, one which adds up all of the elements of a list (of course, this should be used only when all the elements of the list are really numbers).

```
% add up all of the object in a list
sum_list([],0).
sum_list([X|Xs],S) :-
        sum_list(Xs,S1),
        S is S1+X.
```

Frequently, it is useful to apply more than one technique to a single skeleton. For example, suppose we were challenged to write a predicate which took a list of numbers, and computed their average. Using the above predicates "sum_list" and "count" seperately, we might be tempted to write a predicate like this:

```
average(L,Average) :-
    sum_list(L,Sum),
    count(L,Count),
    Average is Sum / Count.
```

which would be correct. However, its not as efficient as it could be. It requires two iterations over the same list, one to compute the sum, and one to compute the count. But by applying both techniques to the same skeleton, we could create a predicated like "sum_and_count" which would compute both the sum and the count in a single pass of the list:

```
averate(L,Average) :-
   sum_and_count(Sum,Count),
   Average is Sum / Count.

sum_and_count([],0,0).
sum_and_count([H|T], Sum, Count) :-
     sum_and_count(T,Sum1,Count1),
     Sum is H + Sum1,
     Count is 1 + Count1.
```

whew! No doubt, before learning about skeletons and techniques, you might have looked at the above program for a long time and wondered exactly how it computed the average. And even when you figured it out, it might have been very mysterious as to why it was written that way.

But notice that if you can look at it and recognize the skeleton as being is_list, and that two techniques have been applied to the skeleton, you can quickly gain a very deep understanding of the predicate, and of the thought processes which the author went through to write the predicate.

So if you see a gnarly-looking predicate, which seems like it has a lot of extra arguments encrusted onto it, chances are it was generated via some kind of elaboration method like skeletons and techniques. To gain an understanding of it, see if you can recognize the skeleton it was based off of, and then see if you can make sense of the tecniques applied to that skeleton.

Stepping back a minute to summarize, the "Skeltons and Techniques" method of generating a procedure works as follows: say you need to write a procedure. Does the general flow of control of that procedure resemble a skeleton you know about? Or does the central data structure of your problem resemble that of a skeleton you know about? And if so, can you enhance that skeleton by applying a technique to do the particular computations you need?

Obviously, the bigger your library of skeletons and your library of techniques is, the more useful the method of skeletons and techniques will be to you. Space prohibits the enumeration of all of the skeletons and techniques we use in this book here (please see the cited references). But when one of our proceedures is derived via skeletons and techniques (and that happens often) we always note both the skeleton and the technique, to aid in understanding.

2.9.1 Self-Modifying Code

Prolog is virtually unique in making self modification of the code of a running program a core feature of the language. This feature is as you might expect, very

prone to being abused, and can result in a program which is can't be debugged unless you can solve the halting problem.

However, there are situations where its use is necessary, so we briefly mention it here. To add a new fact to the prolog database, you can use the assert predicate:

```
assert(my_new_predicate(X)).
```

You can use it on either the command line, or in the body of a rule of your program. To assert a rule, you also use assert, but you have to take care to put parenthesis around the rule so that the prolog parser doesn't get confused:

```
assert((my_new_rule(X) :- my_new_predicate(X))).
```

An example of where this could be useful would be in creating a new constant, i.e. writing a gensym like predicate. Most prologs these days have this predicate build-in, but if yours doesn't it is handy to know how to be able to add it.

```
gensym(Root, Atom) :-
        name(Root,Name1),
        get_num(Root,Num),
        name(Num,Name2),
        append(Name1,Name2,Name),
        name(Atom,Name).

get_num(Root,Num1) :-
        retract(current_num(Root,Num)),!,
        Num1 is Num + 1,
        asserta(current_num(Root,Num1)).

get_num(Root,1) :- asserta(current_num(Root,1)).
```

Chapter 3

The Second Chamber: Tokenization and Parsing

The tokenizer is a list-to-list transducer which takes the raw input list and converts it into something which is easier to write a parser for. Because it is a list transducer, we use difference lists in order to support constant-time concatenation.

A limitation of the tokenizer is that it assumes ascii input. It would have to be reworked if we were to support Unicode.

The tokenizer supplies two high-level predicates: snarf_line/1 and chunks/4. The predicate snarf_line/1 just keeps reading characters until an end line has been reached. The predicate chunks/4 takes a list of characters (such as that supplied by snarf_line/1) and breaks it into "chunks". Chunks correspond roughly to words and punctuation, with caveats as noted below. Generally, we follow the lead of the PENN Treebank project [**?**] on tokenization.

3.1 Getting a line of input from stdin

```
*/
get_sentence(S) :-
        snarf_line(L1),
        chunks(_Dummy,S,L1,_L2).

snarf_line(L) :-
        %  Get a character.
        get0(C),

        % process it.
        ( (C = 10) ->
            L = []
        ;
            snarf_line(Cs),
```

```
            L = [C|Cs]
        ).
/*
```

3.2 Chunks

Now, we break up the input list into "chunks":

```
*/

chunk([A|L]) -->
        optional_white_space,
        apostrophe(A),
        chunk(L).

chunk(W) -->
        optional_white_space,
        word(W).

chunk(W) -->
        optional_white_space,
        numeral(W).

chunk([W]) -->
        optional_white_space,
        non_chunking_punctuation(W).

/*
```

3.2.1 Whitespace

Chunks are seperated by whitespace, so we have to specify what counts as whitespace:

```
*/

white_space([H|T]) -->
        white_space_char(H),
        (
          white_space(T)
        ;
          end_white_space(T)
        ).

white_space_char(H) -->
        (
          space_char(H)
        ;
          tab_char(H)
```

```
        ).

space_char(32) --> [32].
tab_char(9)    --> [9].

end_white_space([]) --> \+ white_space_char(_H).

/*
```

We use this trick for optional white space:

```
*/

optional_white_space --> white_space(_S),!.
optional_white_space -->[].

/*
```

3.2.2 Recognizing Letters

```
*/

letter(H) -->
        (
          lower_case_letter(H)
        ;
          upper_case_letter(H)
        ;
          underscore(H)
        ).


lower_case_letter(H) -->
        [H],
        { H >= 97  },
        { H =< 122 }.

upper_case_letter(H) -->
        [H],
        { H >= 65 },
        { H =< 90 }.

underscore(95) --> [95].

/*
```

3.2.3 Words

The next step is to group segments of letters into words.

```
*/

end_word([]) --> \+ letter(_H).
word([H|T]) -->
        letter(H),
        ( word(T)
        ; end_word(T)
        ).


/*
```

3.2.4 Numerals

In the same way, we can group digits and numbers:

```
*/

digit(H) -->
        [H],
        { H >= 48 },
        { H =< 57 }.

end_numeral([]) --> \+ digit(_H).
numeral([H|T]) -->
        digit(H),
        (
          numeral(T)
        ;
          end_numeral(T)
        ).

/*
```

3.2.5 Punctuation

Words can also be seperated by punctuation:

```
*/

punctuation(C) -->
        apostrophe(C)
    ;
        non_chunking_punctuation(C).


non_chunking_punctuation(C) -->
        comma(C)
    ;
        semicolon(C)
```

```
    ;
        period(C)
    ;
        exclamation_mark(C)
    ;
        question_mark(C)
    ;
        hyphen(C).

comma(44) --> [44].

semicolon(59) --> [59].

apostrophe(39) --> [39].

period(46) --> [46].

exclamation_mark(33) --> [33].

question_mark(63) --> [63].

hyphen(45) --> [45].


/*
```

3.2.6 Defunkification

So-called "funky" words like "gonna" replaced by "gon na"

```
*/

:-discontiguous(funky/1).
:-discontiguous(defunkify/3).

funky(gunna).
funky(woulda).
funky(coulda).
funky(shoulda).
funky(something).
funky(somethings).

defunkify(gunna,[gun,na|X],X).
defunkify(woulda,[would,a|X],X).
defunkify(coulda,[could,a|X],X).
defunkify(shoulda,[should,a|X],X).
defunkify(something,[some,thing|X],X).
defunkify(somethings,[some,things|X],X).

/*
```

The reason we break things up like this is explained on the Penn treebank web page, "This tokenization allows us to analyze each component separately, so (for example) "I" can be in the subject Noun Phrase while "'m" is the head of the main verb phrase."

The following arn't funky in the sense of the Penn treebank, but since we already have the machinerty there, it is convenient to treat them as funky.

```
*/

funky(table).
defunkify(table,[w_table|X],X).

/*
```

3.2.7 Putting the chunks together

The following is really interesting...because a single chunk can induce more than one word in the resulting tokenized stream, we need to use append. To make that fast, we need to use difference lists. This makes the code much harder to understand...

```
*/

chunks(_H,[]) --> \+ chunk(_W).
chunks(H,T1) -->
        optional_white_space,
        chunk(H1),
        { name(H2,H1) },
        { ( funky(H2) ->
              defunkify(H2,T1,T2)
          ;
              T1 = [H2|T2]
          )
        },

        chunks(H,T2).
/*
```

Chapter 4

The Third Chamber: Grammars for Natural Language

Carve the bird at its joints – Julia Child

A grammar for a natural language is supposed to divide lists of words into two groups: those which belong to the language, and those which do not. But if we don't already have a grammar, how do we know which strings of words belong to the language, and which don't?

The answer is that we e start out with pre-theoretical intuition as to which sentences are grammatical sentences and which aren't. For example, the sentences:

```
A block is red.
A frog is green.
The block is on the table.
The frog is on the lillypad.
```

seem to be good English, while sentences like:

```
Green the is. *
block frog lillypad. *
```

don't seem to be. (It is customary to put a "*" after strings of words which arn't sentences.)

So where does this intuition come from? Well, we do understand English, and therefore some strings of words sound "OK" to us, and some sound funny. If this doesn't sound very scientific, well, its not. As Alen Perlis says, "you can't go from informal to formal by formal means." Our initial intuitions about which sentences are "correct" and which are "funny" is the only thing we have at first, but its enough to get started. How do we use them?

If we want to follow the advice of Julia Child, and "carve the bird at its joints," then we have to find the joints of the sentences, the places where the sentences can break down into their natural parts. What is a natural part? Well, a part is something which can be combined with other parts to make a whole. So we say that if we break a sentence into two, we've broken it into two parts if we can re-assemble those parts to make new grammatical sentences. For example: suppose we were to break the following sentences into two parts:

```
"A block is red"  =>  "A block" + "is red"
"A frog  is green"  =>  "A frog"  +  "is green"
```

we know that we've actually carved these sentences at the joints, because we can reassemble the pieces to make new grammatical sentences:

```
"A frog" + "is red" => "A frog is red"
```

But of course, just like taking puzzle pieces apart, we can't put these pieces back together any old way. For example:

```
"is red" + "A frog" + "=> "is red A frog"  *
```

yields a non-grammatical sentence. Again, we're just relying on our pre-theoretical intuition here. "is red a frog" doesn't sound right, but "a frog is red" does sound right.

But lets take a step towards formality now. Its not enough to say that a piece of a sentence is a part of a sentence: we must also indicate somehow how that peice could fit together with other pieces to form a whole. To do this, we will sort the parts of sentences into *categories*. Because the category is supposed to indicate how that part fits into the whole, we say that two different parts of sentences have the same category if we can substitute them in a grammatical sentence and still yield a grammatical sentence. For example: the phrases "A frog" and "The block" have the same category, becase we can substitute one for the other in a grammatical sentence and still yield a grammatical sentence:

```
[A frog] is green
[The block] is green
```

This means that "A frog" and "The block" will have the same category. Similarly, "is on the block" and "is green" will have the same category, because we can substitue them as well:

```
A frog [is on the block]
A frog [is green]
```

However, "is on the block" and "A frog" must be of different categories, because we cannot substitue one for the other without yielding an ungrammatical sentence:

```
[A frog] is green
[is on the block] is green *
```

So the parts of sentences are kind of like puzzle pieces: they will only fit together in certain ways.

Lets give names to these pieces. We've seen that

```
A frog
The block
```

are of the same category. It is traditional to call this category "NP" which stands for "noun phrase".

