First Order Logic and Induction in First order Logic

Source : UCLA notes on predicated calculus

From Natural Language to First Order Logic

In this exercise, it is best not to worry about details of tense and larger concerns with consistent ontologies and so on. The main point is to understand connectives and quantifiers and the use of predicates, functions, constants, and equality.

a. Some students took French in spring 2001

- A given student, course and semester are our constants.
- *French:* specific French course (one could also interpret these sentences as referring to *any* such course, in which case one could use a predicate $Subject(c, f)$ meaning that the subject of course c is field f
- Student(x) : predicate satisfied by members of the category -> x is a student
- $Takes(x, c, s)$: student *x* takes course *c* in semester *s*

$$\exists x\; Student(x) \land Takes(x, French, Spring2001)$$

b. Every student who takes French passes it.

- $Passes(x, c, s)$: student *x* passes course *c* in semester *s*

$$\forall x, s\; Student(x) \land Takes(x, French, s) \Rightarrow Passes(x, French, s)$$

From Natural Language to First Order Logic

In this exercise, it is best not to worry about details of tense and larger concerns with consistent ontologies and so on. The main point is to understand connectives and quantifiers and the use of predicates, functions, constants, and equality.

c. Only one student took Greek in spring 2001.

- *Greek:* specific Greek course (what we respectively defined for French)

$\exists x\ Student(x) \land Takes(x, Greek, Spring2001) \land \forall y\ Student(y) \land (y \neq x) \Rightarrow$
$\neg Takes(y, Greek, Spring2001).$

d. The best score in Greek is always higher than the best score in French

- $Score(x, c, s)$: the score obtained by student x in course c in semester s
- $x > y$: *x* is greater than *y*

$$\forall s\ \exists x\ \forall y\ Score(x, Greek, s) > Score(y, French, s)$$

Φυσικοί αριθμοί

Αξιώματα του Peano:

- *ΦυσικόςΑριθμός*(0)
- $\forall\, n\ \ ΦυσικόςΑριθμός(n) \Rightarrow ΦυσικόςΑριθμός(S(n))$
- $\forall\, n\ \ 0 \neq S(n)$.
- $\forall\, m, n\ \ m \neq n \Rightarrow S(m) \neq S(n)$.
- $\forall\, m\ \ ΦυσικόςΑριθμός(m) \Rightarrow +(0, m) = m$.
- $\forall\, m, n\ \ ΦυσικόςΑριθμός(m) \wedge ΦυσικόςΑριθμός(n) \Rightarrow$
  $+(S(m), n) = S(+(m, n))$

Το τελευταίο αξίωμα, με συντακτικό καλλωπισμό, γράφεται:

- $\forall\, m, n\ \ ΦυσικόςΑριθμός(m) \wedge ΦυσικόςΑριθμός(n) \Rightarrow (m+1) + n = (m+n) +1$

Exercise : Proof using Peano Axioms

Write down a sentence asserting that + is a commutative function.
Does your sentence follow from the Peano axioms?
If so, explain why; if not, give a model in which the axioms are true and your sentence is false.

Exercise : Proof using Peano Axioms

Write down a sentence asserting that + is a commutative function.
Does your sentence follow from the Peano axioms?
If so, explain why; if not, give a model in which the axioms are true and your sentence is false.

$\forall x, y\ (x + y) = (y + x)$
This does follow from the Peano axioms.
Roughly speaking, the definition of + says that $x + y = S^x(y) = S^{x+y}(0)$, *where* S^x *is shorthand for the* S *function applied* x *times.*
Similarly, $y + x = S^y(x) = S^{y+x}(0)$.
Hence, the axioms imply that $x + y$ *and* $y + x$ *are equal to syntactically identical expressions.*
This argument can be turned into a formal proof by induction

Steps to convert a sentence to clause form

1. Eliminate all ⇔ connectives by replacing
   each instance of the form (P ⇔ Q) by expression ((P ⇒ Q) ^ (Q ⇒ P))

2. Eliminate all ⇒ connectives by replacing
   each instance of the form (P ⇒ Q) by (¬P v Q)

3. Reduce the scope of each negation symbol to a single predicate
   by applying equivalences such as converting
   ¬¬P to P
   ¬(P v Q) to ¬P ^ ¬Q
   ¬(P ^ Q) to ¬P v ¬Q
   ¬(∀x)P to (∃x) ¬P
   ¬(∃x)P to (∀x) ¬P

4. Standardize variables:
   - rename all variables so that each quantifier has its own unique variable name. For example, convert (∀x)P(x) to (∀y)P(y) if there is another place where variable x is already used.

