Prolog Lecture 1

David Eyers

Michaelmas 2008

Notes derived from those created
by Andy Rice and Kate Taylor
used with permission

Course aims

Introduce a declarative style of programming

- Explain fundamental elements of Prolog: terms, clauses, lists, arithmetic, cuts, backtracking, negation

Demonstrate Prolog problem-solving techniques

By the end of the course you should be able to:

- Write and understand Prolog programs
- Use difference structures
- Understand basic constraint programming principles

Assessment

One exam question in Paper 3

Assessed Exercise (a tick)

- you must get a tick for either Prolog or C & C++
- tick exercises and submission done in Lent term
- more information to follow closer to the time

Supervision work

Some example questions are provided at the end of the lecture handout

- Note: Prolog examples are often easy to follow ...
- Make sure you can write your own programs too!

I will give some pointers and outline solutions during the lectures

- Make sure you understand the fundamentals well

Recommended text

"PROLOG Programming for Artificial Intelligence", Ivan Bratko, Addison Wesley (3rd edition, 2000)

- Provides an alternative angle on the basics
- Examines problem solving with Prolog in detail
- Not all of the textbook is directly relevant

Lecture 1

Logic programming and declarative programs

Introduction to Prolog

Basic operation of the Prolog interpreter

Prolog syntax: Terms

Unification of terms

Solving a logic puzzle

Imperative programming

Formulate a "how to compute it" recipe, e.g.:

- to compute the sum of the list, iterate through the list adding each value to an accumulator variable

```
int sum(int[] list ) {
    int result = 0;
    for(int i=0; i<list.length; ++i) {
      result += list[i];
    }
    return result;
}
```

Functional programming

Again formulate a "how to compute it" recipe

- Probably will need to do recursive decomposition

```
(* The sum of the empty list is zero and
 the sum of the list with head h and tail
 t is h plus the sum of the tail. *)

fun sum([])    = 0
  | sum(h::t) = h + sum(t);
```

Logic programming

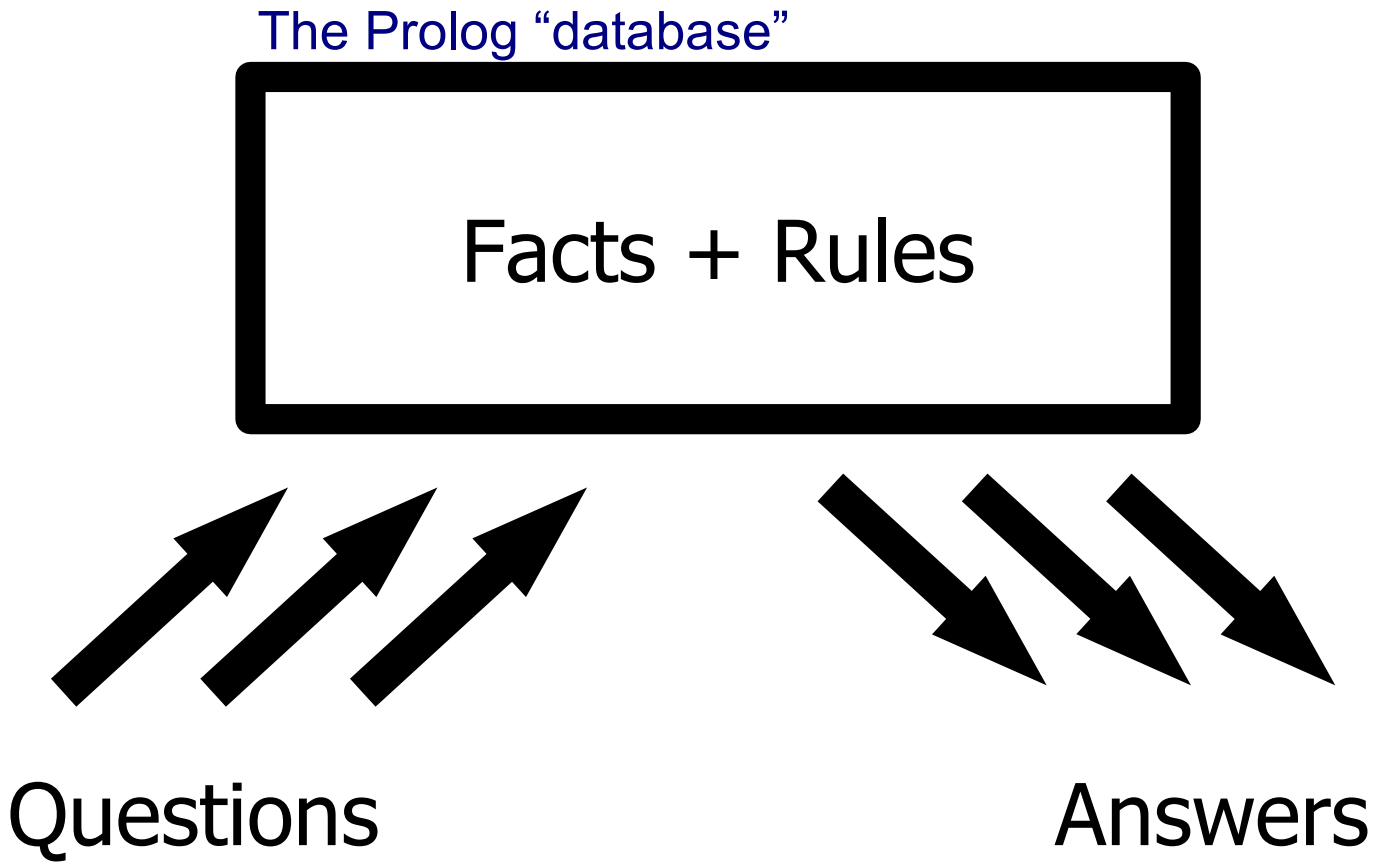
```
% the sum of the empty list is zero
sum([],0).

% the sum of the list with head H and
% tail T is N if the sum of the list T
% is M and N is M + H
sum([H|T],N) :- sum(T,M), N is M+H.
```

