Comp24412: Symbolic AI
Lecture 2: Prolog II

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17
Outline

Arithmetic

Functors and lists

Various goodies

Modifying the database

Conclusion
• Arithmetic is performed with the usual arithmetic operators and the predicate is.

?- X is 57+8.
X = 65

Yes

?- X is 57*8.
X = 456

Yes

?- X is cos(0.5).
X = 0.8775825618903727161

Yes
• Arithmetic operators cannot be used backwards!

?- 6 is Y + 7.
ERROR: Arguments are not sufficiently instantiated

• In other words, is is not really a logic programming construct

• There are several such extra-logical constructs in Prolog
Problem: write a program to compute

\[ \sum_{i=1}^{n} i^r \]

for positive integer \( n \) and arbitrary \( r \).

Solution idea (not quite right):

\[
\text{power_sum}(N,R,\text{Ans}) :- \\
\quad \text{N1 is } N - 1, \text{ power_sum}(\text{N1},R,\text{Ans1}), \\
\quad \text{Ans is } \text{Ans1} + N ^ R. \\
\text{power_sum}(1,\_R,1).
\]

Note the correspondence:

\[
\sum_{i=1}^{n} i^r = \sum_{i=1}^{n-1} i^r + n^r \\
\text{Ans is } \text{Ans1} + N ^ R.
\]
The following program works

% Computes sum of all R^i for i from % i=1 to i=N
power_sum(N,R,Ans):-
    N > 1, N1 is N - 1,
    power_sum(N1,R,Ans1),
    Ans is Ans1 + N ^ R.
power_sum(1,_R,1).

Some Prologs use ** in place of ^. SWI Prolog allows both.
The underscored variable _R means we don’t care about its value.
• In action:
  
  `?- power_sum(3, 2, L)`

  \[ L = 14 \]

  Yes

  `?- power_sum(13, 21.56, L)`

  \[ L = 1.25638e24 \]

  Yes

• What happens if you call `power_sum` with a non-integer first argument?
• Here is a more familiar program

```prolog
factorial(N,F):-
    N > 0,
    N1 is N - 1,
    factorial(N1,F1),
    F is N * F1.

factorial(0,1).
```

• In action:

```
?- factorial(9, L).

L = 362880
Yes
```
• We are not limited in Prolog to simple names, e.g. noel, sue, chris
• We can also have complex names, e.g. couple(sue,chris)
• And we could express a family database as:
  parents(couple(sue,chris),noel).
  parents(couple(cheryl,noel),catherine)
  parents(couple(ann,dave),chris).

• And then define mother as:
  mother(X,Y):- parents(couple(X,Z),Y).
• Some operators are written in **infix** form.
• e.g. 6 + 7 as an alternative to +(6,7)
• Note, however, that 6 + 7 is a complex term, just like couple(mary,peter)
• What happens if you type the following queries?
  
X = 6 + 7.
X is 6 + 7.
Outline

Arithmetic

Functors and lists

Various goodies

Modifying the database

Conclusion
• In a compound term such as `couple(sue, chris)`, `couple` is the *functor* and `sue` and `chris` are its *arguments*.

• Similarly, in `6 + 7`, `+` is the functor and `6` and `7` are the arguments.

• Warning: do not confuse functors with predicates.
  • Predicates are used to make statements
  • Functors are used to refer to (complex) objects
• The most commonly used data-structure in Prolog is the list.
• A list in Prolog is denoted by a sequence of its elements, separated by commas, and enclosed in brackets, [ ], e.g.
  
  [a,b,c,d]
  [[a,b], c, [d,e,f]]
  [[[[[[a]]]]]]
• The empty list is denoted [].

• Lists containing variables unify with each other just like any other Prolog terms:

?- [a, b, c]=[X, b, Y].

X = a, Y = c

?- [a, b, [c, d]]= [X, b, Y]

X = a, Y = [c, d]

?- [a, b, [c, Z]]= [X, b, [Y, X]]

Z = a, X = a, Y = c
• In addition, Prolog syntax allows the notation \(|\) to separate out the head and tail of a list:

?– [a, b, c, d, e]=[X| Y]

X = a, \( Y = [b, c, d, e] \)

?– [a, b, c, d, e]=[X, Y| Z]

X = a, \( Y = b, \quad Z = [c, d, e] \)

?– [X, b, c, d, e]=[a| Y]

