

The 36 Chambers of SHRDLU

Randall A. Helzerman

September 23, 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

1	Introduction	1
1.1	Why all these chambers??	2
1.2	Why are your programs half-baked?	2
1.3	Why are you using Prolog instead of (insert favorite language here)	3
1.4	Why don't you make the programs downloadable?	4
2	The First Chamber: Prolog	5
2.1	Variables	5
2.2	Read-Eval-Print Loop	6
2.3	Unification	7
2.4	Relations	8
2.4.1	Some cautions	9
2.5	The Grandfather Clause	11
2.6	Rules	12
2.7	Counting and Recursion	12
2.8	Meta Interpreters	13
2.9	Useful Prolog Warts	13
2.9.1	Testing For Groundness	13
2.9.2	Self-Modifying Code	13
2.9.3	Testing whether variables are unified to each other	13
2.9.4	Double Negation in order not to bind variables trick	14
	Bibliography	14

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 arranged 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 anything 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 approaching SHRDLU's ability to understand complex sentences. What happened?

This book is an exploration 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 succession of ever cleverer CURTs gives the reader a smooth on-ramp which greatly eases the burden of understanding.

This book follows a similar pattern. We get a very small, but complete system working as quickly as possible, and then each succeeding 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 they can be summarized like this: It is said that a good programming language should have two virtues:

1. It should make 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 fares much better on the second virtue, and in addition, it has a third virtue uniquely its own, which most other programming languages *don't* have:

1. It makes (otherwise) impossible tasks possible.

In this regard, 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 language 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 gonna 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 programming 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 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 chemistry how molecules are made out of atoms). For example, we want to represent a burrito, made out of a tortilla, beans, shredded cheese, and salsa:

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

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

So, to rehearse: 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,shredded_cheese,salsa).
```

```
X = burrito(tortilla,beans,shredded_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 uses 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, whimsically 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, let's go over to the *other* genealogy 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 let's 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, let’s 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, everything 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 actually 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 occasionally, 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 possible 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 misspelled "Matthan", and so we'll just stick with the Genealogy 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 contradictory information, the better programmer (or theologian) 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 second 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 separate 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 separate what you want to put in the list by commas, 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 succeeds 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

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] 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