This is a declarative reading of a program (p23)

- Not "how to compute" the result
- Instead "this is true about the result"

Prolog programs answer questions



Prolog came from the field of Natural Language Processing

PROgramming en LOGique

Colmerauer, A., Kanoui, H., Roussel, P . and Pasero, R. “Un systeme de communication homme-machine en français”, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

Modern Prolog interpreters use the Warren Abstract Machine

WAM is like a logic programming virtual machine

- David H. D. Warren. "An abstract Prolog instruction set." Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Can significantly improve Prolog memory use

- Modern Prolog implementations will is the WAM
- ... or something similar to it

You are expected to use SWI-Prolog

Open-source (GPL) Prolog environment

- http://www.swi-prolog.org/
- Development began in 1987
- Available for Linux, MacOS X and Windows
- Fully featured, with many libraries

We will use SWI-Prolog throughout this course

- Get yourself a copy! (or at least access to one)
- Experiment with it!

SWI-Prolog is available to you

SWI-Prolog is installed on the PWF (5.6.38)

- Reboot a PWF terminal into Linux
- Log into linux.pwf.cam.ac.uk
- Log into the SRCF

You can easily install it on your own computer

- Linux users: use your package manager
  - (e.g. pl on Fedora Core, swi-prolog on Ubuntu)
- Otherwise consult the SWI-Prolog download page
  - Windows and MacOS binary installers are available
  - Build it from source

Prolog can answer simple questions directly from its database

```
> prolog                          .....or maybe pl on your system
?- [user].                        .....get ready to enter a new program
|: milestone(rousell,1972).
|: milestone(warren,1983).
|: milestone(swiprolog,1987).                      type [CTRL]-D
|: milestone(yourcourse,2008).                .....when done
|: % user://1 compiled 0.01 sec, 764 bytes
Yes
?- milestone(warren,1983).              .....ask it a question
Yes                                     .....the answer is “yes”
?- milestone(swiprolog,X).              .....let it find an answer
X=1987                                  .....the answer is 1987
Yes
?- milestone(yourcourse,2007).                .....ask it a question
No                                    .....the answer is “no”
?- halt.                              .....exit the interpreter
```

We will usually load Prolog programs from source files on disk

```
> cat milestone.pl          .....enter your program in a text file
milestone(rousell,1972).          (it should have a .pl extension)
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2008).
> prolog
?- [milestone].          .....instruct Prolog to load the program
?- milestone(warren,1983).
Yes
?- milestone(X,Y).                    .....find answers
X = rousell
Y = 1972 ;          .....you type a semi-colon (;) for more answers

X= warren
Y = 1983          .....you press enter when you've had enough
Yes
?- halt.
```

Our program is composed of facts and queries

```
> cat milestone.pl
milestone(rousell,1972).
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2008).
```

These are facts
(a particular type of clause)

```
> prolog
?- [milestone].
?- milestone(warren,1983).
Yes
?- milestone(X,Y).
X = rousell
Y = 1972 ;

X= warren
Y = 1983
Yes
?- halt.
```

These are queries
(and replies to the queries)

Using the Prolog shell

The Prolog shell at top-level only accepts queries

When a query result is being displayed:

- Press enter to accept a query answer and return to the top level shell
- Type a semi-colon (;) to request the next answer
- Type w to display fully a long result that Prolog has abbreviated

Terms are the building blocks with which Prolog represents data

Constants

Numbers:

1 -2 3.14

Atoms:

tigger

'100 Acre Wood'

Variables

X A_variable _

Compound terms

likes(pooh_bear,honey)

plus(4,mult(3,plus(1,9)))

Each term is either a constant, a variable, or a compound term. (p29)

Prolog can build compound terms from infix operators

Placing compound terms within compound terms builds tree structures:

- E.g. `html(head(title(blah)),body(...))`
- This is a prefix notation
  - i.e. the name of a tree node comes before its children

Prolog also supports infix expressions: (p31)

- e.g. `X-Y`, `2+3`, etc.
- Any infix expression has an equivalent prefix form
- Ensure that you are comfortable with this equivalence

You can ask Prolog to display any term using prefix notation alone

Infix notation is just for human convenience

- Does not affect Prolog's internal data structures
- Requires operator precedence to be defined
  - Otherwise terms such as x - y - z are ambiguous

The query write_canonical(*Term*) will display *Term* without using any infix operators

- Potentially useful for your Prolog experimentation
- We will gloss over how this actually works for now

Unification is Prolog's fundamental operation

Do these terms unify with each other?

