Lecture 11: Database Manipulation and Collecting Solutions

• Theory
  – Discuss database manipulation in Prolog
  – Discuss built-in predicates that collect all solutions to a problem into a single list

• Exercises
  – Exercises of LPN: 11.1, 11.2, 11.3
  – Practical session
Database Manipulation

• Prolog has five basic database manipulation commands:
  – assert/1
  – asserta/1
  – assertz/1
  – retract/1
  – retractall/1
Database Manipulation

- Prolog has five basic database manipulation commands:
  - assert/1
  - asserta/1
  - assertz/1
  - retract/1
  - retractall/1

Adding information
Removing information
Start with an empty database
Start with an empty database

?- listing.
yes
Using assert/1

?- assert(happy(mia)).
yes
Using assert/1

happy(mia).

?- assert(happy(mia)).
yes
?-
Using assert/1

happy(mia).

?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?-
Using assert/1

happy(mia).

?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?- assert(happy(vincent)),
   assert(happy(marsellus)),
   assert(happy(butch)),
   assert(happy(vincent)).
# Using `assert/1`

<table>
<thead>
<tr>
<th>happy(mia).</th>
<th>happy(vincent).</th>
</tr>
</thead>
<tbody>
<tr>
<td>happy(marsellus).</td>
<td>happy(butch).</td>
</tr>
<tr>
<td>happy(vincent).</td>
<td></td>
</tr>
</tbody>
</table>

```prolog
?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?- assert(happy(vincent)),
   assert(happy(marsellus)),
   assert(happy(butch)),
   assert(happy(vincent)).
yes
?- 
```
Changing meaning of predicates

• The database manipulations have changed the meaning of the predicate happy/1

• More generally:
  – database manipulation commands give us the ability to change the meaning of predicates during runtime
Dynamic and Static Predicates

• Predicates which meaning changing during runtime are called **dynamic** predicates
  – happy/1 is a dynamic predicate
  – Some Prolog interpreters require a declaration of dynamic predicates

• Ordinary predicates are sometimes referred to as **static** predicates
Asserting rules

happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

?- assert( (naive(X):- happy(X)).
Asserting rules

happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

naive(A):- happy(A).

?- assert( (naive(X):- happy(X)).
yes
?-
Removing information

• Now we know how to add information to the Prolog database
  – We do this with the `assert/1` predicate

• How do we remove information?
  – We do this with the `retract/1` predicate, this will remove one clause
  – We can remove several clauses simultaneously with the `retractall/1` predicate
Using retract/1

happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

naive(A):- happy(A).

?- retract(happy(marsellus)).
Using retract/1

happy(mia).
happy(vincent).
happy(butch).
happy(vincent).

naive(A):- happy(A).

?- retract(happy(marsellus)).
yes
?-
Using retract/1

happy(mia).
happy(vincent).
happy(butch).
happy(vincent).

naive(A):- happy(A).

?- retract(happy(marsellus)).

yes

?- retract(happy(vincent)).
Using retract/1

<table>
<thead>
<tr>
<th>Happy States</th>
<th>Retract Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>mia</td>
<td>retract(happy(marsellus)). yes</td>
</tr>
<tr>
<td>butch</td>
<td>retract(happy(vincent)). yes</td>
</tr>
<tr>
<td>vincent</td>
<td></td>
</tr>
</tbody>
</table>

naive(A):- happy(A).
Using retract/1

happy(mia).
happy(butch).
happy(vincent).

naive(A):- happy(A).

?- retract(happy(X)).
naive(A):- happy(A).

?- retract(happy(X)).
X=mia;
X=butch;
X=vincent;
no
?-
Using asserta/1 and assertz/1

• If we want more control over where the asserted material is placed we can use the variants of assert/1:
  – asserta/1 places asserted material at the beginning of the database
  – assertz/1 places asserted material at the end of the database
Memoisation

- Database manipulation is a useful technique
- It is especially useful for storing the results to computations, in case we need to recalculate the same query
- This is often called memoisation or caching
Example of memoisation

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
Example of memoisation

:- dynamic lookup/3.
addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.
addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

?- addAndSquare(3,7,X).
Example of memoisation

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).

?- addAndSquare(3,7,X).
X=100
yes
?-
Example of memoisation

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).

?- addAndSquare(3,7,X).
X=100
yes
?- addAndSquare(3,4,X).
Example of memoisation

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).

?- addAndSquare(3,7,X).
X=100
yes
?- addAndSquare(3,4,X).
X=49
yes
Using retractall/1

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).

?- retractall(lookup(_, _, _)).
Using retractall/1

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
  lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
  Res is (X+Y) * (X+Y),
  assert(lookup(X,Y,Res)).

?- retractall(lookup(_, _, _)).
yes
?-
Red cut

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
Red and Green Cuts

Red cut

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
  lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
  Res is (X+Y) * (X+Y),
  assert(lookup(X,Y,Res)).