Steps to convert a sentence to clause form

5. Eliminate existential quantification by introducing Skolem functions.
   - For example,
     convert $(\exists x)P(x)$ to $P(c)$ where c is a brand new constant symbol that is not used in any other sentence.
   - c is called a Skolem constant.

   More generally, if the existential quantifier is within the scope of a universal quantified variable, then introduce a Skolem function that depends on the universally quantified variable.
   - For example,
     $(\forall x)(\exists y)P(x,y)$ is converted to $(\forall x)P(x, F(x))$.
   - f is called a Skolem function, and must be a brand new function name that does not occur in any other sentence.
     Example:
     $(\forall x)(\exists y)Loves(x,y)$ is converted to $(\forall x)loves(x,F(x))$ where in this case
     F(x) specifies the person that x Loves.
     (If we knew that everyone loved their mother, then f could stand for the mother-of function.
     i.e. $F(x) \equiv MotherOf(x)$)

Steps to convert a sentence to clause form

6. Remove universal quantification symbols by
   - first moving them all to the left end and making the scope of each the entire sentence
   - then just dropping the "prefix" part.

     E.g., convert (∀x)P(x) to P(x)

7. Distribute "and" over "or" to get a conjunction of disjunctions called **conjunctive normal form**.
   - convert ((P ^ Q) v R) to (P v R) ^ (Q v R)
     convert ((P v Q) v R) to (P v Q v R)

8. Split each conjunct into a separate clause,
   which is just a disjunction ("or") of negated and nonnegated predicates, called **literals**

9. Standardize variables apart again so that each clause contains variable names that do not occur in any other clause

Examples

Convert the sentence
(∀x)(P(x) => ((∀y)(P(y) => P(f(x,y))) ^ ¬(∀y)(Q(x,y) => P(y))))

1. Eliminate <=>
   Nothing to do here.

2. Eliminate =>
   (∀x)(¬P(x) v ((∀y)(¬P(y) v P(f(x,y))) ^ ¬(∀y)(¬Q(x,y) v P(y))))

3. Reduce scope of negation
   (∀x)(¬P(x) v ((∀y)(¬P(y) v P(f(x,y))) ^ (∃y)(Q(x,y) ^ ¬P(y))))

4. Standardize variables
   (∀x)(¬P(x) v ((∀y)(¬P(y) v P(f(x,y))) ^ (∃z)(Q(x,z) ^ ¬P(z))))

5. Eliminate existential quantification
   (∀x)(¬P(x) v ((∀y)(¬P(y) v P(f(x,y))) ^ (Q(x,g(x)) ^ ¬P(g(x)))))

6. Drop universal quantification symbols
   (¬P(x) v ((¬P(y) v P(f(x,y)))^ (Q(x,g(x)) ^ ¬P(g(x)))))

7. Convert to conjunction of disjunctions
   (¬P(x) v ¬P(y) v P(f(x,y)))
   ^ (¬P(x) v Q(x,g(x))) ^ (¬P(x) v ¬P(g(x)))

8. Create separate clauses
   ¬P(x) v ¬P(y) v P(f(x,y))
   ¬P(x) v Q(x,g(x))
   ¬P(x) v ¬P(g(x))

9. Standardize variables
   ¬P(x) v ¬P(y) v P(f(x,y))
   ¬P(z) v Q(z,g(z))
   ¬P(w) v ¬P(g(w))

Examples : Unify

Unify(*p*, *q*) = ϑ όπου Subst(ϑ, *p*) = Subst(ϑ, *q*)

