The 36 Chambers of SHRDLU

Randall A. Helzerman

April 19, 2009

```
*/

:-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?

> *There is a crack in everything*
> *that's how the light gets in*
> – Leonard Cohen.

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, I'm afraid, 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. Well begun is half done.

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

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 its exact identity 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 Relations

To use prolog, you need to learn to use a new language, whimically called "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.

> 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
```

Here is wisdom: 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.

2.4.1 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 only has 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 has 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 all kinds of 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.

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.

2.5 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
?-
```

2.6 Rules

Now we know how to find grandfathers. There is a problem, however. Its kind of a bummer 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 are durn near the last prolog constructs you will need to learn. 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.

2.7 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.

2.8 Meta Interpreters

2.9 Useful Prolog Warts

2.9.1 Testing For Groundness

2.9.2 Self-Modifying Code

An example of how to use this is creating a new constant. This is used during skolemization. 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)).
```

2.9.3 Testing whether variables are unified to each other

In prolog, X == Y is true iff X and Y the same variable, or that one is an alias of the other.

2.9.4 Double Negation in order not to bind variables trick

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 [6] 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:

1. Those which belong to the language
2. 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 start out with pre-theoretical intuition as to which lists of words are grammatical and which aren't. For example, these lists:

```
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 lists these:

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

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

So where does this intuition come from? Well, we do understand English, and therefore some lists of words sound "OK" to us, and some sound funny.

If this doesn't sound very scientific, well, its not, or at least, not yet. As Alen Perlis says, "You can't go from informal to formal by formal means." Our initial intuitions about which lists of words 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," [1] then we have to find the joints of the sentences, the places where the sentences can be broken 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 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 list of words. 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 part could fit together with other parts 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]
```

[1] Actually, this was Plato [9]

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 parts. 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 noticed that we can substitue [A table] for [A block] in any sentence and get another sentece:

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

We’ve called this category ”NP”.

We also notice that within ”A block” we can substitute “block” for “table” in any sentence and still get a 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(table, n).
/*

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

Now we can define the lexicon entries for the other words:

```
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).
```

Then we have the parser:

```
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).
/*
```

now we can type in something like:

```
?- parse([],[is,the,block,green],C:T), parse_tree_print(T).
```

and it will print out:

```
        -----
        n/n green
        -----
-----
s(q)
```

```
-----
                     -----
                     n block
                     -----
              -----
              np
              -----
                     -----
                     np/n the
                     -----
       -----
       s(q)/ (n/n)
       -----
              -----
              s(q)/ (n/n)/np is
              -----
C = s(q),
T = node(s(q), node(s(q)/ (n/n), leaf(s(q)/ (n/n)/np, is),
node(np, leaf(np/n, the), leaf(n, block))), leaf(n/n, green)) ;
false.
```

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

First, for each of the words, we give its sense:

```
*/
sense(block, n , X^block(X)).

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

sense(green, n/n, X^green(X)).
sense(red,  n/n, X^blue(X)).

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

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 sense predicate. 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) :-
        sense(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).
/*
```

If the following looks hacked and ad hoc, it is. We essentially just enumerate various ways in which these

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

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(a, np/n).
category(the, np/n).
category(there, adv).

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

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

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

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

6.3 Parse Tree to Semantics

```
*/
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, P^X^L1,  s(d)\np,X^P,  s(d),L1).
combine_sem((s(d)\np)/(n/n), _,  (n/n), X^L2, (s(d)\np),X^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).
/*
```

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 compiler 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')
        ).
/*
```

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 ofquantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors Natural Language parsing and Linguistic Theories pages 432-445 Reidel, Dordrecht, 1988.

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

[5] 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....

[6] 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.

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

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

[9] Aristocles, son of Ariston *Phaedrus* Academy, Athens, Greece 400 B.C.