Programming Languages Workshop
CS 67420
Lecture 6

Huji, Spring 2016

Dr. Arie Schlesinger

Prolog

- "Programming with Logic"
- Different from other programming languages
  - Declarative (not procedural)
  - Recursion (no for/ while loops)
  - Relations (no functions)
  - Unification, kind of substitution

1st interpreter: 1972 – Alain Colmerauer, Philippe Roussel, Marseilles

2005- Prolog used to program natural language interface in International Space Station by NASA

2011- Parts of IBM's Watson QA supercomputer were coded in Prolog

Basic Idea

You-

- Describe the situation of interest in a Knowledge Base (KB)
- Ask a question

Prolog-

- logically deduces “new” facts about the situation we described
- gives us its deductions back as answers

consequences

Think declaratively, not procedurally

– Challenging

– Requires a different mindset

- High-level language

– Not as efficient as, say, C

– Good for rapid prototyping

– Useful in many AI applications (knowledge representation, inference)

Free prolog software

swi-prolog:

www.**swi-prolog**.org

TuProlog:

http://apice.unibo.it/xwiki/bin/view/Tuprolog/Download

Both are installed on CSE computers.

There is a lot of free learning material at these sites.

Study Materials

V Good & free materials at : “Learn Prolog Now!”

www.**learnprolognow**.org/

Prolog Programming for Artificial Intelligence,
by Ivan Bratko (4th ed)

The Art of Prolog (2nd ed), by Leon Sterling, Ehud Shapiro

Programming in Prolog (5th ed), by Clocksin and Melish

The prolog charm

How small a Prolog prog can be ?

There are 2 parts:

A “knowledge base” (kb), like this:

```
sunny.
```

Which the user can Query, and get answers:

```
?-sunny.  % query
true      % answer
```

The querying is done from outside of the kb.

kb

The kb is like a "vault of truth" – all its contents are considered to be *true* – like axioms.

To query it, means to check if the query is *also true* - which means:

- does it *itself* literally occur in the kb ? or
- can it be deduced from the contents of the kb ?

Checking if the literal text of the query is in the kb, is easy.

Checking if it can be *deduced* from the kb contents, is the main business of the Prolog interpreter, whose answers include `true, false,` and maybe some other details derived from the deduction process.

Studying *how this deduction works*, is the course business.

Main Idea

By Bob Kowalsky, Edinburgh univ – 74:

Deduction can be viewed as a form of computation:

A declarative statement like:

```
A if X and Y and Z
```

Could be interpreted as:

```
To solve A, solve X and Y and Z
```

Prolog as a system for querying kbs – KB 1

So, this kb is a collection of facts, that describe some situation.
They are not supposed in any way to be true by any real life criterions.
The facts get their "truthfulness" by simply being written in the kb

Example, we can state that :
- Mia, Jody, and Yolanda are women,
- Jody plays the guitar,
- a party is taking place,
using the following five *facts*:

```
woman(mia).
woman(jody).
woman(yolanda).
playsGuitar(jody).
party.
```

The first letter of the names mia , jody , and yolanda , the properties woman and playsGuitar , and the proposition party, are in lower-case.

Queries examples

```
?-woman(mia).
true
```

(because this fact appears *literally* in the kb)

```
?-  playsGuitar(jody).
true
```

```
?-  playsGuitar(mia).
false
```

because this fact is not in the kb, nor it can be deduced from it.

?- is the queries prompt

Queries examples

```
?-party.
true

?-  playsGuitar(vincent).
false

?-  tatooed(jody).
false

?-rockConcert.
false
```

KB1 has no info about the person *Vincent*, nor about the property (being) *tatooed*, or of the fact *rockConcert*.

some Prolog implementations will respond with an *error message*

kb2 with rules: the new kind of "truths"

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):-happy(yolanda).
playsGuitar(mia):-listens2Music(mia).
playsGuitar(yolanda):-listens2Music(yolanda).
```

There are two facts :

