Prolog 1

MSc Computing

Fariba Sadri

With thanks to Keith Clark for the use of some of his lecture material

Prolog

Prolog is a high level declarative programming language based on a subset of predicate logic. It is a logic programming language.

Particularly favoured for applications in:

- AI
- expert system and
- computational linguistics.

Relevance to courses next term:

- Introduction to Artificial Intelligence: uses Prolog
- Argumentation and Multi-Agent Systems: uses Prolog
- Logic-based Learning course: uses HAIL (Hybrid Abductive Inductive Learning) and ASP (answer Set Programming)

- We will be using Sicstus Prolog and Windows. You can use Linux.

- Program files are saved as plain text.

- Prolog tutorials in lab 219 on Thursdays in week 5 (2 November), and other Thursday (will be annonced).

- Assessment is by:
  - An assessed lab exercise and
  - Lab examination in Jan
- Possible Mock test in week 11 (unassessed)

Example:
A very short Prolog program

Recall from Predicate Logic:

/* Anyone passes the MSc if they pass the exams and the project.

∀S (pass_exams(S) ∧ pass_proj(S) → pass_msc(S))

∀S (pass_msc(S) ← pass_exams(S) ∧ pass_proj(S))

*/

In Prolog:

% A *rule*:

```
pass_msc(S) :- pass_exams(S), pass_proj(S).
```

% Add a condition that S is an MSc student?

% A set of *facts*:

pass_exams(mary).

pass_proj(mary).

% A *rule:*

pass_msc(S) :- pass_exams(S), pass_proj(S).

:- corresponds to ←

, corresponds to ∧

Comments in Programs

% This is a comment, ignored by the compiler.

You can use % when the comment is short and runs on one line only.

Otherwise use /* …. */

/* Anything here is a comment */

How to read the rule

pass_msc(S) :- pass_exams(S),
pass_proj (S).

Declaratively:

Anyone who passes the exams and passes the project passes the MSc.

Procedurally:

There are two readings:

1.To show that someone passes the MSc:

a) show that they pass the exams, and

b) they pass the project.

2. To find who passes the MSc:

find who passes the exams and the project.

Demo

```
pass_msc(S) :- pass_exams(S), pass_proj(S).

pass_msc(peter).
pass_exams(john).
pass_exams(mary).
pass_exams(bob).
pass_exams(jill).

pass_proj(john).
pass_proj(mary).
```

Example Queries to the Program

```
| ?- pass_msc(mary).
   yes

| ?- pass_msc(X).
   X = john ? ;
   X = mary ? ;
   X = peter ? ;
   no
```

Who has passed the MSc?

| ?- pass_exams(X), \+ pass_msc(X).

Who has passed the exams but not the MSc?

X = bob ? ;

X = jill ? ;

no

| ?- pass_msc(john), pass_msc(mary).

Have john and mary both passed the MSc?

yes

Prolog syntax

A **Prolog program** is a sequence of ***clauses***.

A clause has the form:

H :- C_1, …, C_k. *conditional clause*

or **H.** *unconditional clause*

A terminating

‘.<space>’,

‘.<newline>’ or

‘.<tab>’

is *essential* after each clause.

Prolog syntax cntd. $H :- C_1,...,C_k.$

H and each C_i is an *atomic formula* of the form:

$p(t_1,..., t_n)$ or **p**

Must be NO space between p and the (

p is the predicate or relation name of the atomic formula. $t_1,..., t_n$ are *terms*.

Clause is *about* the predicate **p** of **H.**

Each C_i is sometimes referred to as a *call* or *condition*.

Later we will see that we can have more complex conditions.

Logical reading

A conditional clause

H :- C1,...,Ck. is read as:

$\forall X_1 \ldots X_m (H \leftarrow C_1 \wedge \ldots \wedge C_k)$

where the X_i are *all* the variables that occur in the clause,
or equivalently:

$\forall X_1, \ldots, X_i (H \leftarrow \exists X_{i+1}, \ldots, X_m (C_1 \wedge \ldots \wedge C_k))$

where $\mathbf{X_{i+1},..,X_m}$ are variables that only appear in the conditions of the clause.

