Künstliche Intelligenz

2. Einführung in Prolog
Teil 2

Dr. Claudia Schon
schon@uni-koblenz.de

Arbeitsgruppe Künstliche Intelligenz
Universität Koblenz-Landau
Die folgenden Folien stammen aus dem ESSLI 2004 Kurs „Prolog programming: a do-it-yourself course for beginners“ von Kristina Striegnitz:
http://cs.union.edu/~striegnk/courses/esslli04prolog/
Recursive Predicate Definitions
Ancestors

Task: Define a predicate \texttt{ancestor.of}(X,Y) which is true if X is an ancestor of Y.
Ancestors (cont.)

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).

→ Doesn’t work for ancestor_of; don’t know “how many parents we have to go back”.

ancestor_of(X,Y) :- parent_of(X,Y).

People are ancestors of their children,

ancestor_of(X,Y) :- parent_of(X,Z), ancestor_of(Z,Y).

and they are ancestors of anybody that their children may be ancestors of (i.e., of all the descendants of their children).
Ancestors (cont.)

\[
\text{grandparent} \_ \text{of}(X,Y) \ :- \ \text{parent} \_ \text{of}(X,Z), \ \text{parent} \_ \text{of}(Z,Y).
\]

\[
\text{greatgrandparent} \_ \text{of}(X,Y) \ :- \ \text{parent} \_ \text{of}(X,Z), \ \text{parent} \_ \text{of}(Z,A), \ \text{parent} \_ \text{of}(A,Y).
\]

\[
\text{greatgreatgrandparent} \_ \text{of}(X,Y) \ :- \ \text{parent} \_ \text{of}(X,Z), \ \text{parent} \_ \text{of}(Z,A), \ \text{parent} \_ \text{of}(A,B), \ \text{parent} \_ \text{of}(B,Y).
\]

\[\Rightarrow \text{Doesn’t work for ancestor} \_ \text{of; don’t know “how many parents we have to go back”}].

\[
\text{ancestor} \_ \text{of}(X,Y) \ :- \ \text{parent} \_ \text{of}(X,Y).
\]

People are ancestors of their children,

\[
\text{ancestor} \_ \text{of}(X,Y) \ :- \ \text{parent} \_ \text{of}(X,Z), \ \text{ancestor} \_ \text{of}(Z,Y).
\]

and they are ancestors of anybody that their children may be ancestors of (i.e., of all the descendants of their children).
Matching and Proof Search
Example 1

KB: wizard(harry).
wizard(ron).
wizard(hermione).
muggle(uncle_vernon).
muggle(aunt_petunia).
chases(crookshanks, scabbars).

Query: ?- wizard(hermione).

yes

Easy: wizard(hermione) is a fact in the knowledge base.
Example 2

KB: wizard(harry).
    wizard(ron).
    wizard(hermione).
    muggle(uncle_vernon).
    muggle(aunt_petunia).
    chases(crookshanks,scabbers).

Query: ?- wizard(X).
        X = harry ;
        X = ron ;
        X = hermione ;
        no

• The query `wizard(X)` matches the fact `wizard(harry)`. This instantiates the variable `X` with `harry`.

• It also matches the facts `wizard(ron)` and `wizard(hermione)`. 
Matching

- Two atoms match if they are the same atom.
  Ex.: harry = harry, but harry \( \neq \) ‘Harry’.

- A variable matches any other Prolog term. The variable gets instantiated with the other term.
  Ex.: X = wizard(harry)

- Two complex terms match if they have the same functor and the same number of arguments and if all pairs of parallel arguments match.
  Ex.: like(harry,hargrid) = like(harry,X)
  Ex.: like(harry,hargrid) \( \neq \) like(harry,X,Y)
  Ex.: like(harry,hargrid) \( \neq \) like(X,X)
Matching

- Two atoms match if they are the same atom.
  Ex.: harry = harry, but harry \(\neq\) ‘Harry’.

- A variable matches any other Prolog term. The variable gets instantiated with the other term.
  Ex.: \(X = \text{wizard}(harry)\)
  Ex.: \(x = y\)
Matching

- Two atoms match if they are the same atom.
  Ex.: harry = harry, but harry \( \not= \) ‘Harry’.

- A variable matches any other Prolog term. The variable gets instantiated with the other term.
  Ex.: \( x = \text{wizard(harry)} \)
  Ex.: \( x = y \)

- Two complex terms match if they have the same functor and the same number of arguments and if all pairs of parallel arguments match.
  Ex.: \( \text{like(harry,hargrid)} = \text{like(harry,X)} \)
  Ex.: \( \text{like(harry,hargrid)} \not= \text{like(harry,X,Y)} \)
  Ex.: \( \text{like(harry,hargrid)} \not= \text{like(X,X)} \)
Back to Example 2

**KB:** wizard(harry).
    wizard(ron).
    wizard(hermione).
    muggle(uncle_vernon).
    muggle(aunt_petunia).
    chases(crookshanks,scabbars).