```
listens2Music(mia),happy(yolanda) .
```

The last three items it contains are rules.

A rule is a small *"logic machinery"*, these here are inference machines :

`A :- B.` means `A if B`.

the `:-` should be read as "if"

The deduction process and its logical machinery is getting clearer

rules

Rules state info that is conditionally true of the situation.
The 1st rule says that Yolanda listens to music if she is happy,
The 3rd rule says that Yolanda plays guitar if she listens to music.
General form of a rule - `Head:-Body`.

In general rules say: *if the body of the rule is true, then the head of the rule is true too*.

The key point:
If a KB contains a rule `head:-body`, and Prolog knows that `body` follows from the information in the KB, then Prolog *can infer the* `head` (as a "truth").

This fundamental deduction step is called *modus ponens*.

example

```
?-playsGuitar(mia).
true
```

Prolog *can't find* `playsGuitar(mia)` as an explicit fact KB2, but it can find the rule:

`playsGuitar(mia):- listens2Music(mia).`

and the fact: `listens2Music(mia)` .

So Prolog can use *modus ponens* to deduce from these 2 that `playsGuitar(mia)`.

Prolog can chain together uses of modus ponens

Deductions chain- from these 3:

```
happy(yolanda).
listens2Music(yolanda):-happy(yolanda).
playsGuitar(yolanda):-listens2Music(yolanda).
```

We get:

```
?- playsGuitar(yolanda).
true
```

`playsGuitar(yolanda).` is inferred knowledge (implicitly present only), it is *not explicitly* in the kb.
Prolog, deduced it by "activating the 2 inference machines" above.

Prolog can then use it just like an *explicitly* recorded fact.

So

Any fact produced by an *application of modus ponens can be used as input to further rules*.

By chaining together applications of modus ponens in this way, Prolog is able to retrieve info that *logically follows* `from the rules and facts in the knowledge base`.

Terminology: clauses, predicates

The facts and rules contained in a kb are called *clauses*.

Thus KB2 contains five clauses, 3 rules and 2 facts.

We'll call the rules *predicates* (or procedures). Here the predicates are:

```
listens2Music
happy
playsGuitar
```

happy predicate is defined using a single clause (a fact).

listens2Music,playsGuitar predicates are each defined using two clauses (in one case, two rules, and in the other case, one rule and one fact).

The facts in the predicates *body* are called goals (standard terminology)

Facts

We can view a fact as :
- a rule with an *empty* body,
- conditionals without conditions, (like if true then…)
- "degenerate" rules.

We'll think about Prolog progs in terms of *the predicates they contain.*

The predicates are the concepts we find important, and the various clauses we write concerning them are our attempts to pin down:
- what they mean, and
- how they are inter-related.

Kb3 with 2 facts and 3 new rules

```
happy(vincent).
listens2Music(joe).
playsGuitar(vincent):-
          listens2Music(vincent),
          happy(vincent).
playsGuitar(joe):-
          happy(joe).
playsGuitar(joe):-
          listens2Music(joe).
```

KB3 defines *differently* the same 3 predicates from KB2:

`happy` , `listens2Music` , and `playsGuitar`

Rule with 2 goals

the rule:

```
playsGuitar(vincent):-
            listens2Music(vincent),
            happy(vincent).
```

has two items in its body, or (to use the standard terminology) two goals.

The comma ,that separates the goals `listens2Music(vincent)` and `happy(vincent)` in the rule's body, means and

This is the way *logical conjunction* is expressed in Prolog.
This rule says: "Vincent plays guitar if:
-he listens to music, and
-he is happy".

example

```
?-  playsGuitar(vincent).
false
```

because although KB3 contains `happy(vincent)` , it does not explicitly contain the `listens2Music(vincent)` , and this fact *cannot be deduced* either.

So KB3 fulfils only *one of the two* preconditions needed to establish `playsGuitar(vincent)` , and our query fails.

space

The spacing used in a rule is irrelevant.

For example, we could have written it as

```
playsGuitar(vincent):-   happy(vincent),
                         listens2Music(vincent).
```

and it would have meant exactly the same thing.

Prolog offers freedom in the way we set out kb's, to keep our code readable

Two heads rules: or

KB3 contains two rules with the same head:

```
playsGuitar(joe):-
            happy(joe).
playsGuitar(joe):-
            listens2Music(joe).
```

This means joe plays guitar either if he *listens to music*, or if he *is happy*.
listing multiple rules with the same head is a way of expressing logical disjunction (or).

So if we posed the query

```
?-  playsGuitar(joe).
True
```

KB3 contains `listens2Music(joe)` ,and Prolog can apply *modus ponens* using the rule:

```
playsGuitar(joe):-
            listens2Music(joe).
```

to conclude that `playsGuitar(joe) .`

Another way to express or

Instead of the pair of rules above, just use ; (semicolon)

```
playsGuitar(joe):-
        happy(joe);
        listens2Music(joe).
```

The semicolon ; is the Prolog symbol for or

Which is better?

-extensive use of semicolon can make Prolog code *hard to read*.

-the semicolon is more *efficient* as Prolog only has to deal with one rule.

why "Prolog" is short for "Programming with logic".

`:-` means implication,

`,` means conjunction,

`;` means disjunction.

`modus ponens` - plays an important role in Prolog programming.

kb4

```
woman(mia).
woman(jody).
woman(yolanda).
loves(vincent,mia).
loves(marsellus,mia).
loves(hemed,motek).
loves(motek,hemed).
```

There are no rules, only a collection of facts.

Queries with variables on kb4

Example:

```
?- woman(X).
```

X is a variable (vars begin with capitals)

A variable isn't a name, rather it's a placeholder for information- a Box.

This query asks Prolog: which of the individuals you know about is a woman.

Prolog goes through KB4, from top to bottom, trying to unify (or match) the expression `woman(X)` with the info in KB4.

first item in kb4 is `woman(mia)`. (order counts..)

So, Prolog unifies `X` with `mia` , and answers:

```
X  =  mia
```

(Terminology :Prolog instantiates X to mia , or it binds X to mia .)

we get the name of the woman , not only "true"

It actually gave us the var binding that led to success.

Many answers

Vars can *unify* with different things.

There is info about *other women* in the kb4.

We can get it by typing a semicolon:

```
X = mia ;
```

; means or , so this means: are there any alternatives ?

Prolog begins working through the kb again

(it starts from where it was last time), and unifies X with jody ,

```
X = mia ;
X = jody
```

If we press ; a second time, Prolog returns the answer:

```
X = mia ;
X = jody ;
X = yolanda
```

Query with vars

if we press ; a 3rd time, Prolog responds `false`.

No other unifications are possible: no other facts start with the symbol woman .

The last four rules in kb4, can not be unified with a query of the form woman(X) .

A new query:

`?- loves(marsellus,X), woman(X).` (, means and)

is there any individual X such that Marsellus loves X and X is a woman ?

yes: Mia is a woman (fact 1) and Marsellus loves Mia (fact 5).

Prolog can search through the kb and find that if it unifies X with Mia, then both conjuncts (goals) of the query are satisfied the answer

`X = mia`

Prolog's ability to perform unifications is crucial.

Kb5-rules with vars

```
loves(vincent,mia).
loves(marsellus,mia).
loves(hemed,motek).
loves(motek,hemed).

jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

KB5 contains four facts about the `loves relation`
and one rule with 3 *vars* (X , Y , Z).
The rule says X will be jealous of Y if there is some
Z that X loves, and Y loves that same Z too.

this is a general statement: not stated in terms of anyone in particular — it's a conditional statement about *everybody* in our world.

Who is jealous – query?

```
?-  jealous(marsellus,W).
```

can you find an individual W such that Marsellus is jealous of W ?

Vincent and Marsellus, both love Mia.

So Prolog will return the value

```
W  =  vincent
```

Questions:

Are there any other jealous people in KB5?

what query would will return the names of all the jealous people.

Prolog Syntax

facts, rules, and queries are built out of terms

A bit of syntax:

Atoms:

- All terms that consist of letters, numbers, and the underscore and start with a *lower-case* letter are atoms:

```
iosi ,old_falafel , listens2Music ,playsGuitar. .
```

- All terms that are enclosed in single quotes are atoms:

```
'Ety','a fly','2-way roaD',' &^%&#@ $  &*',' ','(@ *+ ',
```

- Certain special symbols are also atoms:

```
+, ,, :- , @=,====>,; ,
```

some like `;` and `:-` have a pre-defined meaning.