4.1 Formulating a grammar for blocks world

We want to talk about the blocks world, so we'll try to parse a few declarative sentences like this:

```
A block is green.
A block is red.
A table is green.
```

and we'll want to be able to parse questions too:

```
Is a block green?
Is a block red?
```

So lets see what kinds of categories we need to account for those sentences. We notice that we can substitute "block" for "table" in any grammatical sentence and still get a grammatical sentence:

```
A [table] is green.
A [block] is green.
```

Call this category "n" for ("noun"). We'll store this fact like this:

```
category(block, n).
category(block, n).
```

We also notice that larger groups of words can be so substituted:

```
[A table] is green.
[A block] is green.
[The table] is green.
[The block] is green.
```

We'll call this category "np" (for "noun phrase"). But how to store this fact? We wouldn't want to store it with every pair of words!

```
*/
:-op(400, yfx, \).
:-op(400, yfx, /).
```

```
/*
category(a,np/n).
category(the,np/n).
*/
category(is, (s(d)\np)/(n/n)).
category(is, (s(q)/(n/n))/np).

category(green,n/n).
category(red,n/n).
/*


parse([S],[], S).

parse(Stack,[Word|Buffer],  Answer) :-
        category(Word, C),
        parse([C |Stack],Buffer, Answer).

parse([Cat2,Cat1|Stack],Buffer, Answer) :-
        reduce(Cat1,Cat2, Cat3),
        parse([Cat3|Stack], Buffer, Answer).

% forward and backward application
reduce(A/B,    B,      A).
reduce(B,      A\B,    A).
```

4.2 Parse Trees

```
parse([S],[], S).

parse(Stack,[Word|Buffer],  Answer) :-
        get_category(Word, C),
        parse([C |Stack],Buffer, Answer).

parse([Cat2,Cat1|Stack],Buffer, Answer) :-
        reduce(Cat1,Cat2, Cat3),
        parse([Cat3|Stack], Buffer, Answer).

get_category(Word, C1) :-
        category(Word,C),
        C1=C:leaf(C,Word).
*/

% forward and backward application
reduce(A/B:N1,    B:N2,    A:node(A,N1,N2)).
reduce(B:N1,    A\B:N2,    A:node(A,N1,N2)).

/*
```

4.2.1 Debugging: Printing out the Parse Trees

```
parse_tree_print(T) :-!,parse_tree_print(T,0).

parse_tree_print(leaf(C,W), H) :-
        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(C), write(' '), write(W),nl,
        tab(H), write(-----),    nl.

parse_tree_print(node(F,L,R),H) :-!,
        % Calculate the height for the children to be prited at.
        H1 is H + 7,

        % print out the right branch
        parse_tree_print(R,H1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(F), write(' '), nl,
        tab(H), write(-----),    nl,

        % print out the left branch
        parse_tree_print(L,H1).
```

Chapter 5

The Fourth Chamber: Existential Quantification and Predication

In this chapter, we'll create a very small language–small enough that we can *completely* implement it so that we can:

1. Parse an input sentence which is an assertion or query.
2. If its an assertion compile it to Horn clauses.
3. If it is a query, compile it to a query against a Horn clause database.
4. Take the results of any such query and express them in English.

In particular, we want to be able to have the following conversation with the computer. We want to be able to tell it that there is a block with a particlar color:

```
a block is blue.
ok.
```

and then we want to be able to ask about it:

```
is a block blue?
yes.
```

Since it only knows about a single block which is blue, if we ask it about a red block:

```
is a block red?
no.
```

it will answer "no" when asked about this. But we want it to be able to add to its knowledge if we tell it about another block:

```
a block is red.
ok.

is a block red?
yes.
```

This is, no doubt, not the most scintilating of conversations, but the important point is that we cover *every* part of of an NLP dialog with a computer, inasmuch as we have solutions–vestigial and elementary solutions, to be sure–but solutions nontheless to all phases of a NLP read-eval-print loop. This will be a solid foundation we can use to elaborate upon.

5.1 Logical form

The logical form we'll use for the sentence:

```
A block is blue
```

will be:

```
exists(X,and(block(X),blue(X)))
```

There are several things to notice about this representation. First, we're representing a sentence of logic as a prolog term. Second, we're using prolog variables to represent variables in the logical form. This kind of muddies the waters, inasmuch as it is a mixing of meta- and object-levels, but on the whole it simplifies our task.

5.2 Enhancing the Lexicon for Semantics

```
*/
parse([S],[], S).

parse(Stack,[Word|Buffer],  Answer) :-
        get_category(Word, C:N),
        parse([C:N |Stack],Buffer, Answer).

parse([Cat2,Cat1|Stack],Buffer, Answer) :-
        reduce(Cat1,Cat2, Cat3),
        parse([Cat3|Stack], Buffer, Answer).

get_category(Word, C1) :-
        category(Word,C,_L),
        C1=C:leaf(C,Word).

/*
category(is, (s(d)/(n/n))\(np), _).
category(is, (s(q)/(n/n))/np, _).
```

```
category(block, n , X^block(X)).

category(a,np/n, P^Q^X^exists(X,and(P,Q))).

category(green, n/n, X^green(X)).
category(blue,  n/n, X^blue(X)).
```

5.3 Syntax semantics interface

We walk down the parse tree until we reach a leaf. The semantics for that leaf is stored in the categories. As we recurse back up the tree, we form the semantics for each non-leaf node by combining the semantics of the child nodes. The predicate combine_sem/3 does this combining. Basically right now it has a seperate special case for each possible pair of children which we could see. Because our grammar is small and the number of parse trees is limited, this works, but if you are thinking that this could never scale, you are right. We'll make a better way of doing it in a later chamber.

```
parse_to_semantics(leaf(C,W), C,L) :-
        category(W,C,L).

parse_to_semantics(node(C,L,R), C,L1) :-
        parse_to_semantics(L, CL,LL),
        parse_to_semantics(R, CR,RL),
        combine_sem(CL,LL, CR,RL, C,L1).


combine_sem(s(q)/(n/n)/np,_, np,L, s(q)/(n/n), L).
combine_sem(s(q)/(n/n),L2^X^L1,  (n/n),X^L2, s(q), L1).
combine_sem(np/n,L2^Q^X^L1,  n,X^L2,  np,Q^X^L1).
combine_sem(np,L1,  (s(d)/(n/n))\np, _L2,  s(d)/(n/n),L1).
combine_sem(s(d)/(n/n),L2^X^L1,  (n/n),X^L2, s(d),L1).
```

5.4 Compiling to Horn Clauses

Notice that the above grammar provides the same logical form for both moods: "A block is blue" and "is a block blue?" both get translated into "exists(X,block(X),blue(X))". The only difference is that the category of the sentence is s(d) for declarative sentences and s(q) for questions.

Unfortunately, even though the logical form is the same, *how we use* that logical form is quite different for declarative statements vs. questions. For declarative statements, we want to add to our knowledge base. For questions, we want to query our knowledge base to see if the query is true or not.

Therefore, for each of these two ways of handling the logical form, there will be two different compilers. One compiler compiles for assertion into the database, the other compile will compile for query against the database.

Compiling for Assertion

```
cnf_transform(F,Cs) :- cnf_transform(F,Cs,[]).

cnf_transform(exists(X,R), H,T) :- !,
        skolemise(X),
        cnf_transform(R,H,T).

cnf_transform(and(R,S), H,T) :-!,
        cnf_transform(R, H,I),
        cnf_transform(S, I,T).

cnf_transform(F, [axiom(F)|T],T).
*/
skolemise(X) :- var(X),!,
gensym(sk,X).

skolemise([]).
skolemise([H|T]) :-
skolemise(H),
skolemise(T).
/*
```

5.5 Compling for query

```
lt_transform(exists(_X,R), R1) :- !,
        lt_transform(R,R1).

lt_transform(and(P,Q), (P1,Q1)) :- !,
        lt_transform(P,P1),
        lt_transform(Q,Q1).

lt_transform(F,F).
```

5.6 Dialog

```
dialog :-
        % remove previous resuls
        retractall(axiom(_)),

        read_eval_loop.

read_eval_loop :-
        get_sentence(S),
        ( S=[bye,'.'] ->
            true
        ;
            process_sentence(S),
```

```
            read_eval_loop
        ).

process_sentence(S) :-

        % parse the sentence
        (  parse([], S, (s(M):N)) ->
            parse_to_semantics(N,_,L),
            process_logical_form(M,L)
        ;
             write('couldn\'t understand this:'),
             write(S),nl
        ).


% if this is an assertion, run the nute-covington transform and assert
process_logical_form(d,L) :-
    cnf_transform(L,Fs),
    assert_all(Fs).

assert_all([]).

assert_all([F|Fs]) :-
    assert(F),
    assert_all(Fs).

% if this is a query, run the lloyd-topor transform and query
process_logical_form(q,L) :-
    lt_transform(L,Q),

    (  prove(Q) ->
        write('yes'),nl
     ;
        write('no'),nl
    ).
```

5.7 Theorem Prover

```
prove((A,B)) :-
    prove(A),
    prove(B).

prove(A) :- axiom(A).
```

Chapter 6

The Fifth Chamber: Dialog and Database Update

We already know how to handle simple existential quantification, for sentences like:

```
A block is green.
```

But there are many other ways in which exitential quantification is expressed in English. The goal of this chapter is to explain how to make a program which can engage in a conversation like the following. We should be able to input sentences like:

```
There is a block.
The block is green.
There is a red bock.
```

and then the computer shoulde be able to answer questions like:

```
Is there a block?
yes.

Is there a table?
no.

Is there a blue block?
no.

Is there a red block?
yes.
```

6.1 Problems to Solve

How should we handle senteces like:

```
There is a block.
```

We might get a clue by considering a sentence like:

```
A block is there.
```

which sounds like a verbal pointing, or *ostension*. In other words, if you say "A block is *there*," you are saying, of a particular spatio-temporal region, that it is a block. For the present, we'll treat such verbal ostensions as adverbs modifying the verb "to be."

But the modifications to the grammar are relatively easy next to our next problem–that of database update.

6.2 Grammar

```
category(the,np/n, P^Q^X^exists(X,and(P,Q))).

category(there,  adv, _^true).

% there is a block.
category(is, (s(d)/np)\adv, _).

% a block is there.
category(is, (s(d)\np)/adv, _).

% is there a block?
category(is, (s(q)/adv)/np, _).
% is a block there?
category(is, (s(q)/np)/adv, _).
```

6.3 Parse Tree to Semantics

```
combine_sem(adv,X^true,s(d)/np\adv,_,s(d)/np,X^true).
combine_sem(s(d)/np,X^L1, np,L1^X^L2,s(d),L2).
combine_sem((s(d)\np)/adv,_, adv,X^true, s(d)\np,X^true).
combine_sem(np,L2^X^L1, s(d)\np,X^L2, s(d), L1).

combine_sem(s(q)/np/adv, _, adv, X^true, s(q)/np, X^true).
combine_sem(s(q)/np, X^L1, np, L1^X^L2, s(q), L2).
combine_sem(s(q)/adv/np,_, np, P^X^L, s(q)/adv, P^X^L).
combine_sem(s(q)/adv, L2^X^L1, adv,X^L2, s(q), L1).

combine_sem(n/n, X^L1, n, X^L2, n, X^and(L1,L2)).
```

6.4 Compiling to Horn Clauses

The above grammar takes a sentence like:

```
There is a block.
```

and creates the following logical form:

```
exists(X,and(block(X),true)).
```

which the NC transform of the previous chapter would translate into:

```
[axiom(block(sk1)), axiom(true)]
```

We can insert special code to ignore the axiom(true), and assert block(sk1). But then what should we do with the next sentence input by the user?

```
The block is green.
```

If we were to just follow the same proceedure, we would get another list of atoms to assert:

```
[axiom(block(sk2)), axiom(green(sk2))]
```

But this really isn't what we want at all! First of all, this seems to indicate that there are two blocks, sk1 and sk2. However, this conversation isn't talking about two blocks, its talking about one block which we're updating our information about.