- θ = ενοποιητής (unifier)

Δώστε αν υπάρχει το γενικότερο ενοποιητή για κάθε ζεύγος:

a) $P(A, B, B), P(x, y, z)$

b) $Q(y, G(A, B)), Q(G(x, x), y)$

c) $\text{Μεγαλύτερος}(\text{Πατέρας}(\text{y}), \text{y}), \text{Μεγαλύτερος}(\text{Πατέρας}(\chi), \text{Γιάννης})$

d) $\text{Γνωρίζει}(\text{Πατέρας}(\text{y}), \text{y}), \text{Γνωρίζει}(\text{x}, \text{x})$

Examples : Unify

Unify(*p*, *q*) = *ϑ* όπου Subst(*ϑ*, *p*) = Subst(*ϑ*, *q*)

- θ = ενοποιητής (unifier)

Δώστε αν υπάρχει το γενικότερο ενοποιητή για κάθε ζεύγος:

a) $P(A, B, B), P(x, y, z)$

Unify($P(A, B, B), P(x, y, z)$) *= { x/A , y/B , z/B } (ή κάποια παραλλαγή στη σειρά)*

b) $Q\big(y, G(A, B)\big), Q(G(x, x), y)$

Δεν υπάρχει ενοποιητής. Το x δεν μπορεί να δεσμευθεί και σε A και σε B

c) $\mathrm{Μεγαλύτερος}(\mathrm{Πατέρας}(y), y), \mathrm{Μεγαλύτερος}(\mathrm{Πατέρας}(\chi), \mathrm{Γιάννης})$

{ y/ Γιάννης, x / Γιάννης }

d) $\mathrm{Γνωρίζει}(\mathrm{Πατέρας}(y), y), \mathrm{Γνωρίζει}(x, x)$

Δεν υπάρχει ενοποιητής, γιατί δε γίνεται να ενώσουμε Πατέρας(y) με y

PROLOG

Source : UCLA notes on prolog

Introduction

Prolog stands for Programmable Logic.

It is a computer programming language that is used for
solving problems involving objects and the relationships between those objects.
Programming in Prolog typically consists of:

- defining some *facts* about objects and their relationships
- defining some *rules* about objects and their relationships, and
- asking *questions* about objects and their relationships

Part I –
Defining Facts and Asking Simple Questions

Facts in Prolog are stated in the following manner:

relation(object$_1$,object$_2$,...,object$_n$).

There are three important things to notice here:

1. The name of the relation must start with a lower-case letter.
   In other words, *foo(...)* is ok, but *Foo(...)* is not.
2. The relation can apply to 0 or more objects, though usually it will apply to 1 or more objects.
3. The fact must end with a period.

Part I – Defining Facts and Asking Simple Questions

That said, here are some examples of Prolog facts:

```
tall(dan).
short(napoleon).
sour(lemons).
sister(susan,mary).
plays(charlie_parker,saxophone).
instrument(saxophone).
plays(shaquille,basketball,lakers).
smart_guy(socrates).
```

Part I – Defining Facts and Asking Simple Questions

These facts represent relationships which could be expressed in English as:

tall(dan).	-> Dan is tall.
short(napoleon).	-> Napoleon is short.
sour(lemons).	-> Lemons are sour.
sister(susan,mary).	-> Susan is the sister of Mary.
plays(charlie_parker,saxophone).	-> Charlie Parker plays the saxophone.
instrument(saxophone).	-> A saxophone is an instrument.
plays(shaquille,basketball,lakers).	-> Shaquille plays basketball for the Lakers.
smart_guy(socrates).	-> Socrates is a smart guy.

Part I –
Defining Facts and Asking Simple Questions

After entering a set of facts into Prolog, you can ask questions about the facts. With the facts given above, you could ask questions like:

Are lemons sour?
Is Dan tall?
Is Susan the sister of Mary?

In Prolog these would be stated as

```
sour(lemons).
tall(dan).
sister(susan,mary).
```

You might have noticed that these questions look just like the facts themselves.
The difference is in where the statement is fed to Prolog.
Prolog's default mode is a query mode, where you can ask questions.