1	a	a
3	a	A
5	tree(l,r)	A
7	tree(A,r)	tree(l,C)
9	A	a(A)

2	a	b
4	a	B
6	tree(l,r)	tree(B,C)
8	tree(A,r)	tree(A,B)
10	a	a(A)

Note: _ is a special variable that unifies with anything.

- Each _ in an expression unifies independently
  - (as if they were all unique, one-use named variables)

Zebra Puzzle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra?

(clearly some of the assumptions aren't explicitly stated above!)

Form a model of the situation

Represent each house with the term:

house(Nationality,Pet,Smokes,Drinks,Colour)

Represent the ordered row of houses as follows:

(H1,H2,H3,H4,H5)

We will show that the Zebra Puzzle can be solved using facts, unification and a single (large) query.

- More conventional Prolog programs will define predicates to help solve problems
- (we will talk about these soon)

Question

What sort of a term is:

house(Nationality,Pet,Smokes,Drinks,Colour)

a) number

b) atom

c) compound

d) variable

Question

What sort of a term is:

Nationality

a) number

b) atom

c) compound

d) variable

Question

What sort of a term is:

(H1,H2,H3,H4,H5)

a) number

b) atom

c) compound

d) variable

Define relevant facts

Let's consider one of the puzzle statements:

- The Englishman lives in the red house.
- That is: `house(british,_,_,_,red)`

This term must unify with one of the houses

- Simplify: let's say it unifies with the first house.
- The "houses" 5-tuple would then unify with term:
  - `(house(british,_,_,_,red),_,_,_,_)`

Generalise into a fact:

- firstHouse(HouseTerm,(HouseTerm,_,_,_,_)).
- Really we want "atLeastOneHouse" though...

Define relevant facts

To query two properties about the first house:

```
?- firstHouse(SomeCondition,Houses),
   firstHouse(OtherCondition,Houses).
```

- The comma requires both parts of the query to hold
- Prolog will progressively bind variables via unification
  - Including binding variables within compound terms

Prolog attempts to prove the query

- Variable bindings are a side-effect of the proof

However, we're usually specifically interested in what the variables actually get bound to!

Define relevant facts

Call our atLeastOneHouse fact "exists"

- i.e. there exists a house that has a certain property
- We discussed firstHouse(A,(A,_,_,_,_)).
- The generalisation to "at least one house" is:

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).
```

Feeling lost? Ask Prolog!

Test the `exists` predicate on simpler data:

```
?- exists(1,(1,2,3,4,5)).
```

Other things to try:

```
?- exists(2,(1,2,3,4,5)).
?- exists(A,(1,2,3,4,5)).
?- exists(4,(1,2,3,A,5)).
?- exists(1,apple).
?- exists(1,(1,2,3,4)).
```

More constraint-building facts

The facts we are defining allow us to encode the explicit constraints in the problem statement

- We are encoding just the red highlighted part:

6. The green house is immediately to the right of the ivory house.

```
rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).
```

More constraint-building facts

9. Milk is drunk in the middle house.

```
middleHouse(A,(_,_,A,_,_)).
```

10. The Norwegian lives in the first house.

```
firstHouse(A,(A,_,_,_,_)).
```

More constraint-building facts

11. The man who smokes Chesterfields lives in the house next to the man with the fox.

```
nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

Express the puzzle as one big query

2. The Englishman lives in the red house.

```
?- exists(house(british,_,_,_,red),Houses),
exists(house(spanish,dog,_,_,_),Houses),
exists(house(_,_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

Express the puzzle as one big query

3. The Spaniard owns the dog.

```
?- exists(house(british,_,_,_,red),Houses),
exists(house(spanish,dog,_,_,_),Houses),
exists(house(_,_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

Express the puzzle as one big query

6. The green house is immediately to the right of the ivory house.

```
?- exists(house(british,_,_,_,red),Houses),
exists(house(spanish,dog,_,_,_),Houses),
exists(house(_,_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

Including queries in your source file

Normal lines in the source file define new clauses

- We've used this to defining fact clauses so far...

Lines beginning with `:-` are "immediate" queries

- (that's a colon followed directly by a hyphen)
- Prolog executes those queries when the file is loaded
- We'll have more to say on this later...

The query `print(T)` prints out term `T` (in SWI)

- e.g. `print('Hello World').`

Zebra Puzzle

```
> prolog
?- [zebra].
norwegian
japanese
% zebra compiled 0.00 sec, 6,384 bytes
Yes
?- halt.
```

We use
`print(WaterDrinker),`
`print(ZebraOwner)`
in our query to produce this output

End

- Next lecture:
  - recursive reasoning,
  - lists,
  - arithmetic
  - and more puzzles...