(slide 24 of predicate logic part 2 set)

In Predicate Logic:

If X does not occur free in B then

$\forall X\ \forall Y\ (B \leftarrow A) \equiv \forall Y(B \leftarrow \exists X\ A)$

E.g. $\forall X,Y(has\_criminal\_record(Y) \leftarrow convicted\_for(Y, X))$

$\equiv$

$\forall Y(has\_criminal\_record(Y) \leftarrow \exists X\ convicted\_for(Y, X))$

An unconditional clause

H. is read as:

$\forall X_1 \ldots \forall X_m (H)$

where the X_i are *all* the variables that occur in **H.**

E.G.

beautiful(X). is read as

$\forall X$ beautiful(X)

Prolog terms

- *Constants* - usually alphanumeric sequence of one or more symbols beginning with a *lower case letter*, and possibly containing _

e.g. **bill**, **maryJones**, **mary_jones**, **diamond67**

- *Numbers* - usual syntax e.g. **3**, **-6**, **34.89**

- *Variable names* - alphanumeric sequence of one or more symbols beginning with an upper case letter or _ e.g. **X**, **Apple**, **_456**, **_**

- *Compound terms* - a *function* name (same syntax as constant) applied to n terms of the form ***f(t1,…,tn)***

E.g. Suppose we want to represent data on who the winner of our project prizes are.

We have a lot of choices.

We can use the function names below

name(First_name, Surname)

proj(Department, Degree, Year)

e.g. *proj(computing, msc, 2016)*

E.g. project prize winners

Using winner/2:

winner(name(alex, jones), proj(computing, msc, 2016)).

Using winner/6:

winner(alex, jones, proj, computing, msc, 2016).

Using winner_proj /5:

winner_proj(alex, jones, computing, msc, 2016).

Using winner_proj /4:

winner_proj(name(alex, jones), computing, msc, 2016).

Predicate names have the same syntax as constants, i.e.

alphanumeric sequence of one or more symbols beginning with a *lower case letter*, and possibly containing _

E.g. pass_msc

appointed

win2017

More on syntax

Constants, function symbols and predicate symbols can also be *any* sequence of characters in single quotes, e.g.

'fs@doc.ic.ac.uk'

'Sam '

'bill green'

'****'**

There are two other kinds of terms,

strings and

lists

(we will look at lists in detail later).

Facts and Rules

If an unconditional clause:

H.

contains *no* variables then the clause is called a **fact.**

E.g. **pass_exams(mary).**

no_of_children(john, 3).

All other Prolog clauses are called **rules.**

E.g.

drinks(john) :- anxious(john).

anxious(X):- has_driving_test(X).

covers(sky, X).

Prolog queries

A **query** is a conjunction of conditions, i.e.

?- C_1, . . . , C_n.<newline>

Each C_i is a condition/call (as in a clause).

?- is a prompt displayed by Prolog.

Terminating **.**<newline> is needed.

Prolog queries cntd

?- C_1, . . . , C_n .<newline>

If there are no vars in query, then the query is a request for a report on whether or not the query, as given, is a logical consequence of the program clauses.

E.g.

?- pass_msc(john).

Has john passed the MSc?

?- no_of_children(john, 3).

Does John have 3 children?

If the query **?- $C_1, \ldots, C_n$.** contains variables, the query is a request for a substitution (a set of term values) θ for the variables of the query such each of the conditions:

$C_1\theta, \ldots, C_n\theta$

is a logical consequence of the program clauses, or for a confirmation that there is no such θ.

$C_i\ \theta$ is **C_i** with any variable in **C_i** (given a value in θ**)** replaced by its assigned value.