Variables:

- All terms that consist of letters, numbers, and the underscore and start with a *capital letter* or an underscore are variables:

```
X, Y, Var, _tag, X_6, List, MyList, _head, Tail
```

- `_` is an anonymous variable: two occurrences of `_` are different variables.

terms

Four kinds of terms in Prolog:

-atoms, -numbers, constants

-variables, -complex terms (or structures). simple terms

Basic chars :

The upper & lower case letters, digits

some special chars, like _, + , - , * , / , < , > , = , : ,. , & , ~ ,blanc

Numbers

Prolog is not in numeric calculations, so:

Real numbers are sometimes supported but not important

Integers are supported and sometime useful.

Complex terms- structures

The building blocks :constants, and vars

Complex terms are build with a functor followed by a seq of args separated by commas in parantheses, and placed after the functor.

(no space between the functor and the left parenteses)

The *functor* must be an atom, args can be any kind of term.

Examples,

playsAirGuitar(jody), loves(vincent,mia), jealous(marsellus,W) .

The definition allows to keep nesting complex terms inside complex terms (recursive structure).

Complex cornflex

Example:

```
hide(X,father(father(father(joe))))
```

functor is hide , and it has 2 args:

the var X , and the complex term father(father(father(joe))) .

This complex term has father as its functor, and another complex term, as sole arg: father(father(joe)) , and so on..

Such nested (or recursively structured) terms enable us to represent many problems *naturally*.

In fact the interplay between recursive term structure and variable unification is the “pride” of Prolog.

arity: n of args of a complex term

arity of `woman(mia)` is 1,

Arity is important to Prolog.

`likes(iosi,falafel)`

`likes(iosi,falafel,pizza)` are 2 distinct predicates with the same functor.

Notation for errors, comments..:

`likes/2, likes/3.`

Prolog will understand..

Unification

there are 3 types of terms:

Constants:

- atoms (such as ety), or numbers (such as 7).

- vars (such as X , A1 , and _myData)

Complex terms with the general form:

`functor(term_1,...,term_n)` .

Intuitive first definition for unification:

Two terms unify if:

- they are the *same* term, or
- if they contain variables that can be uniformly instantiated with terms in such a way that *the resulting terms are equal*.

Unification

Trivial Successes:

- the terms mia and mia unify, because they are the same atom.
- the terms 7 and 7 unify, because they are the same number,
- the terms X and X unify, because they are the same variable,
- the terms woman(mia) and woman(mia) unify, because they are the same complex term.

Obvious Failures:

-The terms woman(mia) and woman(lea) , do not unify, as they are not the same (and neither of them contains a var that could be instantiated to make them the same).

Interesting unifications

Interesting successes:

-the terms mia and X are not the same. But the var X can be instantiated to mia which makes them equal. So, by our prev def, mia and X will unify.

-the terms woman(X) and woman(mia) unify, because they can be made equal by instantiating X to mia .

Interesting failures:

loves(vincent,X) , loves(X,mia) . Can you find an instantiation of X that makes the two terms equal? No

X to vincent would give us loves(vincent,vincent) ,loves(vincent,mia) , which are not equal.

X to mia, would yield loves(vincent,mia) , loves(mia,mia) , which aren't equal either.

unification

When unification succeeds, Prolog shows the vars values that did it.

It performs all the necessary instantiations, so that the terms really are *equal* afterwards.

This functionality, together with the fact that we are allowed to build complex terms (recursively structured terms), makes unification a challenging programming mechanism.

Matching

- Two *atoms* match if they are the same atom.

```
Ex.: harry = harry, but harry \= 'Harry'.
```

- A *variable* matches any other Prolog term.

The variable gets instantiated with the other term.

```
Ex.: X = wizard(harry)
Ex.: X = Y
```

- Two *complex terms* match if they have the same *functor* and the same arity, and if all pairs of parallel args match.

