Logic programming I

Henrik Boström
Stockholm University

- Facts
- Queries
- Rules
- Terms
- Recursion
- Lists
- Negation

Logic programs

A *logic program* consists of *facts* and *rules*.
A logic program is executed by responding to *queries*.

Facts

A *fact* is a proposition that a certain relation (predicate) holds between certain objects.

predicate_name(A_1, ..., A_n).

father(abraham,isaac).

Database with some biblical relations

```
father(terach,abraham).
father(terach,nachor).
father(terach,haran).
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

mother(sarah,isaac).
```

```
male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).

female(sarah).
female(milcah).
female(yiscah).
```

Queries

A *query* is a request for an answer to whether or not a certain relation holds for certain objects.

```
?- father(abraham,isaac).
   yes
?- father(abraham,lot).
   no
```

A query for a certain program is answered by looking for whether what is asked is a logical consequence or not.

Variables

A *variable* is an unspecified object.

```
?- father(abraham,X).
   X = isaac
```

Variables are written with an initial capital letter.

Constants are written with lower case letters only.

Existential queries

Variables in queries assume existential quantifiers.

```
?- father(abraham,X).
```

is interpreted as:

"Is there an X such that Abraham is father to X?"

A query may have more than one answer:

```
?- father(haran,X).
   X = lot;
   X = milcah;
   X = yiscah;
   no
```

Conjunctive queries

A *conjunctive* query is written:

?- Q_1, …, Q_n.

where each Q_i is a simple (atomic) query.

```
?- father(abraham,isaac), male(lot).
   yes
?- mother(X,Y), male(Y).
   X = sarah Y = isaac
?- mother(X,Y), male(X).
   no
```

Universal facts

Everyone likes pomegranates is written:

likes(X,pomegranates).

Everyone likes everyone is written :

likes(X,Y).

Everyone likes themselves is written :

likes(X,X).

Rules

A *rule* is written on the (Horn clause) form:

A :- B_1, ..., B_n.

A is called the head

B_1, ..., B_n is called the body

```
grandfather(X,Y):-
   father(X,Z),
   father(Z,Y).
```

Rules

```
grandparent(X,Y):- father(X,Z), father(Z,Y).
grandparent(X,Y):- father(X,Z), mother(Z,Y).
grandparent(X,Y):- mother(X,Z), father(Z,Y).
grandparent(X,Y):- mother(X,Z), mother(Z,Y).
```

```
grandparent(X,Y):- parent(X,Z), parent(Z,Y).
parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).
```

```
?- grandparent(terach,isaac).
```

Rules

```
brother(Brother,Sibling):-
    parent(Parent,Brother),
    parent(Parent,Sibling),
    male(Brother).
```

```
?- brother(lot,lot).
yes
```

```
brother(Brother,Sibling):-
    parent(Parent,Brother),
    parent(Parent,Sibling),
    male(Brother),
    Brother \== Sibling.
```

Terms

A *term* is either a variable, a constant, or a compound term, where a compound term is on the form: $f(t_1, \ldots, t_n)$, where f is a constant and $t_1, \ldots, t_n$ are terms.

```
name(henrik)
s(s(0))
f(X,Y)
list(a,list(b,list(c,nil)))
```

A *grounded* term is a term that does not contain variables.

Types

A *type* is a (finite or infinite) set of terms.

A type may be defined by a unary relation:

```
female(sarah).
female(milcah).
female(yiscah).
```

Recursive logic programs may define infinite types.

Arithmetics

Natural numbers: 0, 1, 2, 3, ...

```
0, s(0), s(s(0)), s(s(s(0))), ...

natural_number(0).
natural_number(s(X)):- natural_number(X).

?- natural_number(s(s(s(0)))).
yes
?- natural_number(N).
N = 0;
N = s(0);
N = s(s(0));
   ...
```

Arithmetic predicates

```
less_or_equal(0,X):- natural_number(X).
less_or_equal(s(X),s(Y)):- less_or_equal(X,Y).

?- less_or_equal(s(s(0)),s(s(s(0)))).

plus(0,X,X):- natural_number(X).
plus(s(X),Y,s(Z)):- plus(X,Y,Z).

?- plus(s(s(0)),s(s(0)),S).
```

Lists

A *list* is either the empty list [] or a binary, compound term .(H,T) where H is an element and T is a list.

A list .(H,T) can preferably be written [H|T].
A list with the elements a, b and c may be written as:

```
.(a, .(b, .(c, [])))
[a|[b|[c|[]]]]
[a,b,c]
```

List predicates

```
first(X,[X|L]).

?- first(a,[a,b,c]).
yes
?- first(X,[d,e,f]).
X = d

last(X,[X]).
last(X,[Y|L]):- last(X,L).

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

More list predicates

```
member(X,[X|L]).
member(X,[Y|L]):- member(X,L).

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

Even more list predicates

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).

?- append([a,b,c],[d,e,f],L).
L = [a,b,c,d,e,f]
?- append(L,[b,c,d],[a,b,c,d]).
L = [a]
```

```
member(X,L1):- append(L2,[X|L3],L1).
```

Defining recursive predicates

Task: define a predicate *delete* that holds for triplets (E,L1,L2) where L2 is the list that is obtained by removing all occurrences of E in L1.

```
?- delete(a,[a,b,b,a],L).
L = [b,b]
```

Three cases:

1. L1 = empty list
2. E = first element in L1.
3. E <> first element in L1.

Defining recursive predicates

Case 1:

```
delete(X,[],[]).
```

Case 2:

```
delete(X,[X|Xs],Ys):-
   delete(X,Xs,Ys).
```

Case 3:

```
delete(X,[Y|Xs],[Y|Ys]):-
   X\==Y,
   delete(X,Xs,Ys).
```

Negation as failure

\+ G is considered to be true if G cannot be proven to be true, otherwise \+ G is considered to be false.

```
?- \+ member(a,[a,b,c]).
   no
```

```
married(bill).
student(bill).
student(joe).
unmarried_student(X):-
   \+ married(X),
   student(X).
```

SICStus Prolog

```
SICStus 3.12.7 (x86-win32-nt-4): Fri Oct  6 00:15:14 WEST 2006
Licensed to dsv.su.se
| ?- [my_file].                     % load my_file
    {consulting my_file...}
    {my_file consulted, 0 msec 320 bytes}

    yes
| ?- father(X,Y).
    X = abraham,
    Y = isaac ?
| ?- halt.                          % quit
```

Tip: run through Emacs: C-x 2   M-x run-prolog