So we'll have to change how we compile our logical form into Horn clauses, and we'll have to make a more sophisticated version of updating the database. As a point of departure here, we'll take our inspiration from what Quine calls the "Maxim of minimum mutilation," that is, when faced with new incommming information, a reasonable thing to do is change your current database to cover the new data, but change it as little as possible. For this example, if we already know that sk1 is a block:

```
[axiom(block(sk1))]
```

and we get the input sentence "the block is green," we would just add one more literal to our database, asserting that ski is green:

```
[axiom(block(sk1)), axiom(green(sk1))]
```

Seems easy enough–just match up the incomming info with the previous info, and add the delta. However, consider if we input another sentence:

```
There is a red block.
```

and follow the same proceedure, we'd get a database which looks like this:

```
[axiom(block(sk1)), axiom(green(sk1)), axiom(red(sk1))]
```

Which seems wrong. What went wrong is that some sentences update information about previously known objects, and some sentences tell us about the existance of new objects. So how can we tell which? Well, one clue is that (all parts of) an object can't be two colors at the same time. So we follow a broadly Thomistic maxim "whenever you find a contradiction, make a distinction." [1] Instead of interpreting the user as saying something *contradictory* about one block, we want the program to interpret the user as saying something *coherent* about two blocks.

Of course, this makes a big assumption–that the user, most of the time at least, actually speaks coherently! This assumption, (known as the "Principle of Charity,") plays an pivital role in the philosophy of Quine and Donald Davidson. The construction of this program can be taken as experimental evidence that they are correct–the principle of charity is an essential presupposition for the possiblity of communication between agents. [2]

The problem we need to solve now is this: how do we know that

```
[red(sk1), green(sk1)]
```

is contradictory? Notice that this is not of the form $(A \wedge \neg A)$, so it isn't *flatly* contradictory [3]. In order to generate the contradiction, we need to have a background theory about colors and about how they are mutually exclusive.

The kicker is that Horn clause logic really doesn't *directly* represent this sort of mutual exclusivity. The solution known to prolog folklore for this is to introduce a new predicate, "false", and have the conjunction of mutually contradictory atoms imply false, like this:

```
false :- green(X), red(X)
```

and then use negation as failure to ensure that "false" can't be derived.

6.4.1 Theorem Prover

So lets start by writing a routine to detect mutually exclusive atoms. We want a routine which will take a list of atomic formulas like:

```
[block(sk1), green(sk1), red(sk1)]
```

succeed if that list is noncontradictory, and fail if it is.

We will do this by adopting our theorem prover to be able to draw conclusions not just from asserted axioms, but also from a list of input atomic formulas. As usual, this is just an application of skeletons and techniques: the skeleton is the vanilla theorem prover, and the technique is to add another variable which is the list of atomic formlas:

[1] I got this from Richard Rorty [**?**]

[2] The philosophical ramifications of this are many and profound, and the reader is encouraged to pursue them. However, our purposes in this book are more engineering than philosophy, we'll just help ourselves to these philosophical results, acknowledge them in passing, and get back to programming.

[3] It was the realization of this that persuaded Wittgenstein that he needed to abandon the views on logical atomism he proposed in his *Tractatus* [**?**]

```
prove(true,_).

prove((A,B), R) :- prove(A, R), prove(B, R).

prove(A, R) :-
    clause((A :-B)),
    prove(B,R).

prove(A,_R) :- axiom(A).

prove(A,R) :- member(A,R).
```

then we need some world knowledge

```
*/
% colors are mutuallly exlcusive
permanent_clause((false :- green(X), red(X))).
permanent_clause((false :- green(X), blue(X))).
permanent_clause((false :- red(X), blue(X))).

% so are blockhood and tablehood
permanent_clause((false :- block(X), table(X))).
/*
```

Sharp-eyed readers can already see some efficiency problems with this technique, don't worry, we'll present a comprehensive solution later. The upside is that it is now quite easy to write our desired predicate:

```
*/
coherent(L) :- \+ prove(false,L).
/*
```

6.4.2 Updating the Database

The first modification we have to make to the tranform clauses is that we don't want to immediately skolemise the existentially quantified variable to *constants* anymore. The reason is that a formula like:

```
exists(X,block(X))
```

might be talking about a block we already know about. So instead, we just want to accumulate the existentially quantified variables, and later try to match them up, if possible, with pre-existing objects.

So using the the c_transform clause as a skeleton, we add a difference list to hold the existentially quantified variables we encounter:

```
*/
c_transform(F,Cs,Vs) :- c_transform(F,Cs,[], Vs,[]).

c_transform(exists(X,R), CH,CT, [X|VH],VT) :- !,
        c_transform(R, CH,CT, VH,VT).
```

```
c_transform(and(R,S), CH,CT, VH,VT) :-!,
        c_transform(R, CH,CI, VH,VI),
        c_transform(S, CI,CT, VI,VT).

c_transform(true, CT,CT, V,V) :- !.
c_transform(F, [F|CT],CT, V,V).
/*
```

in addition, the last clause has been modified so that it simply skips atoms of the form "axiom(true)."

6.5 The Maxim of Minimum Mutilation

For each of the new existentially-quantified variables, we have to make a decision– either it refers either to an already-known object, or it refers to a new object. To make it easy to find out which, we'll change how we store the database. In addition to asserting axioms, we'll also assert two lists. One list will be a list of constants:

```
things([sk1, sk4, sk7,..]).
```

which are the items we've already postulated. Another list will be a list of the atomic formulas we hold true of those objects:

```
atoms([block(sk1), table(sk4), green(sk4), table(sk7), ...]).
```

So the idea is that when we get in the atomic formulas of the new sentence:

```
[block(X), green(X)]
```

we'll see if we can match up the variable X with any of the pre-existing objects without generating a contradictory database. If we can't, then, following the Maxim of Thomas, we will postulate a new object for X.

```
*/
mmm_update(Vs, Os, Fs, Db1,Db3) :-
        generate_domains(Vs,Os,Fs,Db1, Doms),
        find_simultainously_consistent_assignment(Doms,Fs,Db1,Db2),
        sort(Db2,Db3).

generate_domains([], _Os,_Fs,_Db1, []).
generate_domains([V|Vs], Os,Fs,Db1, [domain(V,Pos)|Doms]) :-
     find_formulas_about_variable(Fs,V,Vfs),
     append(Vfs,Db1,F1s),
     find_possible_values(Os,V,F1s, Pos),
     generate_domains(Vs, Os,F1s,Db1, Doms).

find_formulas_about_variable([],_V,[]).
find_formulas_about_variable([F|Fs],V,R) :-
```

```
     F=..[_Functor|Args],
     ( variable_member(V,Args) ->
          R = [F|Vfs]
     ;
          R=Vfs
     ),
     find_formulas_about_variable(Fs,V,Vfs).

variable_member(V,[VT|Vs]) :-
    (
        V==VT
    ;
        variable_member(V,Vs)
    ).


find_possible_values(Os,V,Fs,Pos) :-
    find_possible_preexisting_values(Os,V,Fs,Pos1),
    ( []=Pos1 ->
        skolemise(X),
        Pos=[X]
    ;
        Pos=Pos1
    ).


find_possible_preexisting_values([],_V,_Fs, []).
find_possible_preexisting_values([O|Os],V,Fs,Pos) :-
    ( \+ \+ (V=O, coherent(Fs)) ->
          Pos = [O|Pos1]
    ;
          Pos=Pos1
    ),
    find_possible_preexisting_values(Os,V,Fs,Pos1).


find_simultainously_consistent_assignment(Doms,Fs,Db1,Db2) :-
     append(Fs,Db1,Db2),
     assignment(Doms),
     coherent(Fs).

assignment([]).
assignment([domain(X,D)|Doms]) :-
    select(X,D,_D1),
    assignment(Doms).
/*
```

6.6 Dialog

The procedure mmm_update needs to have a list of formulas we already know about, as well as a list of objects we already know about. Since we store these in the form of asserted axiom/2 predicates, we'll need some routines to assemble the list of formula and extract the known objects.

```
*/
assemble_formulas(Fs) :- findall(F,axiom(F),Fs).

extract_objects_from_formulas(Fs, Os2) :-
        extract_objects_from_formulas_aux(Fs,Os1),
        sort(Os1,Os2).

extract_objects_from_formulas_aux([],[]).
extract_objects_from_formulas_aux([F|Fs],Os3) :-
        extract_objects_from_formula(F,Os1),
        extract_objects_from_formulas_aux(Fs, Os2),
        append(Os1,Os2,Os3).

extract_objects_from_formula(F,Os) :- F=..[_|Os].
/*
```

now we can write the dialog function:

```
*/
dialog :-
        % remove previous resuls
        retractall(axiom(_)),

        read_eval_loop.

read_eval_loop :-
        get_sentence(S),
        ( S=[bye,'.'] ->
            true
        ;
            process_sentence(S),
            read_eval_loop
        ).

process_sentence(S) :-
        % parse the sentence
        (  parse([], S, (s(M):N)) ->
            parse_to_semantics(N,L),
            process_logical_form(M,L)
        ;
             write('couldn\'t understand this:'),
             write(S),nl
        ).
```

```
% if this is an assertion, run the nute-covington transform and assert
process_logical_form(d,L) :-
        c_transform(L,Fs,Vs),
        assemble_formulas(Db1),
        retractall(axiom(_)),
        extract_objects_from_formulas(Db1,Os),
        mmm_update(Vs,Os,Fs,Db1,Db2),
        assert_all_as_axioms(Db2).

assert_all_as_axioms([]).
assert_all_as_axioms([F|Fs]) :-
        assert(axiom(F)),
        assert_all_as_axioms(Fs).

% if this is a query, run the lloyd-topor transform and query
process_logical_form(q,L) :-
        lt_transform(L,Q),

        writeln(prove(Q,[])),

        (  prove(Q,[]) ->
            writeln('yes')
        ;
            writeln('no')
        ).
/*
```

Chapter 7

The Sixth Chamber: Conjunction

What's one and one and one and one and one and one and one?
–The Red Queen, in *Alice Through the Looking Glass*

Conjunction is quite amazingly hard; so hard, in fact, that there are entire NLP texts which just don't treat conjunction at all (see [**?**] for example). Lets explores several reasons why conjunction is just so hard.

7.1 Changes to the Parser

What does "and" do? Here are some grammatical sentences which use "and":

```
There is a red block and there is a green block.
There is a red and a green block.
There is a red and green block.
```

and here are some ungrammatical sentences which use and:

```
There and is a red block. *
There is a and red a green block. *
```

looks like "and" can combine together any two subconstitutents as long as they are of the same type. What consituent type does "and" create? Using the contextual method:

```
There is [a red block].
There is [a red and a green block].

There is a [red] block.
There is a [red and green] block.
```

it looks like it creates a new constituent which is of the same type as the constituents which it combines. So lets try adding "and" to our lexicon with the following type:

```
category(and, (X\X/X), _).
```

(we'll talk about the semantics later, so we've left that slot blank).

```
category(and, (X\X/X), _).
```

7.2 Why we can't use unification

Jack and Jill went up the hill.

Unification a very efficient way to assemble terms, but–as is usually the case–it is efficient because it is strictly limited in power. We need a more general method of describing how parts of things come together to form a whole.

7.3 Lambda Calculus

Lambda calculus is an alternate language, which is Turing-complete. It provides a much more convenient method, however, of describing how parts should be stitched together to form a whole. Its amazing how far we can get with such a simple idea.

7.4 Developing Alpha Conversion and Beta Conversion using Skeletons and Techniques

This parser is the skeleton we use:

7.4.1 Syntax of Lambda Expressions

```
*/

lambda_expression(E) :- var(E),!.

lambda_expression(F*A) :- !,
        lambda_expression_application(F,A).

lambda_expression(X^F) :-!,
        var(X),
        lambda_abstraction(X,F).

lambda_expression(E) :-
        E =.. [_F|Ts],
        map_lambda_expression(Ts).
```

```
lambda_expression_application(F,A) :-
        lambda_expression(F),
        lambda_expression(A).

lambda_abstraction(_X,F) :-
        lambda_expression(F).

map_lambda_expression([]).
map_lambda_expression([T|Ts]) :-
        lambda_expression(T),
        map_lambda_expression(Ts).

/*
```

7.4.2 Bound Variables

To the above skeleton, we apply a techique. The technique is to recognize when a binding expression has been encountered, and save the variable which has been bound in a bag.