Part I –
Defining Facts and Asking Simple Questions

The prompt for query mode looks like this:

```
| ?-
```

When you are in query mode, you can't enter new facts directly,
but only ask questions about the facts that are currently defined.

To add facts to Prolog, you need to either load them from a file,
or enter them directly using user-consult mode.
consult is a Prolog predicate which will accomplish either of these.
To enter user-consult mode, you call the consult predicate with an argument of *user*
(if you're familiar with unix, "user" is equivalent to "stdin" in unix) :

```
| ?- consult(user).
|
```

At this point, the prompt loses the question mark,
indicating that you are no longer asking Prolog questions, but telling it facts instead.

Part I – Defining Facts and Asking Simple Questions

Now we can tell Prolog all the facts I stated previously:

```
| tall(dan).
| short(napoleon).
| sour(lemons).
| sister(susan,mary).
| plays(charlie_parker,saxophone).
| instrument(saxophone).
| plays(shaquille,basketball,lakers).
| smart_guy(socrates).
```

Part I – Defining Facts and Asking Simple Questions

By typing a <control-d>, you send Prolog back into query mode, where we can ask it some questions about the facts just entered

```
| {user consulted, 0 msec 17972 bytes} <--- typed
<control-d> here
| ?-
| ?- sour(lemons).
yes
| ?- tall(dan).
yes
| ?- sister(susan,mary).
yes
```

```
| ?- sister(susan,wanda).
no
| ?- tall(napoleon).
no
```

Part I –
Defining Facts and Asking Simple Questions

For Prolog, answering these five questions is just a matter of verifying that the facts exist in the current data set.
In each of the first three cases, the facts queried on were things that we entered into the set of facts that Prolog knows, so it answered *yes*.
Since the last two were not facts that we entered, Prolog answers *no* to indicate that it couldn't find them.

Note that the query *tall(napoleon)* is not false (from Prolog's point of view) because of the fact *short(napoleon)*
It is merely the case that Prolog can't find the fact *tall(napoleon)*
If we like, we could enter both *tall(napoleon)* and *short(napoleon)* as facts into Prolog, and it wouldn't see any contradiction between them.
This is because Prolog only knows what you enter in as facts and predicates.
You could enter a relationship between the *tall* and *short* predicates, but we'll get to that later.

Part I –
Defining Facts and Asking Simple Questions

First, consider a different type of query. So far, all of the questions asked could be answered with a yes or a no. What about questions like these:

Who is tall?
Who is Mary's sister?
Who plays what instrument?

In these questions, there is some unknown element which we are expecting to be the response; filled by the *who* or *what* in the sentences.
These could be restated like this in Prolog:

```
tall(Who).
sister(Who,mary).
plays(Who,Instrument).
```

In the Prolog versions of the queries, the unknown element is filled in with a variable.
Any term that starts with an upper case letter in Prolog is a variable.
In the three queries above, All 3 uses of *Who* and the term *Instrument* are all variables. The term *mary* is still a constant, since it is lower case. Predicate names (in this case *tall*, *sister*, and *plays*) must always start with a lower case letter. You can't make a query like *AnyRelation(susan,mary).*

Part I –
Defining Facts and Asking Simple Questions

Prolog's response to a query with one or more variables is somewhat different than its response to yes/no questions:

```
| ?- tall(Who).
Who = dan ?
```

At this point, Prolog pauses to ask if Dan is an acceptible answer.
As the user, you have two choices.