C	θ	Cθ
p(X)	{X=john}	p(john)
q(X,Y)	{X=1, Y=2}	
q(X,Y)	{X=1, Y=f(Z)}	
q(X, Y)	{X=1, Y=f(X)}	
q(X, f(X))	{X=g(5)}	

Example query

?- pass_msc(X).

i.e. "Is there someone, X, who has passed the MSc?"
or "Who passed the MSc?"
It is a request for an answer

θ **={X=*name*}**

such that

pass_msc(X)θ

i.e. **pass_msc(*name*)**

follows from the program clauses *or*
for confirmation that there is no such θ (no such name).

Program:

```
pass_exams(mary).
pass_proj(mary).
pass_msc(S) :- pass_exams(S), pass_proj(S).
```


Query:

```
?- pass_msc(X).
```

Answer:

X=mary

Example Program
The Trade Program

```
sells(usa, grain, japan).
sells(Seller, P, Buyer) :-  produces(Seller, P), needs(Buyer, P).
produces(oman, oil).
produces(iraq, oil).
produces(japan, computers).
produces(germany, cars).
produces(france, iron).
needs(germany, iron).
needs(britain, cars).
needs(japan, cars).
needs(_, computers).
needs(Country, oil) :- needs(Country, cars).
```

Anonymous Variables

Variables that appear only once in a rule, can be *anonymous*, i.e. do not have to be named.

You can use _ (underscore) to denote such variables.

```
needs(_, computers).
happy(fs)  :-  likes(_, logic).
```


But be careful!

Two or more "_" in the same rule represent different variables.

```
really_happy(fs)  :- likes(_, logic), likes(_, prolog).
```

is understood as

```
really_happy(fs) :- likes(X, logic), likes(Y, prolog).
```

Demo

?-produces(oman, oil).

yes ‘yes’ means it follows from clauses

?-produces(X,oil).

X = oman; ‘;’ is request for another answer

X = iraq;

no ‘no’ means no more answers

?-produces(japan, X).

X = computers;

no

?-produces(X,Y).
X = oman, Y= oil;
X = iraq, Y= oil;
X = japan, Y= computers;
X = germany, Y= cars;
X = france, Y= iron;
no
?-produces(X, rice).
no
?-produces(britain, cameras).
no
?-produces(iraq, Y), needs(britain, Y).
Y = oil

```
| ?- sells(X, Y, britain).
| ?- sells(X, _, britain).
| ?- sells(_,_, britain).
```

Exercise: Trade Program

Write Prolog Queries for the following:

1. **Does Britain sell oil to the USA?**
2. **Who sells grain to who?**
3. **Who sells oil to Britain?**
4. **Who sells what to Germany?**
5. **Who sells something to Germany?**

Exercise Trade Program ctnd.

6. **Which two countries have mutual trade with one another?**

7. **Which two different countries have mutual trade with one another? (X\=Z means X and Z are different from one another.)**

8. **Express a prolog rule for "bilateral_traders(X,Z)" such that X and Z are two different countries that have mutual trade with one another.**

9. **Express the following query in Prolog.**

Who produces something that is needed by both Britain and Japan?

What answer(s) will Prolog give?

Scope of identifiers

- The scope of a variable is just the clause or query in which it occurs.

- The scope of any other name (constant, function name, predicate name) is the whole program and any query.

Example Program Work-Manager

```
% worksIn(Person, Department)
    worksIn(bill, sales).
    worksIn(sally, accounts).

% deptManager(Department, Manager)
    deptManager(sales, joan).
    deptManager(accounts, henry).

% managerOf(Worker, Manager)
    managerOf(joan, james).
    managerOf(henry, james).
    managerOf(james, paul).
```

Exercise

1. Define *colleague/2*, such that *colleague(W1,W2)* holds if W1,W2 are *different* workers that work in the same department.
2. Add a new clause for *managerOf(W,M)* to express that M is the manager of W if M is the manager of the department in which W works.

Disjunction in bodies of rules and queries

In Prolog **;** is the same as the logical symbol ∨.
E.g.

inelligible_to_vote(X) :-under_age(X) ; in_prison(X).