X = a, \( Y = [b, c, d, e] \)
- Actually, lists are really quite ordinary Prolog data-structures with a special notation.
- the term \([a,b,c,d]\) is really 
  \(\text{.}(\text{.}(\text{.}(\text{.}(\text{.}(a, \text{.}(\text{.}(\text{.}(b, \text{.}(\text{.}(\text{.}(c, \text{.}(\text{.}(\text{.}(d, [])])])])])])])\)
  as we can see:
  \[
  \text{?- .'}(a, .'(b, .'(c, .'(d, [])))) = X.
  \]
  \(X = [a,b,c,d]\)
  (The quotes are so as not to confuse the parsing process.)
• As in other programming languages, lists can be pictured as special sorts of structures:
• Lists are normally operated on using recursive predicates.
• The following tests whether an element is in a list
  
  member(X,[X|_L]).
  member(X,[_Y|L]):-
      member(X,L).

• Note the (optional) use of underscored variables
• Aside: like most list-predicates, member/2 is pre-defined in SWI-Prolog
• Again, there is an obvious correspondence between the facts
  \[ \forall x \forall z \text{member}(x, x \cdot z) \]
  \[ \forall x \forall y \forall z (\text{member}(x, z) \rightarrow \text{member}(x, y \cdot z)) \]

  and the Prolog program

  \text{member}(X, [X|Z]).
  \text{member}(X, [Y|Z]) :- \text{member}(X,Z).
• This predicate works:

```prolog
:- member(c, [a, b, c, d, e]).

yes

:- member(f, [a, b, c, d, e]).

no
```
Here is the program again:

```prolog
member(X,[X|_L]).
member(X,[_Y|L]):-
    member(X,L).
```

and here is a trace of the first query:

```prolog
call member(c, [a, b, c, d, e])
UNIFY 2 []
call member(c, [b, c, d, e])
UNIFY 2 []
call member(c, [c, d, e])
UNIFY 1 []
exit  member(c, [c, d, e])
ext  member(c, [b, c, d, e])
ext  member(c, [a, b, c, d, e])
```
• Another example:

  \[
  \text{append}([X|L1],L2,[X|L3]):- \\
  \text{append}(L1,L2,L3).
  \]

  \[
  \text{append([],L,L)}.
  \]

• Pictorially:
in operation:

:- append([a, b, c], [1, 2, 3], L).

L = [a, b, c, 1, 2, 3]
• And another
  length([],0).

  length([X|L],N):-
      length(L,N1), N is N1 + 1.

• in operation:
  :- length([a, b, c], N).

  N = 3

  :- length([[a, b, c]], N).

  N = 1
• And another

\[
\text{rev1}([X|L], \text{L}_\text{ans}) : - \\
\text{rev1}(L, \text{L}_\text{ans}1), \\
\text{append}(\text{L}_\text{ans}1, [X], \text{L}_\text{ans}).
\]

\[
\text{rev1}([], []). 
\]

• Pictorially:
• in operation:
  :- \texttt{rev1([1, 2, 3], L)}.

  \texttt{L = [3, 2, 1]}

  :- \texttt{rev1([[1, 2, 3]], L)}.

  \texttt{L = [[1, 2, 3]]}
• Often we need to do something to each member of a list

```prolog
my_operation(X,Y):-
    Y is X^3 - 5.
```

```prolog
list_my_operation([X|L1],[Y|L2]):-
    my_operation(X,Y),
    list_my_operation(L1,L2).
```

```prolog
list_my_operation([],[]).
```

• In operation:

```prolog
?- list_my_operation([1,2,3],L).
```

L = [-4, 3, 22]
• Want to make things easy on yourself?
• Then use the pre-defined predicate maplist
  
  \[- \text{maplist(my\_operation, [1, 2, 3], L).}\]

  \[L = [-4, 3, 22]\]

• **Warning**: SWI-specific
• This predicate actually has some tricky features!
  
  \[ triple_{op}(Z,Y,X,\text{Result}) :- \]
  \[ \text{Result is } X + Y^2 + Z^3. \]

• Now we can supply the third argument from a list

\[- \text{ maplist}(\text{triple}_{op}(1,2),[1,2,3],L). \]

\[ L = [6, 7, 8] \]

Check this!
Outline

Arithmetic

Functors and lists

Various goodies

Modifying the database

Conclusion
The predicate =.. (pronounced univ for universal) converts complex terms into lists

?- parent(sue, noel) =.. List.
List = [parent, sue, noel] ?
yes

and back again:

?- Term =.. [parent, sue, noel].
Term = parent(sue, noel) ?
yes
• Strings in Prolog are lists of ASCII

?- "Ian" = L.