- You can either type a carriage return, in which case your query predicate will exit with success.
- Or you can type in a semi-colon and a carriage return, and Prolog will fail the current set of variable bindings and attempt to resolve the query with different variables.

```
| ?- tall(Who).   | ?- tall(Who).
Who = dan ?       Who = dan ? ;
yes               no
```

Part I –
Defining Facts and Asking Simple Questions

In this case, since we only have one person being defined as tall in the fact set,
Prolog fails on the re-try.
Note that choice of the variable "Who" is arbitrary.
It has no meaning, other than the fact that the upper case starting letter makes it a variable.
We could as easily have used "Why" or "Checkers" or "X" and gotten a similar response:

```
| ?- tall(X).
X = dan ?
yes
```

In answering this type of question, Prolog shows a bit more of its matching process.
Prolog is trying to find some way to *unify* variables and constants so that the query matches some fact in the data set.
In this case Prolog is able to unify the query *tall(X)* with the fact *tall(dan)* from the data set by assigning *dan* to the variable *X*.
When it finds this match, it shows us a list of the variable bindings which will allow the query to succeed.

Part I – Defining Facts and Asking Simple Questions

When we typed a semi-colon before, we told Prolog to fail that unification.
In other words, we told it to throw away those variable bindings and try to find a different solution. Here are some other examples of Wh-type queries:

```
| ?- sister(Who,mary).
Who = susan ?
Yes
| ?- plays(Who,WhatInstrument).
Who = charlie_parker, WhatInstrument = saxophone ? ;
No
| ?-
```

The last example illustrates another facet of Prolog's processing. When asked for an alternate answer, it couldn't find one. There is another "plays" clause in the data set, but it has three arguments while the query only has two, so it doesn't match

Part I –
Defining Facts and Asking Simple Questions

To summarize, Prolog's method for trying to match up queries with facts is this:

1. Go through the list of facts in the order in which they were entered.
2. Look for a fact that has the same predicate name (i.e. plays = plays) as the query.
3. Verify that it has the same number of arguments.
4. For each pair of arguments ($queryarg_i$, $factarg_i$) try to unify them:
   - If $queryarg_i$ & $factarg_i$ are both constants, then they must match exactly.
   - If one is a constant and one is a variable, then bind the constant to the variable
   - If both are variables, then map them to the same variable on the variable stack and (we'll see where this applies later).
5. If all the variables can be unified, then the match has succeeded - return the list of variable bindings.
6. If the user types a ;, then look for another fact that unifies.

Most Prolog interpreters are actually a little bit smarter than what is indicated in steps 1-2.
They usually index the set of predicates names, so they don't have to try to match the predicate name, just the variables.
Also note that you can have predicates as arguments to other predicates (more on this later).

Part I –
Defining Facts and Asking Simple Questions

At this point, let's set up a new set of facts to deal with, and try some queries that have more than one possible match.

```
| ?- consult(user).
| sister(gertrude,wilbur).
| sister(sally,mary).
| sister(susan,mary).
| {user consulted, 0 msec 0 bytes}
yes
| ?- sister(Who,mary).
Who = sally ? ;
Who = susan ? ;
no
```

Here I've just found both of Mary's sisters with the query.

Part I – Defining Facts and Asking Simple Questions

A common mistake people make at first is to accidently capitalize a proper name when trying to make a query.
Another common error is forgetting to capitalize something that's supposed to be a variable. Here are examples of both of these errors:

```
| ?- sister(Who,Mary).
Who = gertrude, Mary = wilbur ?
yes
| ?- sister(who,mary).
no
```

In the first example, we entered the variable *Mary* rather than the constant *mary*.
The error should be apparent when we see that we're getting back one more variable binding than expected.
In the second example, we forgot to capitalize the variable *who*, so Prolog couldn't find the fact sister(who,mary).

PROLOG

Part II - Multi-clause Queries

Sometimes a query involves more than one relation.
For example, suppose we have the original data set and we want to know who plays an instrument.
We could make a query like this:

```
plays(Person,X), instrument(X).
```

Here we have asked Prolog to find "a person who plays something" and then verify that that "something" is an instrument.
This is called a *conjunctive* query, because it's the conjunction of two clauses.
The important point is that the X in the *plays* predicate is the same X as in the *instrument* predicate.