So, the first thing we do is think about the bag we want to store the bound variables in. We'll put them in a list, but if we want to test if a variable is a member off that list, we can't just use member/2, because member/2 will unify. So we write our own member-esque function which uses == instead of unification:

```
*/

variable_in_bag(X1,[X2|_]) :- X1 == X2.
variable_in_bag(X,[_|T]) :- variable_in_bag(X,T).

/*
```

Now for the visitor itself. We rename "lambdaExpression" to "visitBound-Variables" and add an extra argument which holds, at any point in the lambda expression, a list of the variables which are bound.

```
*/

visit_bound_variables(E, Bs) :- var(E),!,
        write(E), write(' is '),
        ( variable_in_bag(E,Bs) ->
            writeln(' bound')
        ;
            writeln(' free')
        ).

visit_bound_variables(F*A, Bs) :- !,
        visit_bound_variables_application(F,A, Bs).
```

```
visit_bound_variables(X^F, Bs) :-!,
        var(X),
        visit_bound_variable_abstraction(X,F,Bs).

visit_bound_variables(E, Bs) :-
        E =.. [_F|Ts],
        map_visit_bound_variables(Ts, Bs).

visit_bound_variables_application(F,A, Bs) :-!,
        visit_bound_variables(F, Bs),
        visit_bound_variables(A, Bs).

% notice that when a variable binder is encountered, it packs
% the variable on top of stack of bound variables
visit_bound_variable_abstraction(X,F,Bs) :-
        visit_bound_variables(F, [X|Bs]).

map_visit_bound_variables([], _Bs).
map_visit_bound_variables([T|Ts], Bs) :-
        visit_bound_variables(T, Bs),
        map_visit_bound_variables(Ts, Bs).

/*
```

7.4.3 Alpha Conversion

The bound variable visitor can in turn be used as a skeleton. Alpha conversion is the process of changing the names of the bound variables.

The first step is to make a data structure which we can use to associate one variable with another. This is also derived using skeletons and techniques; the skeleton being the variable_in_bag/2 procedure above. Instead of just containing variables, the bag will contain elements of the form sub(X,Y), which indicates that Y should be substituted for X.

```
*/

substitute_bound_variable(sub(X1,Y),[sub(X2,Y)|_]) :- X1 == X2.
substitute_bound_variable(X,[_|T]) :- substitute_bound_variable(X,T).

/*
```

We then add another parameter to the visit_bound_variables predicate–an output parameter. The alpha conversion process then works as a transducer, walking over the input lambda expression, substituting a fresh variable for every bound variable, and passing everything else along unchnaged to the output parameter:

```
*/
```

```
alpha_convert(F1, F2) :- alpha_convert(F1,[], F2).

alpha_convert(X1, Bs, X2) :- var(X1),!,
        ( substitute_bound_variable(sub(X1,X2),Bs) ->
            true
        ;
            X2=X1
        ).

alpha_convert(F1*A1, Bs, F2*A2) :- !,
        alpha_convert_application(F1,A1, Bs, F2,A2).

alpha_convert(X1^F1, Bs, X2^F2) :-!,
        var(X1),
        alpha_convert_abstraction(X1,F1, Bs, X2,F2).

alpha_convert(F1, Bs, F2) :-
        F1 =.. [Op|Ts1],
        map_alpha_convert(Ts1, Bs, Ts2),
        F2 =.. [Op|Ts2].

alpha_convert_application(F1,A1, Bs, F2,A2) :-
        alpha_convert(F1,Bs,F2),
        alpha_convert(A1,Bs,A2).

alpha_convert_abstraction(X1,F1, Bs, X2,F2) :-
        alpha_convert(F1, [sub(X1,X2)|Bs], F2).

map_alpha_convert([],_,[]).
map_alpha_convert([T1|Ts1], Bs, [T2|Ts2]) :-
        alpha_convert(T1,Bs,T2),
        map_alpha_convert(Ts1,Bs,Ts2).

/*
```

7.4.4 Free Variables

Free variables are variables appearing in a lambda expression which arn't bound. If we want to collect these, the easiest way is to also collect the bound variables, and whenever we run across a variable, check to see if it is a bound variable. If its not, then its a free variable, so we collect it.

So it would be plausible to use the boundVariable routine as a skeleton. However, there's another consideration. Consider the formula:

```
foo(X,X^bar(X))
```

Should our predicate indicate that X is free or bound? Its free in some contexts, and its bound in others.

The easiest (and safest) way out of this is to punt: Since the names of bound variables are arbitrary, why not just use alpha conversion to rename them? That way, we're guarunteed that none of the variables reported as free are bound in any context.

We could perform alpha conversion before collecting the free variables, but then we'd have to make two passes over the input lambda term. This extra pass can be avoided simply by choosing as a skeleton the alpha conversion predicate.

The technique is then to add a difference-list to the predicate to collect the free variables. The reason it is Perierran instead of fregian is that the returned list of free variables is the appendage of the lists from the subterms, so to get rid of the overhad of appending we use different lists.

```
*/

free_variables(F1, Us, F2) :- free_variables(F1, [], [],Us, F2).

free_variables(X1, Bs, U1,U2, X2) :- var(X1),!,
        ( substitute_bound_variable(sub(X1,X2),Bs) ->
            U1=U2
        ;
            X2=X1,
            U2=[X1|U1]
        ).

free_variables(F1*A1, Bs, U1,U2, F2*A2) :- !,
        free_variables_in_application(F1,A1, Bs, U1,U2, F2,A2).

free_variables(X1^F1, Bs, U1,U2, X2^F2) :-!,
        var(X1),
        free_variables_in_abstraction(X1,F1, Bs, U1,U2, X2,F2).

free_variables(F1, Bs, U1,U2, F2) :-
        F1 =.. [Op|Ts1],
        map_free_variables(Ts1, Bs, U1,U2, Ts2),
        F2 =.. [Op|Ts2].

free_variables_in_application(F1,A1, Bs, U1,U3, F2,A2) :-
        free_variables(F1, Bs, U1,U2, F2),
        free_variables(A1, Bs, U2,U3, A2).

free_variables_in_abstraction(X1,F1, Bs, U1,U2, X2,F2) :-
        free_variables(F1, [sub(X1,X2)|Bs], U1,U2, F2).

map_free_variables([], _, U,U, []).
map_free_variables([T1|Ts1], Bs, U1,U3, [T2|Ts2]) :-
        free_variables(T1, Bs, U1,U2, T2),
        map_free_variables(Ts1, Bs, U2,U3, Ts2).

/*
```

7.4.5 Alphabetic Variants

```
*/

alphabetic_variants(A1,B1) :-
        % grab the free variables in both the terms
        free_variables(A1,FA,A2),
        free_variables(B1,FB,B2),

        % see if the free variables are the same
        FA == FB,

        % instantiate all of the variables.
        % note: we cant stipulate that N1 is the same as
        % N2, because if A2 and B2 share free variables,
        % they will be instantiated during the first numervars
        % call and the second will indicate a different number of
        % variables...
        numbervars(A2,0,_N1),
        numbervars(B2,0,_N2),

        % see if the resulting terms are the same
        A2=B2.

/*
```

7.4.6 Beta Conversion

Beta conversion is the process of applying beta reduction over and over again until there's no more beta conversion which can be done.

```
*/
% driver
beta_convert(X,X) :- var(X),!.

beta_convert(X*A1,X*A3) :- var(X),!,
        alpha_convert(A1,A2),
        beta_convert(A2,A3).

beta_convert(X^F*A, R) :- !,
        alpha_convert(X^F*A,CF),
        beta_convert_application(CF,R).

beta_convert(E1, E3) :-
        E1 =.. [F|Ts1],
        map_beta_convert(Ts1, Ts2),
        E2 =..[F|Ts2],
        ( \+ \+ E1=E2
        -> E3=E2
        ; beta_convert(E2, E3)
```

```
        ).

beta_convert_application(X^F*A1, X^F*A2) :- var(F),X\==F,!,
        beta_convert(A1,A2).

beta_convert_application(X^F1*X, F2) :-
        beta_convert(F1,F2).

map_beta_convert([], []).
map_beta_convert([T|Ts], [R|Rs]) :-
        beta_convert(T, R),
        map_beta_convert(Ts, Rs).

/*
```

sectionAlpha and Beta Conversion Test Suite
Adopted from Blackburn and Bos, who adopted and extended from Milward.

```
*/
expression(1,P^(P*mia)*(X^walk(X)),
           walk(mia)).

expression(2,some(X,and(man(X),(P^some(X,and(woman(X),P*X)))*(Y^love(X,Y)))),
           some(X,and(man(X),some(Y,and(woman(Y),love(X,Y))))) ).

% Simple reduction
expression(3,(A^sleep(A))*mia,
           sleep(mia)).

% Reduction to another function:
expression(4,A^B^like(B,A)*mia,
           C^like(C,mia)).

% Reduction of inner expression leaving outer function:
expression(5,A^(B^like(B,A)*vincent),
           (C^like(vincent,C))).

% Reduction of inner expression leaving outer function, with var y shared.
expression(6,A^and(((B^like(B,A))*vincent),sleep(A)),
           (C^and(like(vincent,C),sleep(C)))).

% Reduction twice:
expression(7,A^B^like(B,A)*mia*vincent,
           like(vincent,mia)).

% Reductions in a nested position
expression(8,p((A^sleep(A))*vincent),
           p(sleep(vincent))).

% Reductions inside a variable predicate
```

```
expression(9,(A^(A*((B^sleep(B))*vincent))),
           (A^(A*sleep(vincent)))).

% No reductions possible
expression(10,(A^(A*sleep(vincent))),
           (A^(A*sleep(vincent)))).

% Nested reductions (apply to a function which must be applied again)
expression(11,((A^(A*vincent))*(B^sleep(B))),
           sleep(vincent)).

expression(12,((A^believe(mia,(A*vincent)))*(B^sleep(B))),
           believe(mia,sleep(vincent))).

% Copied functions
expression(13,((A^and((A*vincent),(A*mia)))*(B^sleep(B))),
           and(sleep(vincent),sleep(mia))).

expression(14,(((A^(B^and(((C^(C*(A*vincent)))*(D^probably(D))),
                          ((C^(C*(B*mia)))*(D^improbably(D))))))
               *(E^walk(E))) *
              (E^talk(E))),
           and(probably(walk(vincent)),improbably(talk(mia)))).

% Double application.
expression(15,(((A^(B^((C^((C*A)*B))*(D^(E^love(D,E))))))*jules)*mia),
           love(jules,mia)).

% Two functions with the same use of variable.
expression(16,(((A^(B^some(C,and((A*C),(B*C)))))*(D^boxer(D)))*(D^sleep(D))),
           some(C,and(boxer(C),sleep(C)))).

% Test for correctly dealing with the same variable name occurring twice
% (loops without alpha-conversion)
expression(17,(A^(_*A)) * (C^(A^like(A,C))),
           (_*(E^(F^like(F,E))))).

expression(18,((A^(B^(A*B)))*(C^(B^like(B,C)))),
           (D^(E^like(E,D)))).

% Test for correctly performing alpha conversion
expression(19,(((A^(B^((C^((C*A)*B))*(B^(A^love(B,A))))))*jules)*mia),
           love(jules,mia)).

expression(195, A^B^(C^(C*A)*B)* (B^A^love(B,A)) *jules*mia,
           love(jules,mia)).

% Test for correctly performing alpha conversion
expression(20,(A^(B^((C^((C*A)*B))*(B^(A^love(B,A)))))),
           (D^(E^love(D,E)))).
```

```
% Further alpha conversion testing
expression(21,(((A^(B^and(some(B,(A*B)),(A*B))))*(C^boxer(C)))*vincent),
          and(some(B,boxer(B)),boxer(vincent))).

% reduction of an application to an expression
expression(22,foo((X^X)*b)*a,foo(b)*a).
expression(23,foo((X^X)*b)*a*c*d*e,foo(b)*a*c*d*e).

expression(24,X*Y*X*Z, X*Y*X*Z).
expression(25,a*b*c*d, a*b*c*d).

/*
```