**Query:** ?- wizard(X).
          X = harry ;
          X = ron ;
          X = hermione ;
          no

- Prolog checks for facts that **match** the query. (There are three.)
- Prolog starts from the top of the knowledge base and, therefore, finds **wizard(harry)** first.
- Typing ; forces Prolog to check whether there are other possibilities.
Example 3

KB: eating(dudley).

happy(aunt_petunia) :- happy(dudley).
happy(uncle_vernon) :- happy(dudley), unhappy(harry).
happy(dudley) :- kicking(dudley,harry).
happy(dudley) :- eating(dudley).

kicking(dudley,ron).
unhappy(ron).

Query: ?- happy(aunt_petunia).
    yes

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB: eating(dudley).

\[
\begin{align*}
& \text{happy(aunt\_petunia)} :\!\!\! - \text{happy(dudley)}. \\
& \text{happy(uncle\_vernon)} :\!\!\! - \text{happy(dudley)}, \text{unhappy(harry)}. \\
& \text{happy(dudley)} :\!\!\! - \text{kicking(dudley,harry)}. \\
& \text{happy(dudley)} :\!\!\! - \text{eating(dudley)}.
\end{align*}
\]

kicking(dudley,ron).
unhappy(ron).

Query: \text{?- happy(aunt\_petunia)}. \\
\text{yes}

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB: eating(dudley).

happy(aunt_petunia) :- happy(dudley).
happy(uncle_vernon) :- happy(dudley), unhappy(harry).
happy(dudley) :- kicking(dudley,harry).
happy(dudley) :- eating(dudley).

kicking(dudley,ron).
unhappy(ron).

Query: ?- happy(aunt_petunia).
yes

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB: eating(dudley).

happy(aunt_petunia) :- happy(dudley).
happy(uncle_vernon) :- happy(dudley), unhappy(harry).
happy(dudley) :- kicking(dudley, harry).
happy(dudley) :- eating(dudley).

kicking(dudley, ron).
unhappy(ron).

Query: ?- happy(aunt_petunia).

yes

- Check for a fact or a rule's head that match the query.
- If you find a fact, you're done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB: eating(dudley).

happy(aunt_petunia) :- happy(dudley).
happy(uncle_vernon) :- happy(dudley), unhappy(harry).
happy(dudley) :- kicking(dudley,harry).
happy(dudley) :- eating(dudley).
kicking(dudley,ron).
unhappy(ron).

Query: ?- happy(aunt_petunia).
yes

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB: eating(dudley).

happy(aunt_petunia) :- happy(dudley).
happy(uncle_vernon) :- happy(dudley), unhappy(harry).
happy(dudley) :- kicking(dudley,harry).
happy(dudley) :- eating(dudley).

kicking(dudley,ron).
unhappy(ron).

Query: ?- happy(aunt_petunia).

yes

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 3

KB:
- eating(dudley).
- happy(aunt_petunia) :- happy(dudley).
- happy(uncle_vernon) :- happy(dudley), unhappy(harry).
- happy(dudley) :- kicking(dudley,harry).
- happy(dudley) :- eating(dudley).
- kicking(dudley,ron).
- unhappy(ron).

Query: ?- happy(aunt_petunia).

yes

- Check for a fact or a rule’s head that match the query.
- If you find a fact, you’re done.
- If you find a rule, prove all goals specified in the body of the rule.
Example 4

KB:

- eating(dudley).
- happy(aunt_petunia):-happy(dudley).
- happy(uncle_vernon):-happy(dudley),unhappy(harry).
- happy(dudley):-kicking(dudley,harry).
- happy(dudley):-eating(dudley).

Query:  ?- happy(X).
Example 5

father(albert, james).
father(james, harry).
mother(ruth, james).
mother(lili, harry).

wizard(lili).
wizard(ruth).
wizard(albert).
wizard(X) :-
  father(Y, X),
  wizard(Y),
  mother(Z, X),
  wizard(Z).
Ancestors (cont.)

parent_of(paul,petunia).
parent_of(helen,petunia).
parent_of(paul,lili).
parent_of(helen,lili).
parent_of(albert,james).
parent_of(ruth,james).
parent_of(petunia,dudley).
parent_of(vernnon,dudley).
parent_of(lili,harry).
parent_of(james,harry).

ancestor_of(X,Y) :-
    parent_of(X,Y).
ancestor_of(X,Y) :-
    parent_of(X,Z),
    ancestor_of(Z,Y).

ancestor_of(albert,harry)

parent_of(albert,harry)

parent_of(albert,I1)
ancestor_of(I1,harry)
I1=james

ancestor_of(james,harry)

parent_of(james,harry)
Practical Session

- Matching
- Proof Search
- Recursion

Take a look at the exercises at:
http://cs.union.edu/~striegnk/courses/esslli04prolog/practical.day2.php
Arithmetic
Arithmetic in Prolog