Part II - Multi-clause Queries

Prolog will evaluate this query by first trying to match the first predicate, plays(Person,X).
Assuming it has the first set of facts that I suggested here, then it will match this with plays(charlie_parker,saxophone).
This will bind the variable *Person* to the value *charlie_parker*, and the variable *X* to the value *saxophone*.
Prolog will then try to match the second predicate, but since X is already bound, what it tries to match will be *instrument(saxophone)*.
Since that exact predicate is in the set of facts we entered, Prolog succeeds, and tells us the bindings of the variables that were in the query:

```
| ?- plays(Person,X), instrument(X).
X = saxophone,
Person = charlie_parker ?
Yes
| ?-
```

Part II - Multi-clause Queries

You could make a more complicated query by asking, "Which of Mary's siblings plays an instrument?" Now there are 3 clauses involved:

```
sister(S,mary), plays(S,X), instrument(X).
```

In this case the query would fail, as charlie_parker is the only person currently in our data set who plays an instrument, and Mary's not his sister (at least as far as Prolog knows).

This is an important point**. What Prolog knows is only what you tell it**.
So something can only be true if you have provided it as a fact in the data set, or you have provided a combination of facts and rules from which Prolog can prove it.
This is known as the **Closed World Assumption**. *If you can't prove it, then it must be false.*
Obviously this isn't always a nice assumption, as there will always be many things that are true but can't be proven (you'd have to have an awfully big computer to hold *all* the facts that are true).

Part II - Multi-clause Queries

In the previous example, note that the order of the clauses is mostly irrelevant. You would get the same result[s] if you did any of these searches:

1. sister(S,mary), instrument(X), plays(S,X).
2. instrument(X), plays(S,X), sister(S,mary).
3. instrument(X), sister(S,mary), plays(S,X).
4. plays(S,X), instrument(X), sister(S,mary).
5. plays(S,X), sister(S,mary), instrument(X).

The reason you may want to choose one order over another is to reduce choices for further search. For example, it's a good idea to have successive clauses use variable bindings from the previous clause to restrict their search.
For example, version 2 above first finds something that is an instrument, and binds it to X. Then that value is bound in the second clause, so it can search on *plays(S,trumpet)* or *plays(S,guitar)* rather than *plays(S,X)*.

Part II - Multi-clause Queries

The original form we gave for the query is actually the best of all, since it starts out with its first clause, sister(S,mary), being partially restricted.
Then it passes the binding of S from the first clause to the second clause, so the second step in the search is partially restricted as well.
Finally, it passes the binding of X from the plays clause to the instrument clause, so the last step is just to test if the clause is there or not.

You can do a conjunctive query with totally unrelated clauses, just as you can ask "What's the time and what year were you born?“
On the other hand, there's no advantage to asking them as a conjunctive query rather than two separate queries.

PROLOG

Part III - Defining Rules

S far, you might get the impression that you need to state every individual fact in order to make Prolog do anything useful.
Actually there is a way to state a relationship that doesn't require you to state every possible detail of the relationship.
You can do it with a rule.

For example, suppose I want to assert the fact that *anyone who is a tall person plays basketball*.
If I have a data set that has 200 people listed as tall, I could add 200 clauses for the same people that indicates that each one plays basketball, or I could write the following rule:

```
plays(X,basketball) :- tall(X).
```

You could read this as "X plays basketball on the condition that X is tall."

Part III - Defining Rules

Now if I want to find out if
Dan plays basketball,
and this is my data set:

```
short(john).
tall(dan).
tall(joe).
thin(joe).
plays(X,basketball) :- tall(X).
plays(X,piano) :- short(X)
```

Then here's how Prolog will work:

1. I give the query "plays(dan,basketball)."
2. Prolog matches this against
   the head of the rule "plays(X,basketball)".
3. Prolog binds X to dan, and tests the conditional clause.
4. Now Prolog has the query "tall(dan)"
5. tall(dan) is in the data set, so Prolog succeeds.
6. Since there were no variables in the original query, Prolog just responds with "yes".
   (If the query was like plays(dan,X), then Prolog would tell me X = basketball)

Part III - Defining Rules

Ok, so what if only tall, thin people play basketball? Then I could write the right hand side of the rule as a conjunctive goal:

```
plays(X,basketball) :- tall(X), thin(X).
```

Now when Prolog matches the head of this query,
it has to succeed on each of the clauses in the right hand side for the overall match to succeed.