L = [73, 97, 110]

?- "Ian" = [H|T].

H = 73
T = [97, 110]
• `name/2` – converts between atoms and their strings

```prolog
?- name(cat,String).
String = [99, 97, 116] ;
No
?- name(Atom,[99, 97, 116]).
Atom = cat ;
No
```
• Example: pluralizing and singularizing

pluralizer(WS,WP):-
    name(WS,WSChars),
    append(WSChars,"s",WPChars),
    name(WP,WPChars).

singularizer(WP,WS):-
    name(WP,WPChars),
    append(WSChars,"s",WPChars),
    name(WS,WSChars).

Note: both are needed.
Why can’t we simply singularize by:

?- pluralizer(S,cats).
Here is an alternative way of pluralizing:

```prolog
pluralizer(WS, WP):-
    name(WS, WSChars),
    reverse(WSChars, WSCharsRev),
    reverse([115|WSCharsRev], WPChars),
    name(WP, WPChars).
```
• Useful testing predicates
  • var/1: a variable?
  • atom/1: an atom?
  • atomic/1: an atom (including numbers)?
  • compound/1: a term that is neither a constant nor a variable
  • is_list/1: any list?
  • proper_list/1: a list whose tail is non-empty?
• Input is a bit grim in Prolog
• Output is even grimmer
• This is a good point to advertise the SWI Prolog manual: http://www.swi-prolog.org, and the online manual (under the help menu in the SWI window).
• The best way to see how output is done is to look at some examples—for example, the auxiliary files for the labs.
• It is often important to test for equality *without* doing unification.

• The predicate used to do this is `==`

4 ?- A = b.
A = b
Yes
5 ?- A == b.
No
6 ?- A == B.
No
7 ?- A == A.
A = _G186
Yes
• The pre-defined predicate `setof` is useful.

• Suppose we have:

  ```
  ?- parent(X,noel).
  X = sue ;
  X = chris ;
  No
  ```

• Then we will also have

  ```
  ?- setof(X,parent(X,noel),List).
  X = _1
  List = [chris, sue] ;
  No
  ```
• Suppose we have:

?- parent(X,Y).
X = sue
Y = noel ;

X = chris
Y = noel ;

X = noel
Y = ann ;

No
• Then we will also have

?- setof(parent_of(X,Y),parent(X,Y),List).
X = _1
Y = _2
List = [parent_of(chris, noel),
        parent_of(noel, ann),
        parent_of(sue, noel)];

No
• Finally, compare

?- setof(X, parent(X,Y), List).
X = _1
Y = ann
List = [noel] ;

X = _1
Y = noel
List = [chris, sue] ;

No

• Note that separate solutions are given for the various possible values of Y.
• But suppose we wanted a list of all values of X such that there exists a Y such that parent(X,Y) succeeds.

• To do this, we use the syntax Y^:

  ?- setof(X,Y^parent(X,Y),List).
  X = _1
  Y = _2
  List = [chris, noel, sue] ;

  No
Incidentally, you can make your own infix operators using a call to the predicate `op/3`, thus:

20  ?- op(300, xfy, :).
Yes
21  ?- X = ’:’(sue, chris).
X = sue:chris
Yes
23  ?- A:B = a:b:c.
A = a
B = b:c ;
No
Outline

Arithmetic

Functors and lists

Various goodies

Modifying the database

Conclusion
• Other Prolog “funnies”:
  • asserta/1
  • assertz/1
  • retract/1

• For example, here is a program that writes the times tables up to 12.

```prolog
timesTable:-
  L = [1,2,3,4,5,6,7,8,9,10,11,12],
  member(X,L),
  member(Y,L),
  Z is X * Y,
  asserta(timesTable(X,Y,Z)),
  fail.

timesTable.
```
• In action

1  ?- timesTable.
   true.

2  ?- timesTable(2,Y,24).
   Y = 12 ;
   false.

3  ?- timesTable(11,11,Z).
   Z = 121 ;
   false.

• These predicates modify the program currently being written.
• They are not really in the spirit of logic programming, and should be used sparingly.
• You will, however, need them for your second lab.
Outline

Arithmetic

Functors and lists

Various goodies

Modifying the database

Conclusion
• We went through a lot of tedious Prolog stuff.

• What should I do next?
  • Read *Learn Prolog Now!* Chh 4 and 5, now!
  • Also read Chh. 6, 10 for next lecture.
  • Acquire a copy of *Representation and Inference for Natural Language*, and start reading Chh. 1 and 5.
  • Make sure you try out Prolog under Linux!