PROLOG

Database Manipulation

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 14

15/11/04

Contents

- Database Manipulation
  - assert/1, asserta/1, assertz/1
  - retract/1, retractall/1
  - dynamic/1, clause/2
  - `Caching` solutions.
  - `Listing‘ solutions to an output file.

Database Manipulation

- A Prolog program can be viewed as a database where the specification of relations between data is partly explicit (facts) and partly implicit (rules).
- Sometimes we want to be able to manipulate this database during execution (e.g. update facts, deduce new rules, etc).
- There exist built-in predicates that allow us to do this:
  - adding to the database: `assert/1, asserta/1, assertz/1`
  - deleting entries: `retract/1, retractall/1.`
- CAUTION: these facilities are easily misused and can lead to malfunctioning code as relationships that are necessary for the program to function are altered during execution.

  The database should only be manipulated if absolutely necessary and you should carefully control every change your code makes.

`assert`-ing New Entries

- New clauses can be added to the Prolog database with three built-in predicates.

`assert(Clause)`

  – succeeds by adding Clause to the current database.

- However, this doesn't specify *where* the entry is added in the database. The order of clauses for a predicate is usually very important as Prolog attempts to match clause heads in the order they were consulted.
- Therefore, we can specify where the clause is asserted:

`asserta(Clause)`

  – succeeds by adding Clause as the *first* clause for this predicate in the current database.

`assertz(Clause)`

  – succeeds by adding Clause as the *last* clause for this predicate in the current database.

`assert`-ing New Entries (2)

- Both facts and rules can be asserted:
  - Facts are written as they should appear in the database minus the final full-stop.

```
?- assert( son(tom, sue) ).
```

  - Rules are enclosed in brackets and *without the final full-stop*.

```
?- assert( ( mother(X,Y):- son(Y, X), female(X) ) ).
```

- Example database manipulation.

```
|?- listing.
yes
|?- assert(son(tom,sue)).
yes
|?- assert(female(sue)).
yes
|?- assert(( mother(X,Y):-
          son(Y,X),female(X))).
true ?
yes
```

```
| ?- listing.
female(sue).
mother(A,B):-
        son(B, A),
        female(A).
son(tom, sue).
yes
| ?- mother(sue,tom).
yes
```

dynamic/1

- Asserted entries are added to the *dynamic* copy of the consulted Prolog programs stored in Sicstus' memory buffer.
  - They are not written to the original program, i.e. the program doesn't change and once Sicstus is closed all changes to the database are lost.
- In order for a predicate already consulted in the Prolog program to be manipulated during runtime it needs to be declared as dynamic. We do this by adding a directive at the beginning of the code:

```
:- dynamic PredicateIndicator
```

  - where PredicateIndicator is the predicate followed by the arity e.g. **`append/3`**
- If the predicate is "new" then it is automatically dynamic and does not need to be declared.

clause/2

- Once a predicate is declared as dynamic you can check for its existence using **`clause(Head, Body)`**
- **`clause(Head, Body)`** succeeds if there is a clause in the current Prolog database which is unifiable with:
  - **`Head :- Body.`**

```
E.g.        :- dynamic a/2.
            a(1,2).
            a(3,4).
            a(X,Y):- b(X), b(Y).

|?- clause(a(Arg1,Arg2), Body).
Arg1 = 1, Arg2 = 2, Body = true?;
Arg1 = 3, Arg2 = 4, Body = true?;
Body = b(Arg1), b(Arg2)?;
no
```

- Note that if the clause is a fact, and has no body, then the second argument of `clause/2` is instantiated to *true*.

`retract`-ing entries

- We may also find that we need to remove clauses from the database.

`retract(Clause)`

- succeeds by removing from the Prolog database the *first* clause which matches Clause.

`retractall(Head)`

- succeeds by removing *all* clauses from the Prolog database *whose head match Head* (the body does not need to match).
- This is used to remove all definitions of a particular predicate.

```
|?- listing.
p(A):- a(A).
p(A):- a(A),b(A).
p(A):- a(A),b(A), c(A).
yes
|?- retract((p(Z):- a(Z))).
true ? yes
```

```
|?- listing.
p(A):- a(A),b(A).
p(A):- a(A),b(A), c(A).
yes
|?- retractall(p(Z)).
true ? yes
|?- listing.
yes
```

‘Caching’ solutions

- You will remember that there are no “global” variables in Prolog; all variables only retain a value within a clause (they are “local”).
- This can be very frustrating as it makes generating new data and then processing that data further very difficult. Data is lost across executions of Prolog commands.
- *A way to overcome this is to assert the new data to the database.*
- It can then be accessed as long as Sicstus remains open.
- If a solution to a query is asserted to the database in the same form as the query, next time the query is called an answer can be returned without the need for any computation.

```
|?- solve(problem1, Solution),
        asserta(solve(problem1, Solution)).
```

Writing the cache

- The problem with this is that the cache is deleted as soon as Sicstus is terminated.
- What we need is some way of permanently adding the new data to the Prolog file.
- We can use I/O commands to write the contents of the ‘cache; to a new Prolog file after we have asserted our new data to it.