7.5 Changes to the Syntax/Sematics Interface

```
*/

parse_to_semantics(L,S2) :-
        parse_to_semantics(L,_,S1),
        beta_convert(S1,S2).

parse_to_semantics(leaf(C,W), C,L) :-
        category(W,C,L).


parse_to_semantics(node(C,N), C, P^(P*L)) :-!,
        parse_to_semantics(N, _,L).


parse_to_semantics(node(C,L,R), C,L1) :-
        parse_to_semantics(L, CL,LL),
        parse_to_semantics(R, CR,RL),
        combine_sem(CL,LL, CR,RL, C,L1).

% forward and backward application
combine_sem(A/B, L1, B,L2,  A,L1*L2).
combine_sem(B,L1,  A\B,L2,  A,L2*L1).

:-op(400, yfx, \\).
:-op(400, yfx, //).


combine_sem(A//B, L1, B,L2,  A,L1*L2).
combine_sem(B,L1,  A\\B,L2,  A,L2*L1).

/*
```

7.6 Changes to the Lexicon

```
*/
category(the, np/n,  P^Q^exists(X,and(P*X,Q*X))).
category(a,   np/n,  P^Q^exists(X,and(P*X,Q*X))).

category(there,  adv, _^true).

% there is a block.
category(is, (s(d)/np)\adv, Q^P^(P*Q)).

% is green
category(is, (s(d)/adj)\(np), P^Q^(P*Q)).
category(is, (s(q)/adj)/np,   P^Q^(P*Q)).

% a block is there.
category(is, (s(d)\np)/adv, P^Q^(Q*P)).

% is there a block?
category(is, (s(q)/adv)/np, P^Q^(P*Q)).

% is a block there?
category(is, (s(q)/np)/adv, P^Q^(Q*P)).

category(block, n , X^block(X)).
category(table, n , X^table(X)).

category(green, adj, X^green(X)).
category(red,   adj, X^red(X)).
category(blue,  adj, X^blue(X)).

category(green, (n/n), P^(X^and(green(X),P*X))).
category(red,  (n/n), P^(X^and(red(X),P*X))).
category(blue,  (n/n), P^(X^and(blue(X),P*X))).


%category(and, (adv\adv/adv), P^Q^Y^(and(P*Y,Q*Y))).
%category(and, (np\np/np), P^Q^Y^(and(P*Y,Q*Y))).
%category(and, (n\n/n), P^Q^Y^(and(P*Y,Q*Y))).
%category(and, ((X\Z)\(X\Z)/(X\Z)), P^Q^Y^(and(P*Y,Q*Y))).
%category(and, ((X/Z)\(X/Z)/(X/Z)), P^Q^Y^(and(P*Y,Q*Y))).
%category(and, (s(M)\s(M)/s(M)), P^Q^(and(P,Q))).

/*
```

7.7 debugging

```
*/
```

```
parse_tree_print(T) :-!,parse_tree_print(T,0).

parse_tree_print(leaf(C,W), H) :-
        parse_to_semantics(leaf(C,W),_C,Sem), beta_convert(Sem,Sem1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(C), write(' '), write(W), write(:), write(Sem1),nl,
        tab(H), write(-----),    nl.


parse_tree_print(node(C,N),H) :-!,
        parse_to_semantics(node(C,N),_C,Sem), beta_convert(Sem,Sem1),
        % Calculate the height for the children to be prited at.
        H1 is H + 7,

        % print out the kid
        parse_tree_print(N,H1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(C), write(' '), write(:), write(Sem1), nl,
        tab(H), write(-----),  nl,

        tab(H1), write([]),nl.

parse_tree_print(node(F,L,R),H) :-!,
        parse_to_semantics(node(F,L,R), _C,Sem), beta_convert(Sem,Sem1),

        % Calculate the height for the children to be prited at.
        H1 is H + 7,

        % print out the right branch
        parse_tree_print(R,H1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(F), write(' '), write(:), write(Sem1), nl,
        tab(H), write(-----),    nl,

        % print out the left branch
        parse_tree_print(L,H1).
/*
```

7.8 dialog

Chapter 8

The Seventh Chamber: Disjunction

We want the language to be able to handle sentences like this:

```
The block is green or red.
The table is blue.
```

and then ask quesitons like

```
Is the table blue?
Is the block red?
Is the block green or red?
```

in other words, we want to be able to handle disjunction.

8.1 Grammar

Note, we'll be rewinding a bit here, because in this chapter we won't start from a language which includes conjunction. As will soon be obvious, disjunction introduces enough wrinkles of its own, and focusing exlucively on disjunction will allow us to more directly address those particular problems. The complexifications resulting from the interactions between conjunction and disjunction will be the task of the next chapter.

We won't be rewinding all the way, however. Basically, the grammar shown is exactlylike the grammar of the previous chamber, but only with entries for "and" changed to "or":

```
*/

category(or, (adj\adj/adj), P^Q^Y^(or(P*Y,Q*Y))).
category(or, (adv\adv/adv), P^Q^Y^(or(P*Y,Q*Y))).
category(or, (np\np/np), P^Q^Y^(or(P*Y,Q*Y))).
```

```
category(or, ((n/n)\(n/n)/(n/n)), P^Q^R^Y^(or(P*R*Y,Q*R*Y))).
category(or, (n\n/n), P^Q^Y^(or(P*Y,Q*Y))).
category(or, ( s(M) \ s(M)/ s(M)), P^Q^(or(P,Q))).
/*
```

8.2 The problem of representing disjunctive information in prolog

By convention, a prolog program describes one world, its least Herbrand model. But what about a theory which contains disjunctions? E.g. suppose we want to represent the following model, We know there is a block:

```
block(sk1)
```

and we know it is either red or green, but we don't know which:

```
or(red(sk1),
   green(sk1))
```

if we restrict the above theory to just one least Herbrand model, we really would not have captured our knowledge that "block" could have been either red or green. We also would not have captured our ignorance as to which one it is.

So what we'll do is represent the alternative possibilities as alternate possible worlds. Each world will still be a least Herbrand model for some Horn clause theory.

For example, for the example formula above, we would consider *two* worlds as being possible, and represent each world with a seperate Herbrand Model. The first model would be:

```
block(sk1).  red(sk1).
```

and the second would be

```
block(sk1).  green(block1).
```

So how should we actually realize this in a prolog program? We could take our cue from, say, situation calculus, and relativize each predicate to a particular world. For the example above, if we reify the two worlds (calling them, say, "w1" and "w2"), we could represent them as a single Horn clause theory:

```
formula(block,[sk1,w1]).  formula(red,[sk1,w1]).
formula(block,[sk1,w2]).  formula(green,[block1,w2]).
```

The problem with this method of representation is that it is rather prolix–e.g. notice that the blookhood of sk1 has to be explicitly restated for each possible world.

Instead, lets factor out the facts common to all the worlds we consider possible, and explicitly represent the multiplicity *only* for facts which are disjunctive.

We'll use the following metaphor. Consider the block which is either red or green. Consider a cosmic switch which has two positions. When the switch is in the first position, the block is red. When the switch is in the second position, the block is green. Our ignorance of whether the block is red or green therefore is our ignorance as to the position the switch is in.

In general, disjunctions can have more than two possibilities, so instead of using binary switches, we'll use dials. A dial has a pointer which can be set any of several discrete positions, numbered 1 through n. Suppose that 'sk_dial' is the name of a dial, and we want to represent the fact that it can be in one of n positions. Then we can represent the fact that sk_dial's pointer is in position number 2, with the following prolog fact:

```
dial(sk_dial,2).
```

For every fact or clause which is only true when sk_dial is in position 2, we add "dial(sk_dial,2)" as a condition for it. So say we want the axiom "formula(foo,[])" and the clause "formula(bar,[]) :- formula(foo,[])" to both be true only when k_dial is in position 2. We then use the facts:

```
formula(foo,[]) :- dial(sk_1,2).
formula(bar,[]) :- dial(sk_1,2)), formula(foo,[]).
```

So consider our blocks world example. Lets make a dial for the block called 'block1_dial'. We'll represent the fact that if the dial is in position 1, the block is red and if the dial is in position 2, the block is green, with the following prolog rules:

```
axiom(block(block1)).
formula(green,[sk1])  :-  dial(block1_dial,1).
formula(red,[sk1])    :-  dial(block1_dial,2).
```

We have achieved a more compact representation, but the price is we will have to complicate our inference engine a bit. The only fact we can derive from this prolog program (aside from the possible values for the dial) is that block1 is a block. We'll have to augment prolog's inferential capability in order to be able to conclude things like "the block *might* be green."

8.3 Inference Engine

We have to add several capabilities to the standard prolog theorem proving process. We'll do this by creating several metainterpreters which will work in concernt to implement the additional functionality we need.

We'll start out with the standard, vanilla, metainterpreter: augmented for some extra logical constants. In the metainterpreters, we will use the standard infix punctuations for the logical constants (";" for *mand*, ";" for *or*) and we'll use "ot()" for*not*.

```
prove(X) :- axiom(X).
prove((X,Y)) :- prove(X), prove(Y).
prove((X;Y)) :- prove(X).
prove((X;Y)) :- prove(Y).
prove(not(X)) :- \+ prove X.
```

The first thing we'll want to do, of course, is tell the metainterpreter which dial settings we are using, so it knows which Horn clause theory it should be using to reason with. We do this by taking the standard vanilla metainterpreter, and adding to it an additional clause to hold atomic formula which serve as extra premises. Although we can pass in any ground atomic formula, the ones which are of interest will be ones of the form "D=V" where D is the name of a dial, and V is a number 1 through n which tells which position the dial is in.

```
*/
prove_in_least_herbrand_model(A,_R) :- axiom(A).

prove_in_least_herbrand_model((A;_B),R) :-
        prove_in_least_herbrand_model(A,R).
prove_in_least_herbrand_model((_A;B),R) :-
        prove_in_least_herbrand_model(B,R).

prove_in_least_herbrand_model((A,B),R) :-
        prove_in_least_herbrand_model(A,R),
        prove_in_least_herbrand_model(B,R).

prove_in_least_herbrand_model((A),R) :-
        clause((A:-B)),
        prove_in_least_herbrand_model(B,R).

prove_in_least_herbrand_model(not(A),R) :-
        \+ prove_in_least_herbrand_model(A,R).

prove_in_least_herbrand_model(dial(D,V),R) :-
        ground(D),
        member(D=V,R).
/*
```

So now, if we happen to know which dial settings we should be using, we can prove the elements of the least Herbrand model which are apropos to that Horn clause theory.

Now, what if we don't know what world we're in–what if we just want to know whether there *is* a possible world where what we want to prove is true In other words, we want to know whether what we are trying to prove is possibly true. In this case, we want the computer to figure out whether there is a possible combination of dial setings which would make what we want to prove to be true. So to do this we use the technique of abduction. We add two extra arguments to each clause of the vanilla metainterpreter. These argments function as a

difference list, and accumulate the various dial settings we need to prove our input formula. This clause is used to answer "maybe" type queries.

The first few clauses are just a straighforward elaboration:

```
*/
prove(A,R,R) :- axiom(A).

prove((A;_B),R1,R2) :-
        prove(A,R1,R2).
prove((_A;B),R1,R2) :-
        prove(B,R1,R2).

prove((A,B),R1,R3) :-
        prove(A,R1,R2),
        prove(B,R2,R3).

prove((A),R1,R2) :-
        clause((A:-B)),
        prove(B,R1,R2).

prove(not(A),R,R) :-
        \+ prove(A,R,_).
/*
```

This clause has the salient changes: to prove an atom of the form dial/2, we first check to see if one such is already in the residue. If not, then we see if the prolog database has such an atom asserted. If so, we add it to the residue.

```
*/

prove(dial(D,V),R1,R2) :-
        ground(D),
        ( member(D=_,R1) ->
            R2=R1
        ;
            dial(D,V),
            R2=[D=V|R1]
        ).
/*
```

Finally, with the above tools in hand, we can implement the main prove predicate. The first few clauses are, once again, just standard vanilla:

```
*/
prove(A)      :- axiom(A).
prove(A)      :- clause((A:-B)), prove(B).
prove((A,B))  :- prove(A),prove(B).
prove((A;_B)) :- prove(A).
prove((_A;B)) :- prove(B).
prove(not(A)) :- \+ prove(A).
/*
```

But the next clause is complex. Suppose we know that either a or b is true:

```
or(a,b).
```

and that a or b implies c:

```
implies(or(a,b), c)
```

It is apparent that c is a consequence of this theory. But so far, our prover couldn't prove it. Why not? Well, recall we would represent the above theory as two least herbrand models, one for which a is true, and one for which b is true:

```
Least Herbrand Model #1:          Least Herbrand Model #2:
a.                                b.
c :-                              c :-
     a                                 a
;    b.                           ;    b.
```

In each of these least Herbrand Models, we could prove c. But, we would have to put in the residue a term indicate which least Herbrand model this was being proved in, The kicker is, that if we can prove it in ALL least Herbrand models, that means that don't have remember the least Herbrand models we proved it in. The reason is that if we can prove something in all least Herbrand models, then it is unconditional–we can prove it no matter what.