The Prolog rule

p:-c1;c2.

has the same meaning as the two rules

p:-c1.

p:-c2.

Exercise: Prove in logic that

p←c1∨ c2 ≡ (p ← c1)∧ (p ← c2).

So

inelligible_to_vote(X) :- under_age(X) ;
in_prison(X).

Can be written as:

inelligible_to_vote(X) :- under_age(X).

inelligible_to_vote(X) :- in_prison(X).

Arithmetic

- **is/2** is a primitive Prolog predicate for evaluating arithmetic expressions.
- The call
  X is Exp

where Exp is an arithmetic expression, *unifies* X with the value of Exp

- Operators work in the same way as in most languages + - * /

- X can be a number or an unbound variable but not another expression.
- Note that at the time of evaluation of condition
  X is Exp, Exp must be *ground*, i.e. contain no unbound vars.
- Arithmetic values can be compared using built in relations:
  <, =< , >, >=

Arithmatic Examples

- X is 2*4 (unifies/binds X to 8)
- W=4, U is 25*W, X is U/5
  (unifies/binds U to 100, and X to 20)
- X is 4, X is X+1 (will fail!)
- X is 4, NewX is X+1
  (unifies/binds NewX to 5)
- The difference between is and =.
  Try X is 2+1, Y=2+1.

X1 =:= X2

Succeeds if X1 and X2 evaluate to the same number.

X1 =\= X2

Succeeds if X1 and X2 do not evaluate to the same number.

Example: Factorial

The Factorial of a non-negative integer N,

denoted N!,

is the product of N and all the non-negative, non-zero integers below it.

O! = 1

1! = 1

2! = 2 * 1

3! = 3*2*1

4! = 4*3*2*1

N! = N*(N-1)*(N-2)* …*1 if N>0

N! = N*(N-1)! if N>0

In Prolog

Let *fact(N, FN)* stand for factorial of *N* is *FN*.

O! = 1	**fact(0,1).** *** we can also write this as:** **fact(N, FN):- N=0, FN=1.** ***/**
N! = **N*(N-1)!** if N>0	**fact(N, FN):-** **N>0,** **X is N-1,** **fact(X,FX),** **FN is N*FX.**

Example Uses

Find the factorial of a number

?- fact(4,X).

X=24

Check the factorial of a number

?- fact(3,6).

yes

Combined in any conjunction

?- fact(4, X), fact(5, Y), Y is 5*X.

X = 24, Y = 120 yes

Cannot use invertibly:

?- fact(X,2).

! Instantiation error in argument 1 of >/2

because the condition: **N > 0** needs **N** to be known.

trace / notrace

```
| ?- trace.
% The debugger will first creep -- showing everything
    (trace)yes% trace
```


See the difference between:


```
| ?- fact(2,X).
| ?- fact(X,2).
```



```
| ?- notrace.
```

```
1    1 Call: fact(2,_523) ?
2    2 Call: 2>0 ?
3    2 Exit: 2>0 ?
3    2 Call: _1162 is 2-1 ?
4    2 Exit: 1 is 2-1 ?
4    2 Call: fact(1,_1172) ?
5    3 Call: 1>0 ?
6    3 Exit: 1>0 ?
6    3 Call: _4519 is 1-1 ?
7    3 Exit: 0 is 1-1 ?
7    3 Call: fact(0,_4529) ?
8    3 Exit: fact(0,1) ?
8    3 Call: _1172 is 1*1 ?
9    3 Exit: 1 is 1*1 ? ?
4    2 Exit: fact(1,1) ?
9    2 Call: _523 is 2*1 ?
10   2 Exit: 2 is 2*1 ? ?
1    1 Exit: fact(2,2) ?
X = 2 ?
Yes
% trace
```

```
| ?- notrace.
1    1 Call: notrace ?
% The debugger is switched off
Yes
| ?-
```