Ex.: `like(harry,hargrid) = like(harry,X)`

Ex.: `like(harry,hargrid) = like(harry,X,Y)`

Ex.: `like(harry,hargrid) \= like(X,X)`

Examples and the =/2 predicate

The =/2 predicate tests whether its two args unify.

```
?- =(mia,mia).
true

?- =(mia,vincent).
false
```

Prolog syntatctic sugar: use the infix notation

```
?-mia = mia.
```

Unify more

```
?-  2  =  2.
true
```

small surprise :

```
?-  'mia'  =  mia.
true
```

In Prolog , 'mia' and mia are the same atom.

But:

?- '2' = 2.

False

2 is a number, but '2' is an atom, they can not be the same.

= with vars

Example with a var:

```
?-  mia  =  X.
X  =  mia
true
```

the var X can be unified with the constant mia.

what happens with the following query?

```
?- X = Y.
```

Depending on your Prolog implementation, you may just get back the output

```
?-  X  =  Y.
true
```

Prolog agrees that the two terms unify
(variables unify with any term, so they unify with each other) ,
from now on, X and Y denote the same object, that is, share values.

Unifying vars

On the other hand, you may get the following output:

```
X = _5071
Y = _5071
true
```

Prolog has created a new var _5071, and from now on both X and Y share the value of this var.
In effect, Prolog creates a common var name for the two original variables.

Another example involving only atoms and vars:

```
?- X = mia, X = vincent.
false
```

Separately, Prolog would succeed at each of them.
After the first goal, X is instantiated/equal to mia,
So it can't unify with vincent anymore. Hence the second goal fails.

An instantiated variable isn't really a variable anymore: it has become what it was instantiated with.

Unify complex terms

```
?-  k(s(g),Y)  =  k(X,t(k)).
X  =  s(g)
Y  =  t(k)
true
```

the same functor and arity.

```
s(g) to X
Y  to t(k)
```

Another example:

```
?-  k(s(g),  t(k))  =  k(X,t(Y)).
X  =  s(g)
Y  =  k
true
```

Another example:

```
?-  loves(X,X)  =  loves(marcellus,mia).
false
X to marcellus , X to mia
```

No unification here

The occurs_check

```
?- father(X)  =  X.
```

A standard unification algorithm will not perform it, because it has an infinite size term.

```
X = father(father(father(father(father(father
        (father(father(father(father(father(father
        (father(father(father(father(father(father
        (father(father(father(father(father(father
        (father(father(father(father(father(father.....
```

Prolog decides that the terms can unify, but represents the inf term as `father(X).`

(a finite representation of an infinite term)

And here?

```
?- X = father(X), Y = father(Y), X = Y.
```

The same "grace": `X=Y, Y=father(Y).`

unify_with_occurs_check()

A *standard unification alg* first carries out a `occurs check:` it checks if the var occurs in the term.

If yes, it declares that unification is *impossible*

If not, the alg tries to carry out the unification.

Prolog assumes that no one will give it anything dangerous.

So it omits the occurs check.

But it has a built-in pred that does unification with the occurs check:

```
?-unify_with_occurs_check(father(X),X).
false
```

Programming with unification

what it means for a line to be vertical or horizontal respectively:

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

2 facts , no rules, deepest levels args are vars

(by Ivan Bratko.)

```
?- vertical(line(point(1,1),point(1,3))).
true
?- vertical(line(point(1,1),point(3,2))).
false

?-horizontal(line(point(1,1),point(2,Y))).
Y = 1 ;
false
```

The beauty and the Prolog

```
?-horizontal(line(point(2,3),P)).
P = point(_G1829,3) ;
false
```

(_G1829 is a var !)

family

Great, but not ancestor enough..

```
grandparent of(X,Y) :- parent of(X,Z), parent of(Z,Y).
greatgrandparent of(X,Y) :- parent of(X,Z),
                            parent of(Z,A),
                            parent of(A,Y).
greatgreatgrandparent of(X,Y) :- parent of(X,Z),
                                 parent of(Z,A),
                                 parent of(A,B),
                                 parent of(B,Y).
```

Good for 3 “greats”, not enough for all ancestors.

Recursion helps:

```
ancestor of(X,Y) :- parent of(X,Y).
ancestor of(X,Y) :- parent of(X,Z),
                    ancestor of(Z,Y).
```