So that's what this final clause does. If we can't prove A any other way, we first try to prove it in one least Herbrand model. Then, we in effect, try to prove it in all other least Herbrand models. If we can prove it in all other least Herbrand models, we consider it unconditionally proven.

```
*/
prove(A) :-
        copy_term(A,A1),
        prove(A1,[],Rs),
        get_complementary_dial_settings(Rs,Rss),
        prove_in_least_herbrand_models(Rss,A).

/*
```

Now this sounds amazingly inefficient, and it is. But we use a trick to speed things up. We don't actually have to prove A with *every* possible combination of dials in order to prove that it is true in all least Herbrand models.

```
*/
get_complementary_dial_settings(Rs,Rss) :-
        sort(Rs,Rs1),
        get_dials(Rs1,Ds),
        possible_scenarios_for_dials(Ds,Rss1),
        select(Rs1,Rss1,Rss).

prove_in_least_herbrand_models([],_A).
```

```
prove_in_least_herbrand_models([Rs|Rss],A) :-
        copy_term(A,A1),
        prove_in_least_herbrand_model(A1,Rs),
        prove_in_least_herbrand_models(Rss,A).


get_dials(Rs,Ds) :-
        get_dials_from_scenarios(Rs,Ds1),
        sort(Ds1,Ds).


get_dials_from_scenarios([],[]).
get_dials_from_scenarios([D=_|Rs],[D|Ds]) :-
        get_dials_from_scenarios(Rs,Ds).


possible_scenarios_for_dials(Ds,Rss) :-
        possible_assignments_for_dials(Ds,As),
        findall(Rs,possible_scenario_for_dials(As,Rs), Rss).

possible_scenario_for_dials([],[]).
possible_scenario_for_dials([D|Ds],[A|As]) :-
        select(A,D,_),
        possible_scenario_for_dials(Ds,As).


possible_assignments_for_dials([],[]).
possible_assignments_for_dials([D|Ds],[Rs|Rss]) :-
        possible_assignments_for_dial(D,Rs),
        possible_assignments_for_dials(Ds,Rss).

possible_assignments_for_dial(D,Rs) :- ground(D), findall(D=V,dial(D,V),Rs).

/*
%----------   testing ----------

% all red and green blocks are made of wood.
clause((wooden(X) :- red(X))).
clause((wooden(X) :- green(X))).

% all blocks are either red or green.
clause((red(X) :- block(X), dial(sk(X),1))).
clause((green(X) :- block(X), dial(sk(X),2))).

dial(sk(X),1) :- ground(X), \+ member(X,[sk1]).
dial(sk(X),2) :- ground(X), \+ member(X,[sk1]).

axiom(block(sk1)).
axiom(red(sk1)).
axiom(block(sk2)).
```

8.4 Lloyd-Topor Transform

```
*/

lt_transform(exists(_X,R), R1) :- !,
        lt_transform(R,R1).

lt_transform(and(true,Q), Q1) :- !,
        lt_transform(Q,Q1).

lt_transform(and(P,true), P1) :- !,
        lt_transform(P,P1).

lt_transform(and(P,Q), (P1,Q1)) :- !,
        lt_transform(P,P1),
        lt_transform(Q,Q1).

lt_transform(or(P,Q), (P1 ; Q1)) :- !,
        lt_transform(P,P1),
        lt_transform(Q,Q1).

lt_transform(F,F).
/*
```

8.5 Compiling

```
*/

compile -->
       remove_exists([],Vs),
       { skolemise(Vs) },
       remove_trues,
       compute_dials.

remove_exists(V1,V3, and(A1,B1),and(A2,B2)) :-!,
       remove_exists(V1,V2, A1,A2),
       remove_exists(V2,V3, B1,B2).

remove_exists(Vi,V3, or(A1,B1), or(A2,B2))  :-!,
       remove_exists(Vi,V2, A1,A2),
       remove_exists(V2,V3, B1,B2).

remove_exists(V1,V2, exists(X,F1), F2) :-!,
       remove_exists([X|V1],V2, F1,F2).

remove_exists(V,V, F,F).
```

```
remove_trues(exists(X,F1), exists(X,F2)) :-!,
       remove_trues(F1,F2).

remove_trues(and(A1,B1), F) :-!,
       remove_trues(A1,A2),
       remove_trues(B1,B2),

       ( A2=true
       -> F=B2
       ; ( B2=true
         -> F=A2
         ; F=and(A2,B2)
         )
       ).

remove_trues(or(A1,B1), F) :-!,
        remove_trues(A1,A2),
        remove_trues(B1,B2),

        ( A2=true
        -> F=true
        ;  ( B2=true
           -> F=true
           ; F=or(A2,B2)
           )
        ).

remove_trues(true, true) :-!.

remove_trues(F,F).


compute_dials(F1,F4) :-
        compute_dials(F1, [] ,F3,[]),
        convert_dial_lists_to_clauses(F3,F4).

compute_dials(and(F1,F2), S,  O1,O3) :-!,
        compute_dials(F1, S, O1,O2),
        compute_dials(F2, S, O2,O3).

compute_dials(or(F1,F2), S,  [dial(D,1),dial(D,2)|O1],O3) :-!,
        skolemise(D),
        compute_dials(F1, [dial(D,1)|S], O1,O2),
        compute_dials(F2, [dial(D,2)|S], O2,O3).

compute_dials(F, S, [F-S|O],O).
```

```
convert_dial_lists_to_clauses([],[]).
convert_dial_lists_to_clauses([F1|F1s],[F2|F2s]) :-
        convert_dial_list_to_clause(F1,F2),
        convert_dial_lists_to_clauses(F1s,F2s).

convert_dial_list_to_clause(A-[],A).
convert_dial_list_to_clause(A-[DH|DT],(A:-B)) :-
        convert_dial_list_to_body([DH|DT],B).
convert_dial_list_to_clause(dial(D,V), dial(D,V)).


convert_dial_list_to_body([D],D).
convert_dial_list_to_body([DH,DT],(DH,DT)).
convert_dial_list_to_body([DH,DT,T|Ts],(DH,B)) :-
        convert_dial_list_to_body([DT,T|Ts],B).


retract_program :-
     retractall(axiom(_)),
     retractall(clause(_)).

assert_program([]).
assert_program([dial(D,V)|Cs]) :-!,
        assert(dial(D,V)),
        assert_program(Cs).

assert_program([C|Cs]) :-
       ( C = (_:-_)
       ->  assert(clause(C))
       ;   assert(axiom(C))
       ),

       assert_program(Cs).
/*
```

8.6 Maxim of Minimum Mutilation

Think about the boggling potential inefficiencies inherent in reasoning disjunctivly! If you have to search through all combinations of n dials, there are 2^n different combinations!

Clearly, we need to mitigate against this as much as possible. One of the ways in which we do this is to always prefer models which have fewer dials. A prime example of how we can do this is during update. For example, consider the conversation:

```
There is a green block or a red table.
There is a red table.
```

After processing the first sentence, we have added an additional dial in order to account for the disjunct. But suppose we interpret the second sentence as being a clarification of the first sentence–saying that perhaps once we were unsure of whether it was a table or a block which exists, but now we know that it is a table and not a block. This would allow us to remove the dial from the database.

Of course, there are times when this isn't the proper thing to do–the person really is talking about yet another table. Communication isn't always perfect. But it does tend to be self correcting: human language tends to be very redundant, so even if we make a mistake, we have a fighting chance of getting further clarifications.

Under what conditions should we consider two noun phrases to be refering to the same object, and under what conditions should we consider two noun phrase to be refering to different objects? Here's what we'll do:

Consider the sentence:

```
There is a block.
```

We've described an object as being a block, but this leaves open many possibilities. It is still ambiguous what color the block is. It is open for further clarification. But there are some ways in which we can't elaborate on this object. For example, we can't say that it is a dog.

```
There is a blue table or a red block.
There is a block.
```

So what we are going to do is say that if something is entirely within the cone of clarification of one of the disjuncts, and it is incompatible with the other disjuncts, we can prune the other disjuncts away. This in accordance with the intuition that we wouldn't have mentioned a block–and left vague exactly what color it is–if it weren't distinct from the object we were trying to describe in the first sentence?

When asked "how many blocks are there?" what should we answer? "At least one," or "One or two," or "I don't know." The question is do these possibilities multiply or do they start to converge? It depends, but if we are describing a particular, concrete situation, it should converge, or so we hope. Eventually the program should also be able to ask clarifying questions.

Perhaps we should keep a few models around. That's probably the best solution. Our best guess would be that there is just one block, and it is red. Seems like these alternatives should be different kinds of worlds. They should perhaps follow a modal logic of knowledge a la Reiter.

```
*/

mmm_update(NVs,NFs) :-
% get the pre-existing formulas
findall(A, axiom(A), As),
findall(C, clause(C), Cs),
append(As,Cs, F1s),
```

```
% get the pre-existing objects
extract_objects_from_formulas(F1s, Os1),

% make new objects for the new variables
copy_term(NVs, Os2),
skolemise(Os2),

% make a binding and figure out if it is ok
assign_variables_in_occam_order(NVs,Os1,Os2),
( consistent_binding(NFs, NFs1)
        -> % go with the first one we find, because it
   % will have the least number of objects

   % remove previous program; assert new one.
   % (remember, we may have altered the original
   % program)
   retract_program,
   assert_program(NFs1)
         ;
           % let everybody know this didn't work
           fail
         ).

/*
```

This generates assignments for variables. It first tries to satisefy the formula only with pre-existing objects. If it can't do that, it will keep adding new objects as necessary.

```
*/

assign_variables_in_occam_order(Vs,O1,[]) :-
assign_variables(Vs,O1).
assign_variables_in_occam_order(Vs,O1,[O|O2]) :-
( assign_variables(Vs,O1)
          ;
  assign_variables_in_occam_order(Vs,[O|O1],O2)
        ).


assign_variables([],_O).
assign_variables([V|Vs],O1) :-
select(V, O1,O2),
assign_variables(Vs,O2).

/*
```

This runs through, sees if we can prove a contradiction, if we can, it tries to resolve the contradiction by kicking out dial value.