Part III - Defining Rules

If I now want to ask Prolog "who plays basketball?" using the same data set, then here's how Prolog will work:

1. I ask the query plays(Who,basketball).
2. Prolog matches this to the head of the rule plays(X,basketball).
3. Prolog binds Who = X
4. Now it will substitute the two goals, "tall(X)" and "thin(X)"
5. Prolog tries to match tall(X)
6. It succeeds, by matching on the fact tall(dan)
7. Now Prolog binds X=dan and tries the second goal, thin(dan)
8. Prolog fails to find thin(dan).
9. It goes back and tries to find a different solution for the goal tall(X). As part of this, it undoes the binding of X=dan
10. Prolog finds a new match in tall(joe).
11. Prolog binds X=joe and tries the second goal, thin(joe).
12. Prolog finds the fact thin(joe), so it succeeds
13. Since both of the conditions of the rule succeeded, Prolog succeeds on the rule by returning the variable bindings from the original query, remember Who=X, and from step 11, we have X=joe, so...
14. Prolog displays "Who = joe" and waits to see if we accept the answer.
15. I type a carriage return, so Prolog cheerfully tells me "yes", indicating it answered my query affirmatively.

Of course, it does most of this behind the scenes, so all I see is the result from step 13, and the "yes" from step 14. Here's what it looks like:

```
| ?- plays(Who,basketball).
Who = joe ?
Yes
| ?-
```

Part III - Defining Rules

If you want to see the details of what clauses Prolog is trying out in a search, you can turn on trace by calling the trace predicate.
Here's what a trace of the previous query looks like:

```
| ?- trace.
{The debugger will first creep -- showing everything (trace)}
yes
{trace}
| ?- plays(Who,basketball).
   1 1 Call: plays(_187,basketball) ?
   2 2 Call: tall(_187) ?
   2 2 Exit: tall(dan) ?
   3 2 Call: thin(dan) ?
   3 2 Fail: thin(dan) ?
   2 2 Redo: tall(dan) ?
   2 2 Exit: tall(joe) ?
   3 2 Call: thin(joe) ?
   3 2 Exit: thin(joe) ?
   1 1 Exit: plays(joe,basketball) ?
Who = joe ?
Yes
{trace}
| ?-
```

PROLOG

Part IV - Lists in Prolog

Prolog has a very nice format for dealing with lists.
Prolog represents the head (the equivalent of the CAR in Lisp) and the tail (the equivalent of the CDR) like this:

```
[Head|Tail]
```

Notice that both Head and Tail are capitalized, so they're both variables in this case.
Of course, they could be constants, or terms containing variables, like these:

```
[a|Tail]
[X|[a,b,c]]
[drove(john,car)|OtherFacts]
[implies(X,Y)|OtherFacts]
```

For the last two examples here, think of the problem where you're storing some set of facts, but within some sub-context. For example, the Encyclopedia Brown problems. You need to be able to distinguish things that someone *says* happened from things that are *observed* to happen.
If you just define it all as facts in Prolog, then you can't tell what's real from what's a lie.
Thus, you can keep two lists, one of real facts and one of "facts" stated by the person who Encyclopedia accuses.
Then you search for something in the "facts" that contradicts something in the real facts.

Part IV - Lists in Prolog

There is a special symbol for the null list, it's [ ].
You can't pull the head or tail off of a null list. If you try to match [ ] with [H|T], it won't match.

If you don't care what the value of some variable is, you can use an _ for the variable name. For example, if you have a rule that only cares about the head of a list, you could start the rule like this:

```
process_head([Head|_]) :- ...
```

On the other hand, if you don't care about the head, but you only plan on dealing with the tail of the list, you could do it this way:

```
process_tail([_|Tail]) :- ...
```

The benefit of this is that if Prolog knows that you don't care about a value, it doesn't try to store its variable binding on the stack, so you save space.
It'll become clear where you could do this in the next section.