?- 3+5 = +(3,5).
yes

?- 3+5 = +(5,3).
no

?- 3+5 = 8.
no

- 3+5 is a normal complex term.
- Prolog has to be told explicitly to evaluate it as an arithmetic expressions.
- This done using certain built-in predicates, such as `is/2`, `=/2`, `/2`, etc.
The built-in predicate is

?- X is 3+5.
X = 8 ;
no
?- X is 30-4.
X = 26 ;
no
?- X is 3*5.
X = 15 ;
no
?- X is 9/4.
X = 2.25 ;
no

Attention:
?- 3+5 is 8.
no
?- 8 is 3+X.
ERROR: Arguments are not sufficiently instantiated
?- X = 4, 8 is 3+X.
no
‘=’ vs. ‘is’

?- X = 2 * 5.

?- X is *(2,5).
‘=’ vs. ‘is’

?- X = 2 * 5.
X = 2 * 5

?- X is *(2,5).
‘=’ vs. ‘is’

?- X = 2 * 5.
X = 2 * 5

?- X is *(2,5).
X = 10
Lists
Lists

- Intuitively: sequences or enumerations of things
- In Prolog: a special kind of data structure, i.e., special kinds of Prolog terms
Prolog lists

Prolog lists either look like this:

the empty list: [ ]

or like this:

non-empty lists: .(Head, Tail)

a Prolog term;

i.e., an atom (dobbey), a variable (x), a complex term (house_elf(dobbey)), a list,
a number

i.e., the empty list [ ] or a non-empty list of the form .(Head, Tail)
Examples of lists and non-lists

Lists:

[ ]
  . (a, [ ]): this list has one element
  . (a, . (b, [ ])): this list has two elements
  . (b, . (a, [ ])): this list also has two elements
  . (. (a, [ ]), . (b, [ ])): this list also has two elements. The first element is the list . (a, [ ]), and the second element is the atom b.

No lists:

. ([ ])
  . (a, b)
Another way of writing lists in Prolog

- \((a, \text{Tail}) = [a|\text{Tail}]\)
- \((a,.(b,\text{Tail})) = [a,b|\text{Tail}]\)
- \((a,.(b,.(c,[]))) = [a,b,c]\)
Working with lists (1)

trans_a_b/2: a predicate for “translating” a list of as into a list of bs.

trans_a_b(X,Y) should be true if X, the ‘input’, is a list of as and Y, the ‘output’, is a list of bs which has just as many bs as the input has as.

\[
\text{trans}_a_b([],[]).
\]

If the input is empty, then the output is empty as well.

\[
\text{trans}_a_b([a|\text{InputTail}], [b|\text{OutputTail}]) :-
\text{trans}_a_b(\text{InputTail}, \text{OutputTail}).
\]

Otherwise the first a in the input list has to correspond to a b in the output list. The tail of the output list has to be the “translation” of the input list.
Working with lists (2)

element_of/2: a predicate for testing whether a list contains a given Prolog term.

element_of(X,Y): should be true if X is an element of Y.

\[
element_of(X, [X|Tail]).
\]

If the first element of the list is the one that we are looking for, we are done.

\[
element_of(X, [\_|Tail]) :- element_of(X,Tail).
\]

Otherwise, check whether the term we are looking for is in the tail of the list.

In SWI-Prolog element_of/2 is predefined under the name member.
Working with lists (3)

concatenate/3: a predicate for concatenating two lists.

concatenate(X, Y, Z) should be true if Z is the concatenation of X and Y; for example, concatenating [a] with [b, c] yields [a, b, c].

\[
\text{concatenate([],L,L).} \\
\text{Concatenating the empty list with any other list } L \text{ yields } L.
\]

\[
\text{concatenate([Head|Tail],L,[Head|NewTail]) :-} \\
\text{concatenate(Tail,L,NewTail).}
\]

Otherwise, the first element of the output list has to be the same as the first element of the first input list. And the tail of the output list is the concatenation of the tail of the first input list with the second input list.
Concatenating lists

concatenate([Head|Tail],L,[Head|NewTail]) :-
    concatenate(Tail,L,NewTail).

In SWI-Prolog concatenate/3 is predefined under the name append.
Prolog predicates can be used in many ways

?- trans_a_b([a,a,a],L).
L = [b,b,b] ;
no

?- trans_a_b([a,a,a],[b]).
no

?- trans_a_b(L,[b,b]).
L=[a,a] ;
no

?- member(a,[a,b,c]).
yes

?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no

?- member(a,L).
L = [a|__G280] ;
L = [__G279, a|__G283] ;
L = [__G279, __G282, a|__G286] ;
L = [__G279, __G282, __G285, a|__G289]
Yes
Practical Session

Take a look at the exercises „Getting Started“ and „Some easy list exercises“ at:

http://cs.union.edu/~striegnk/courses/esslli04prolog/practical.day3.php