So how is this for a strategy:

1. For each dial
2. for each dial value
3. not prove(false,[dial(dial,value)],_R)
4. if that suceeds, we keep the dial value.
5. if that fails, we prune the dial value.

```
*/

consistent_binding(NFs1, NFs2) :-
% find all of the worlds in which this is contradictory
findall(Ds, find_inconsistant_dials(NFs1, Ds), Dss),

% if there are no inconsistant branches, great!
% we don't have to do anything
( Dss=[]
        -> NFs2=NFs1,!
        ;   % there are inconsistant branches.  If this is
    % inconsistant under all branches, then there's no
    % hope, just fail
    ( member([],Dss)
    -> fail
    ;  % see if we can fix it
       prune_inconsistant_dial_values(Dss, NFs1,NFs2)
    )
        ).

% the name of this function only makes sense if
% we stipulate that a subset of dials specifies a
% cyllnder set, i.e. if it returns an empty
% list, then all dial combinations are contradictory
find_inconsistant_dials(NFs1, Ds) :-
prove_from_list(false,NFs1, [],Ds).

prove_from_list(A,R, D,D) :- member(A,R).

prove_from_list((A;_B),R, D1,D2) :-
        prove_from_list(A,R, D1,D2).

prove_from_list((_A;B), R, D1,D2) :-
        prove_from_list(B, R,  D1,D2).

prove_from_list((A,B),R, D1,D3) :-
        prove_from_list(A,R, D1,D2),
        prove_from_list(B,R, D2,D3).
```

```
prove_from_list((A),R, D1,D2) :-
        permanent_clause((A:-B)),
        prove_from_listprove(B,R, D1,D2).

prove_from_list(dial(D,V),R, D1,D2) :-
        ground(D),
        ( member(D=_,D1) ->
            D2=D1
        ;
            member(dial(D,V),R),
            D2=[D=V|D1]
        ).

/*
% we have to eliminate at least one dial value from
% each of the ways to fail.  We have to do this without
% getting rid of the other stuff
prune_inconsistant_dial_values(Dss, NFs1,NFs2) :-
findall(L-S, possible_inconsistent_dial_fix(Dss, NFs1, S,L), Ss1),
keysort([Fix|Ss2]),
fix_inconsistent_dials(Fix,NFs1,NFs2).

possible_inconsistent_dial_fix(Dss, NFs1, S,L) :-
select_dials(Dss, Ss).

select_dials([], [],[]).
select_dials([S|Ss], [H1|T1],[H2|T2]) :-
select(S,H1,H2),
select_dials(Ss, T1,T2).


*/

remove_dial_info(_D, [], [], [],[]).

remove_dial_info(D, [dial(D,V)|Dvs], Cs, [dial(D,V)|L1],L2) :-!,
        remove_dial_info(D, Dvs, Cs, L1,L2).

remove_dial_info(D, Dvs, [H:-dial(D,V)|Cs], [H:-dial(D,V)|L1],L2) :-!,
        remove_dial_info(D, Dvs, Cs, L1,L2).

remove_dial_info(D, Dvs, Cs, [L|L1],[L|L2]) :-!,
        remove_dial_info(D, Dvs, Cs, L1,L2).

/*
```

Chapter 9

The Eight Chamber: Conjunction and Disjunction

9.1 Syntax

It is quite easy to enhance the lexicon to support both conjunction and disjunction; just take the grammar from the previous chapter, cut and paste the lexicon entries for "or" and search and replace "or" with "and":

```
*/
category(and, (adj\adj/adj), P^Q^Y^(and(P*Y,Q*Y))).
category(and, (adv\adv/adv), P^Q^Y^(and(P*Y,Q*Y))).
category(and, (np\np/np), P^Q^Y^(and(P*Y,Q*Y))).
category(and, ((n/n)\(n/n)/(n/n)), P^Q^R^Y^(and(P*R*Y,Q*R*Y))).
category(and, (n\n/n), P^Q^Y^(and(P*Y,Q*Y))).
category(and, ( s(M) \ s(M)/ s(M)), P^Q^(and(P,Q))).
/*
```

So why devote a whole chamber to just the combination of conjunction and disjunction? Well, because this is the first time that ambiguity starts to play a material role.

Amazingly enough, for the grammars we have so far, most of the sentences we've typed only have one parse tree. As we will see, this is unfortunately not the norm for CCG-based parsers–they are notorious for being almost spectacularly ambiguous. In this chapter, we'll start to develop coping mechanisms for this unfortunate reality.

Why is this a problem? After all, so far sentences like:

```
The block is red or green.
```

have had only one parse. But what happens when we add another disjunct:

```
The block is red or green or blue.
```

Already, even the grammar of the last chamger will give two different parse trees, which translate into two different logical forms:

```
exists(X, and(block(X), or(or(blue(X), red(X)), green(X))))
exists(X, and(block(X), or(blue(X), or(red(X), green(X)))))
```

This of course, make no difference in terms of what formulas follow from this formula, so even though the sentence is syntactically ambiguous, the syntactic ambiguity does not lead to semantic ambiguity. But, watch what happens if we have both conjunction and disjunction available:

```
exists(X, and(block(X), or(and(blue(X), red(X)), green(X))))
exists(X, and(block(X), and(blue(X), or(red(X), green(X)))))
```

These two logical forms are clearly not equivalent, so the syntactic ambiguity echos in the semantics ambiguity.

9.2 Semantics

The mmm_update proceedures of the previous chambers all made the assumption that we never had to go back and revisit a decision. Clearly that's not the case. Consider the following conversation:

```
There is a blue table and a green block or red block.
```

suppose we compile it down into the following program:

```
table(t). blue(t).
block(b).
```

Chapter 10

The Ninth Chamber: Universal Quantification

10.1 Hole Semantics

Hole semantics is a comprehensive, yet simple and elegant, solution to the problem of scope phenomena in natural languages.

Ok, the key to hole semantics is this: your parent in the search tree tells you what your label is. If you are creating a label for a kid, you have to create an existential quantifier for it as well. Same for holes.

Leaf nodes are going to be told what their position in the parse tree and what their scope is is by their parents. So they are functions which take a hole argument and a label argument. In addition, they might be quantified over, as in the case of nouns. For example:

```
adjective(X^H^L^[L:formula(red,   [X]), L=<H])   -->  [red].
```

Non-leaf nodes (also called *combining nodes*) have to do everything which leaf nodes do, plus more. Just like leaf nodes, they have to be functions which take a scope and a label which their parent will assign to them. But in addition to that, they have to create new labels which name the positions which their children are going to be in. For example:

```
conjunction(P^Q^X^H^L^L1^L2^[label(L1), label(L2),
                              L:and(L1,L2), P*X*H*L1, Q*X*H*L2,L=<H]) -->
      [and].

*/

prepair_store(R0,R4) :-
        strip_extra_abstractions(R0,R1),
        flatten(R1,R2),
        sort(R2,R3),
```

```
        write('store='), writeln(R3),
        process_store(R3,R4).


% we use this trick to force the renaming of the label variables for
% subtrees which get duplicated during
strip_extra_abstractions([],[]) :-!.
strip_extra_abstractions(_^F1,F2) :- !,
        strip_extra_abstractions(F1,F2).
strip_extra_abstractions([H1|T1],[H2|T2]) :-!,
        strip_extra_abstractions(H1,H2),
        strip_extra_abstractions(T1,T2).
strip_extra_abstractions(F,F).


process_store(S,Fs) :-
        seperate_store(S, Hs,Ls,As,Ds),
        skolemize_labels(Ls,0),
        possible_values_for_holes(As, Ls, Vs),
        possible_plugging(Hs,Vs),
        satisfactory_plugging(As,Ds),
        retree_formulas(As,Fs).

/*
```

The store has mixed together all of the holes, labels, constraints, and formula in one list. So the first item of business is to seperate them out into different buckets:

```
*/
seperate_store([], [],[],[],[]).
seperate_store([hole(X)|T], [X|H], L, A, D) :-
        seperate_store(T, H,L,A,D).

seperate_store([label(X)|T], H, [X|L], A, D) :-
        seperate_store(T, H,L,A,D).

seperate_store([X:Y1|T], H, L, [X:Y2|A], D) :-
        term_list(Y2,Y1),
        seperate_store(T, H,L,A,D).

seperate_store([X=<Y|T], H, L, A, [X=<Y|D]) :-
        seperate_store(T, H,L,A,D).

/*
```

Notice that while we were seperating out the formula, we flatten them into a more convenient form. They are now converted into lists. The first element of the list will be either "formula" if it is a predicate, or it will be a logical constant. This way we can switch on the first element of the list to know what to do with

the rest, and the arguments are easier to manipulate. Of course, eventually, we'll want to convert them back into the standard form, so the term_list/2 predicate is written so it can convert back and forth between these two representations.

```
*/
term_list([formula,F,Args], P)  :-  P=..[formula,F,Args].
term_list([and|Args], P)        :-  P=..[and|Args].
term_list([and1|Args], P)       :-  P=..[and1|Args].
term_list([and2|Args], P)       :-  P=..[and2|Args].
term_list([and3|Args], P)       :-  P=..[and3|Args].
term_list([or|Args], P)         :-  P=..[or|Args].
term_list([exists|Args], P)     :-  P=..[exists|Args].
term_list([forall|Args], P)     :-  P=..[forall|Args].
term_list([not|Args], P)        :-  P=..[not|Args].
term_list([implies|Args], P)    :-  P=..[implies|Args].
term_list([true], P)            :-  P=..[true].

/*
```

This goes through and replaces the unbound variables for the labels with ascending terms of the form sk(X). The cool thing is that since this doesn't depend on some sort of gensym-ish deal, the numbering is deterministic and repeatable.

```
*/
skolemize_labels([],_N).
skolemize_labels([skl(N)|T],N) :-
        N1 is N+1,
        skolemize_labels(T,N1).
/*
```

Only some of the labels which are in the store are good for plugging the holes. Most of them are used only to express the tree structure.

So the following precates determine which labels are actually up for grabs. It goes through all of the flattened formulas and finds, the labels, and only returns the labels which haven't been used yet.

```
*/
possible_values_for_holes(As, Ls1, Ls2) :-
        used_labels(As,[],Uls1),
        remove_variables(Uls1,Uls2),
        sort(Uls2,Uls3),
        filter_labels(Uls3,Ls1,Ls2).

used_labels([], Uls,Uls).
used_labels([_L:[_F|Args]|As],Uls1,Uls3) :-
        extract_labels(Args,Uls1,Uls2),
        used_labels(As,Uls2,Uls3).

% there can be logs of stuff in the argument (constant, variables, etc)
```

```
% so this extracts just the labels and appends them to an incomming list.
extract_labels([], Ls,Ls).
extract_labels([H|T], Ls1,Ls3) :-
        (   var(H)
        ->  Ls2=Ls1
        ;   ( H=skl(_)
            -> Ls2 = [H|Ls1]
            ; Ls2 = Ls1
            )
        ),
        extract_labels(T, Ls2,Ls3).



remove_variables([],[]).
remove_variables([H|T1],R) :-
        (   ground(H)
        ->  R=[H|T2]
        ;   R=T2
        ),
        remove_variables(T1,T2).

filter_labels(_Uls, [], []).
filter_labels(Uls, [H|Ls1], Ls3) :-
        (   member(H,Uls)
        ->  Ls3 = Ls2
        ;   Ls3 = [H|Ls2]
        ),
        filter_labels(Uls, Ls1, Ls2).


possible_plugging(Hs,Vs1) :-
        permutation(Vs1,Vs2),
        bind_holes(Hs,Vs2).

bind_holes([],[]).
bind_holes([H|T1],[H|T2]) :-
        bind_holes(T1,T2).

/*

*/
satisfactory_plugging(As,Ds) :-
        extract_dominance(As,Das),
        transitive_closure(Das,Das_star),
        satisfies_constraints(Ds,Das_star).
/*
```

    puts it into form ugraphs c

```
*/
```

```
extract_dominance([], []).
extract_dominance([L:[_F|Args1]|As], [L-Args2|Das]) :-
        remove_variables(Args1,Args2),
        extract_dominance(As,Das).
/*

*/
satisfies_constraints(Ds,Das_Star) :-
        is_tree(Das_Star),
        statisfies_dominance_constraints(Ds, Das_Star).


is_tree([]).
is_tree([N-Ns|T]) :-
        \+ member(N,Ns),
        is_tree(T).
/*

*/
statisfies_dominance_constraints(Ds,Das_Star) :-
        forall(member(D,Ds), satisfies_dominance_constraint(D, Das_Star)).

satisfies_dominance_constraint(X1=<X2,Das_Star) :-
        (   X1=X2
        ;   member(X2-E,Das_Star),
            member(X1,E)
        ).
/*
```

Now, we have to unflatten the formulas in order to reconstruct a formula which is in standard form. The general idea is to go through and replace each label with a variable, keeping track of which label corresponds with the variable. Then, we plug in the term which is labels with the label for each variable.

```
*/

retree_formulas(As,Tree) :-
        replace_labels_with_variables(As, Fs, [],VLs),
        assemble_tree(Fs,VLs,Tree).

replace_labels_with_variables([], [], VLs,VLs).
replace_labels_with_variables([L:A|As], [L:F2|Fs], VLs1,VLS3) :-
        replace_formula_labels_with_variables(A,F1,VLs1, VLs2),
        term_list(F1,F2),
        replace_labels_with_variables(As, Fs, VLs2,VLS3).

replace_formula_labels_with_variables([],[], VLs,VLs).
replace_formula_labels_with_variables([A|As],[F|Fs],VLs1, VLs3) :-
        (   var(A)
        ->  F=A,
```

```
            VLs2=VLs1
        ;   (   A=skl(X)
            ->  VLs2 = [skl(X)-F|VLs1]
            ;   F=A,
                VLs2=VLs1
            )
        ),
        replace_formula_labels_with_variables(As,Fs,VLs2, VLs3).

assemble_tree([],_Vs,_T).
assemble_tree([L:F|LFs],Vs,T) :-
        (   member(L-V,Vs)
        ->  V=F
        ;   T=F  % the root is the only label which isn't in Vs.
        ),
        assemble_tree(LFs,Vs,T).

/*
```