Green cuts

:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
  lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
  \+ lookup(X,Y,Res), !,
  Res is (X+Y) * (X+Y),
  assert(lookup(X,Y,Res)).
A word of warning…

- A word of warning on database manipulation:
  - Often is a useful technique
  - But can lead to dirty, hard to understand code
  - It is non declarative, non logical
  - So should be used cautiously

- Prolog interpreters also differ in the way `assert/1` and `retract/1` are implemented with respect to backtracking
  - Either the assert or retract operation is cancelled over backtracking, or not
Consider this database

\[
\text{child}(\text{martha}, \text{charlotte}). \\
\text{child}(\text{charlotte}, \text{caroline}). \\
\text{child}(\text{caroline}, \text{laura}). \\
\text{child}(\text{laura}, \text{rose}). \\
\text{descend}(X, Y) : - \text{child}(X, Y). \\
\text{descend}(X, Y) : - \text{child}(X, Z), \\
\hspace{1cm} \text{descend}(Z, Y).
\]

?- \text{descend}(\text{martha}, X). \\
X = \text{charlotte}; \\
X = \text{caroline}; \\
X = \text{laura}; \\
X = \text{rose}; \\
\text{no}
Collecting solutions

• There may be many solutions to a Prolog query
• However, Prolog generates solutions one by one
• Sometimes we would like to have all the solutions to a query in one go
• Needless to say, it would be handy to have them in a neat, usable format
Collecting solutions

• Prolog has three built-in predicates that do this: `findall/3`, `bagof/3` and `setof/3`

• In essence, all these predicates collect all the solutions to a query and put them into a single list

• But there are important differences between them
The query

?- findall(O,G,L).

produces a list \( L \) of all the objects \( O \) that satisfy the goal \( G \)

- Always succeeds
- Unifies \( L \) with empty list if \( G \) cannot be satisfied
A findall/3 example

child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
              descend(Z,Y).

?- findall(X,descend(martha,X),L).
L=[charlotte,caroline,laura,rose]
yes
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
            descend(Z,Y).

?- findall(f:X,descend(martha,X),L).
L=[f:charlotte,f:caroline,f:laura,f:rose]
yes
Other findall/3 examples

\[
\text{child}(\text{martha}, \text{charlotte}). \\
\text{child}(\text{charlotte}, \text{caroline}). \\
\text{child}(\text{caroline}, \text{laura}). \\
\text{child}(\text{laura}, \text{rose}).
\]

\[
\text{descend}(\text{X}, \text{Y}) :\text{ if } \text{child}(\text{X}, \text{Y}) \\
\text{descend}(\text{X}, \text{Y}) :\text{ if } \text{child}(\text{X}, \text{Z}), \\
\text{descend}(\text{Z}, \text{Y}).
\]

?- \text{findall}(\text{X}, \text{descend}(\text{rose}, \text{X}), \text{L}). \\
\text{L} = [ ] \\
yes
Other findall/3 examples

child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y):= child(X, Y).
descend(X, Y):= child(X, Z),
               descend(Z, Y).

?- findall(d, descend(martha, X), L).
   L = [d, d, d, d]
   yes
findall/3 is sometimes rather crude

child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

\[
\text{descend}(X,Y) :\text{-} \text{child}(X,Y).
\]

?- findall(Chi,descend(Mot,Chi),L).
L=[charlotte,caroline,laura, rose, caroline,laura,rose,laura,rose,rose]
yes
The query

?- bagof(O,G,L).

produces a list \( L \) of all the objects \( O \) that satisfy the goal \( G \)

- Only succeeds if the goal \( G \) succeeds
- Binds free variables in \( G \)
Using bagof/3

child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).

?- bagof(Chi,descend(Mot,Chi),L).
Mot=caroline
L=[laura, rose];
Mot=charlotte
L=[caroline,laura,rose];
Mot=laura
L=[rose];
Mot=martha
L=[charlotte,caroline,laura,rose];
no
Using bagof/3 with ^

?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose, caroline, laura, rose, laura, rose, rose]
setof/3

- The query

```?- setof(O,G,L).```

produces a sorted list \( L \) of all the objects \( O \) that satisfy the goal \( G \)
- Only succeeds if the goal \( G \) succeeds
- Binds free variables in \( G \)
- Remove duplicates from \( L \)
- Sorts the answers in \( L \)
Using setof/3

child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).

?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose, caroline, laura, rose, laura, rose, rose]
yes

?-
Using setof/3

\[
\text{child}(\text{martha}, \text{charlotte}). \\
\text{child}(\text{charlotte}, \text{caroline}). \\
\text{child}(\text{caroline}, \text{laura}). \\
\text{child}(\text{laura}, \text{rose}). \\
\]

\[
\text{descend}(X, Y):= \\
\quad \text{child}(X, Y). \\
\text{descend}(X, Y):= \\
\quad \text{child}(X, Z), \\
\quad \text{descend}(Z, Y). \\
\]

?- \text{bagof}(\text{Chi}, \text{Mot}^\lor \text{descend}(\text{Mot}, \text{Chi}), \text{L}). \\
\text{L}=[\text{charlotte}, \text{caroline}, \text{laura}, \text{rose}, \\
\quad \text{caroline}, \text{laura}, \text{rose}, \text{laura}, \text{rose}, \text{rose}] \\
\text{yes} \\

?- \text{setof}(\text{Chi}, \text{Mot}^\lor \text{descend}(\text{Mot}, \text{Chi}), \text{L}). \\
\text{L}=[\text{caroline}, \text{charlotte}, \text{laura}, \text{rose}] \\
\text{yes} \\

?-
Next lecture

• Working with Files
  – Discuss how predicate definitions can be spread across different files
  – Modular Prolog components
  – Writing and reading from files