1. Open the new output file, `tell('newfile.pl').`
2. List the contents of Prolog’s memory buffer using `listing/0`.
   - This is automatically written to the current output stream.
3. When the output stream is then closed, using `told/0`, the new file is written with all the new data included.

- *This is a very devious technique and should be used under great caution.* Listing the database turns everything into Prolog’s internal representation which is very hard to read and eliminates system calls (such as :- dynamic). There is also the chance that will you overwrite important data.

‘Caching’ solutions: example

- The program shown below can be used to find the position in the alphabet of a particular number. It is called with **`letter(Letter,Pos)`**, where either **`Letter`** or **`Pos`** can be variables:

```
|? listing.
alphabet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]
letter(f, 6).
letter(A, B):-
   alphabet(C),
   letter(A, C, B).

letter(A, [A|_], 1).
letter(A, [B|C], D):-
   A\==B,
   letter(A, C, E),
   D is E+1.
yes
```

- To find a solution the program must recurse through the alphabet up to 26 times and then backtrack the same number of times.

‘Caching’ example continued

```
| ?- letter(y,P).
1 Call: letter(y,_475) ?
2 Call: alphabet(_1038) ?
2 Exit: alphabet([a,b,c,d,e,f,g,h,i|...]) ?
2 Call: letter(y,[a,b,c,d,e,f,g,h,i|...],_475) ?
3 Call: y\==a ?
3 Exit: y\==a ?
3 Call: letter(y,[b,c,d,e,f,g,h,i,j|...],_2758) ?
4 Call: y\==b ?
4 Exit: y\==b ?
4 Call: letter(y,[c,d,e,f,g,h,i,j,k|...],_4528) ?
::::                                ← trace skipped to save space
4 Exit: letter(y,[c,d,e,f,g,h,i,j,k|...],23) ?
4 Call: _2758 is 23+1 ?
4 Exit: 24 is 23+1 ? ?
3 Exit: letter(y,[b,c,d,e,f,g,h,i,j|...],24) ?
3 Call: _475 is 24+1 ?
3 Exit: 25 is 24+1 ? ?
2 Exit: letter(y,[a,b,c,d,e,f,g,h,i|...],25) ? ?
1 Exit: letter(y,25) ?
P = 25 ?
```

‘Caching’ example continued

- If we then want to ask the question again we have to do exactly the same amount of processing.
- This is a waste as we already have the answer.
- If we *assert the answer as a new fact* in the database then next time it is asked it will unify directly without the need for further processing.

```
| ?- letter(y,P), asserta(letter(y,P)).
P = 25 ?
yes
```

Notice the new fact matches the format of the original query.

```
| ?- listing.
   alphabet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).

   letter(y, 25).
```

← new fact asserted as the first clause for this predicate

```
   letter(A, B) :-
        alphabet(C),  letter(A, C, B).

   letter(A, [A|_], 1).
   letter(A, [B|C], D) :-
        A\==B, letter(A, C, E), D is E+1.
yes
```

‘Caching’ example continued

- Now when the same query is asked the answer is immediate.

```
| ?- trace, letter(y,P).
% The debugger will first creep -- showing everything (trace)
1      1 Call: letter(y,_487) ? ?
1      1 Exit: letter(y,25) ?
P = 25 ?
yes
```

- Now the answer to every query has to be found only once. After that it is remembered as a fact and as long as the facts and rules from which it was deduced stay the same so will the answer.
- Now the ‘cached’ answers can be listed to a new file to store them for future use.

```
| ?- tell('cache1.pl').
yes
| ?- listing.
yes
| ?- told.
yes
```

‘Caching’ example conclusion

- The new file looks like this:

  ← Notice the ‘dynamic’ declaration is missing. This has to be re-added if the letter/2 is to be modified again

```
alphabet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
letter(y, 25).  <- new fact
letter(A, B) :-
     alphabet(C),
     letter(A, C, B).

letter(A, [A|_], 1).
letter(A, [B|C], D) :-
     A\==B,
     letter(A, C, E),
     D is E+1.
```

- A better solution might be to create a file that just contains solutions.
- These can be checked first whenever a query is asked and then if no solution is found it can be deduced and added to this file.
- The benefit of this is that your program is not modifying itself, which might lead to instability, it is modifying a separate, non-critical file.