10.2 lexicon

```
% is green
category(is, (s\np)/(n/n) , Q^(P^(P*(Q*(_X^H^L^[L:formula(true,[]), L=<H])))))).

% is on the table
category(is, (s\np)/(n\n) , Q^(P^(P*(Q*(_X^H^L^[L:formula(true,[]), L=<H])))))).

% is a block
category(is, (s/np)\np , P^(Q^(Q*(X^(P*(Y^H^L^[L:formula(equal,[Y,X]), L=<H])))))).

% intransitive
category(slept, s\np , P^(P*(X^H^L^[L:formula(sleep,[X]), L=<H]))).

% transitive
category(loves, (s/np)\np , P^(Q^(Q*(X^(P*(Y^H^L^[L:formula(loves,[Y,X]), L=<H])))))).
category(hates, (s/np)\np , P^(Q^(Q*(X^(P*(Y^H^L^[L:formula(hates,[Y,X]), L=<H])))))).

% events
category(stabs, (s/np)\np , P^(Q^(Q*(X^(P*(Y^H^L^L1^L2^L3^L4^L5^[label(L1), label(L2), label(L3)
                                            label(L4), label(L5),
                                            L:exists(Z,L1),
                                            L1:and(L2,L3), L2:formula(stab,[Z]),
                                            L3:and(L4,L5), L4:formula(subject,[Z,Y]),
                                                           L5:formula(object,[Z,X]),
                                            L=<H])))))).


category(john, np , P^(P*john)).
category(fred, np , P^(P*fred)).
```

```
category(there, np , Q^H^L^H1^L1^L2^L3^[hole(H1),label(L1),label(L2),label(L3),
                                        L2:exists(X,L3), L3:and(L1,H1),
                                        L=<H1, L2=<H,
                                        L1:formula(true, []), L1=<H,Q*X*H*L]).

category(man,   n , X^H^L^[L:formula(man,  [X]), L=<H]).
category(woman, n , X^H^L^[L:formula(woman,[X]), L=<H]).
category(dog,   n , X^H^L^[L:formula(dog,  [X]), L=<H]).
category(block, n , X^H^L^[L:formula(block,[X]), L=<H]).
category(table, n , X^H^L^[L:formula(table,[X]), L=<H]).
category(back,  n , X^H^L^[L:formula(back, [X]), L=<H]).
category(knife, n , X^H^L^[L:formula(knife,[X]), L=<H]).

category(green, n/n , P^X^H^L^L1^L2^[label(L1),label(L2), L:and(L1,L2), L1:formula(green,[X]), P*X*H*L2, L=<
category(red,   n/n , P^X^H^L^L1^L2^[label(L1),label(L2), L:and(L1,L2), L1:formula(red,  [X]), P*X*H*L2, L=<

category(on, (n\n)/np , Q^P^X^H^L^H2^L2^L3^[hole(H2),label(L2),label(L3),
                                    L:and(L2,H2),
                                    P*X*H*L2,
                                    Q*(Y^H1^L1^[L1:formula(on,[X,Y]), L1=<H1])*H*L3,
                                    L=<H,
                                    L3=<H2]).

category(with, (n\n)/np , Q^P^X^H^L^H2^L2^L3^[hole(H2),label(L2),label(L3),
                                        L:and(L2,H2),
                                        P*X*H*L2,
                                        Q*(Y^H1^L1^[L1:formula(with,[X,Y]), L1=<H1])*H*L3,
                                        L=<H,
                                        L3=<H2]).


category(every, np/n , P^Q^H^L^H1^L1^L2^L3^[hole(H1),label(L1),label(L2),label(L3),
                                        L2:forall(X,L3), L3:implies(L1,H1),
                                        L=<H1, L2=<H,
                                        P*X*H*L1, Q*X*H*L]).

category(a, np/n , P^Q^H^L^H1^L1^L2^L3^[hole(H1),label(L1),label(L2),label(L3),
                            L2:exists(X,L3), L3:and(L1,H1),
                            L=<H1, L2=<H,
                            P*X*H*L1,Q*X*H*L]).

category(the, np/n , P^Q^H^L^H1^L1^L2^L3^[hole(H1),label(L1),label(L2),label(L3),
                            L2:exists(X,L3), L3:and(L1,H1),
                            L=<H1, L2=<H,
                            P*X*H*L1,Q*X*H*L]).



category(and, (X\\X//X) , P^Q^H^L^H1^H2^L1^L2^[hole(H1),hole(H2),
```

```
                                                label(L1),label(L2),
                                                L:and(H1,H2),
                                                P*H1*L1, Q*H2*L2,
                                                L1=<H1, L2=<H2,
                                                L=<H]).
category(and, (X\\X//X) , P^Q^Y^H^L^H1^H2^L1^L2^[hole(H1),hole(H2),
                                                label(L1),label(L2),
                                                L:and(H1,H2),
                                                P*Y*H1*L1, Q*Y*H2*L2,
                                                L1=<H1, L2=<H2,
                                                L=<H]).


sentence(S,F) :-
        parse([],S, s:P),
        parse_tree_print(P),
        parse_to_semantics(P,_,Sem),
        beta_convert([hole(H),label(L), Sem*H*L], F2),
        prepair_store(F2,F).


% Syntax semantics interface

parse_to_semantics(leaf(C,W), C,L) :-
        category(W,C,L).


parse_to_semantics(node(C,N), C, P^(P*L)) :-!,
        parse_to_semantics(N, _,L).


parse_to_semantics(node(C,L,R), C,L1) :-
        parse_to_semantics(L, CL,LL),
        parse_to_semantics(R, CR,RL),
        combine_sem(CL,LL, CR,RL, C,L1).

% for conjunction/coordination
combine_sem(A//B,L1,   B,L2,  A,L1*L2).
combine_sem(B,L1,  A\\B,L2,  A,L2*L1).


% forward and backward composition
combine_sem(A/B,L1,  B/C,L2,  A/C, Z^(L1*(L2*Z))).
combine_sem(B\C,L2,  A\B,L1,  A\C, Z^(L1*(L2*Z))).

% forward and backward application
reduce(A//B:N1,    B:N2,    A:node(A,N1,N2)).
reduce(B:N1,    A\\B:N2,    A:node(A,N1,N2)).
```

```
% print the parse tree

parse_tree_print(T) :-!,parse_tree_print(T,0).

parse_tree_print(leaf(C,W), H) :-
        parse_to_semantics(leaf(C,W),_C,Sem), beta_convert(Sem,Sem1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(C), write(' '), write(W), write(:), write(Sem1),nl,
        tab(H), write(-----),    nl.


parse_tree_print(node(C,N),H) :-!,
        parse_to_semantics(node(C,N),_C,Sem), beta_convert(Sem,Sem1),
        % Calculate the height for the children to be prited at.
        H1 is H + 7,

        % print out the kid
        parse_tree_print(N,H1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(C), write(' '), write(:), write(Sem1), nl,
        tab(H), write(-----),  nl,

        tab(H1), writeln([]).

parse_tree_print(node(F,L,R),H) :-!,
        parse_to_semantics(node(F,L,R), _C,Sem), beta_convert(Sem,Sem1),

        % Calculate the height for the children to be prited at.
        H1 is H + 7,

        % print out the right branch
        parse_tree_print(R,H1),

        % now print out this node
        tab(H), write(-----),    nl,
        tab(H), write(F), write(' '), write(:), write(Sem1), nl,
        tab(H), write(-----),    nl,

        % print out the left branch
        parse_tree_print(L,H1).
```

Chapter 11

The Tenth Chamber: Negation

Chapter 12

The Eleventh Chamber: Equality and Functions

Chapter 13

The Twelfth Chamber: Prepositional Phrases

Chapter 14

The Thirteenth Chamber: Building a better NP

Chapter 15

The Fourteenth Chamber: Building a better VP

Chapter 16

The Fifteenth Chamber: Neo-Davidsonian Events

Chapter 17

The Sixteenth Chamber: Cause, Effect, and The Frame Problem

Chapter 18

The Seventeenth Chamber: The Subjunctive Mood

Modal knowledge. With each of the combinations of switches, we associate a set of worlds. e.g.

```
a or (pos[b] and c)

prove:  pos (a or b)

pos(a or b)

nec (a or b)

not pos not(a or b)

not pos not a
not pos not b

-----------------

dials examples

pos a

dial_values(sk1,[1,2]).
a :- dial(sk1, 1).
-----------------------

a or pos b

dial_values(sk1,[1,2]).
```

```
a :- dials(sk1,1).
pos b  :- dials(sk1,2).

dial_values(sk2,[1,2]).
b :- dial(sk1,2), dial(sk2,1).

-----------------------

pos (b or c).

dial_values(sk1,[1,2,3]).
b:-dial(sk1,1).
c:-dial(sk1,2).

-----------------------
```

Chapter 19

The Eigthteenth Chamber: Tense

Chapter 20

The Nineteenth Chamber: Pronouns and Anaphora

Chapter 21

The Twentyeth Chamber: Numbers and Plurals

Chapter 22

The Twenty-firstChamber: Spatial Reasoning

Chapter 23

The Twenty-second Chamber: Planning

Chapter 24

The Twenty-third Chamber: Belief

Chapter 25

The Twenty-fourth Chamber: Multiple Agents

Chapter 26

The Twenty-fifth Chamber: Degrees of Belief

Chapter 27

The Twenty-sixth Chamber: Presupposition

Chapter 28

The Twenty-seventh Chamber: Conversational Implicature

Chapter 29

The Twenty-eighth Chamber: Natural Lanugae Generation

Chapter 30

The Twenty-eighth Chamber: Dialog

Chapter 31

The Twenty-ninth Chamber:

Chapter 32

The Thirtyeth Chamber:

Chapter 33

The Thirty-first Chamber:

Chapter 34

The Thirty-second Chamber:

Chapter 35

The Thirty-third Chamber:

Chapter 36

The Thirty-fourth Chamber:

Chapter 37

The Thirty-fifth Chamber:

Chapter 38

The Thirty-sixth Chamber:

Bibliography

[1] The Fracas Consortium, Robin Cooper, et al. Describing the Approaches The Fracas Project deliverable D8 Available online, just google for it... December 1994

[2] The Fracas Consortium, Robin Cooper, et al. The State of the Art in Computational Semantics: Evaluating the Descriptive Capabilities of Semantic Theories The Fracas Project deliverable D9 Available online, just google for it... December 1994

[3] William Ralph (Bill) Keller. Nested Cooper storage: The proper treatment of lquantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors Natural Language parsing and Linguistic Theories pages 432-445 Reidel, Dordrecht, 1988.

[4] Lakhotia, A. Incorporating Programming Techniques into Prolog Programs. Proc. 1989 North American Conference on Logic Programming eds. Lusk, E. and Overbeek, R. pp. 4260449, MIT Press, 1989

[5] Larson, R.K. (1985). On the syntax of disjunction scope. Natural Language and Linguistic Theory 3:217, 265

[6] Naish, Lee Higher-order logic programming in Prolog Tecnnical Report 96/2 Department of Computer Science University of Melborne, Prakville, Victoria 3052, Australia. 1996

[7] Naish, Lee, and Leon Sterling "Stepwise enhancement and higher-order programming in prolog." Journal of Functional and Logic Programming 4 (2000): 2000.

[8] Patrick Blackburn and Johan Bos Computational Semantics for Natural Language Course Notes for NASSLLI 2003 Indiana University, June 17-21, 2003 Available on the web, just google for it....

[9] Mitchell P. Marcus, Beatrice Santorini and Mary ann Marciniewicz. Building a large annotated corpus of English: the Penn Treebank. in Computational Linguistics, vol. 19, 1993.

[10] Terrence Parsons Events in the semantics of English MIT press, Cambridge, Ma. 1990

[11] Fernando C.N. Pereira and Stuart M. Shieber Prolog and Natural Language analysis CSLI lecture notes no. 10 Stanford, CA 1987