PROLOG

Part V - Prolog Programming

Now that we've got all the building blocks covered, let's consider how to program a simple function in Prolog.
The member function, takes an element and a list as arguments and returns true if the element is contained somewhere in the list.
The code in Lisp looks something like this

```
(defun member (E L)
        (cond ((null L) ‘F)
              ((equal E (CAR L)) ‘T)
              ('T (member E (CDR L)))))
```

The first condition checks if the list is ( ), and if so it fails.
The second condition checks if the element is at the head of the list, and the last condition recurses to check if the element is in the tail.
The definition of this function in Prolog is very similar:

```
member(H,[H|T]).
member(X,[H|T]) :- member(X,T).
```

The first statement here is like the second condition of the Lisp code. It says that the first argument (the element) is a member of the second argument (the list) if it is the head of the list.
The second statement is like the recursive call in the Lisp code. It says that the element is a member of the list on the condition that it is a member of the tail.

Part V - Prolog Programming

The only thing missing here that was in the Lisp code is the first condition, which checked for the null list.
Remember from the last section that the null list in Prolog can never match a [H|T] type representation.
Thus, the extra check for the null list isn't needed in Prolog.

To see how this works, suppose we call this function with the query member(2,[1,2,3]).
Here's how Prolog handles it:

1. Prolog tries to match our call, *member(2,[1,2,3])* to the first statement, *member(H,[H|T])*, and fails, since 2 is not equal to 1 (remember, the H of the first argument has the same binding as the H in the list argument).
2. Prolog tries the second statement, and succeeds in matching our query to the head of the second statement
3. Prolog binds the variables X = 2, H = 1 and T = [2,3]
4. Now Prolog must try to solve the right hand side of the rule, so it tries the goal "member(2,[2,3])"
5. Prolog succeeds in matching member(2,[2,3]) to the first statement, so that goal succeeds.
6. Since the right hand side goal was successfully matched, Prolog succeeds the head that was matched in step 2.
7. Since that match was generated by the initial call, Prolog has solved the whole query, and it returns "yes".

Part V - Prolog Programming

One neat thing about Prolog functions is that they usually don't distinguish what's an input and what's an output.
Thus we could call the member function like this:

```
member(X,[a,b,c]).
```

In this case, Prolog would succeed in matching our query to the first statement and return "X = a". If we asked for more results by typing in a ";", Prolog would first give us"X = b", then "X = c", then it would fail and say "no".

One more thing to notice here is that the first statement *member(H,[H|T]),* doesn't really do anything with the tail. Thus this would be a good place to use the _ to indicate we don't care about its value. Similarly, the second statement doesn't really use the head, so we could use it again there. Here's the revised code:

```
member(H,[H|_]).
member(X,[_|T]) :- member(X,T).
```

Part V - Prolog Programming

Another simple function on lists can be implemented simply in Prolog as well; namely, the append function. If we were implementing append in Lisp, we would do it something like this:

```
(defun append (L1 L2)
     (cond ((null L1) L2)
                (t (cons (car L1) (append (cdr L1) L2)))
     )
)
```

The processing here works the following way.
If the first list is ( ), then you just return the second list.
Otherwise, you take the car of the first list, and put it back on to the front of the result you get when you append the cdr of the first list to the second list.
When this is executed in Lisp, it'll take elements off the first list one by one until it's ( ), then it'll go back and put them back on one by one with the cons.

Part V - Prolog Programming

A Prolog implementation works *exactly* the same way:

```
append([],L,L).
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```

The symmetry between the two functions is obvious:
the first line of the Prolog program is equivalent to the first condition of the Lisp program,
and the second line is equivalent to the second condition.
The only differences are

1) the way lists and their manipulation are represented,
2) that the result is returned as the third argument instead of as a return value for the function (since Prolog functions can only return yes or no).

In case it's not clear, let us change the variable names to represent what they are in the Lisp code

```
append([],L2,L2).
append([CarL1|CdrL1],L2,[CarL1|RecursiveAppendResult]) :- append(CdrL1,L2,RecursiveAppendResult).
```