

CSc 372

Comparative Programming Languages

28 : Prolog — The Database

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2013 Christian Collberg

Introduction

Manipulating the Database

- So far we have assumed that the Prolog database is **static**, i.e. that it is loaded once with the program and never changes thereafter.
- This is not necessarily true; we can add or remove facts and rules from the database at will.
- This is not necessarily good programming practice, but sometimes it is necessary and sometimes it makes for elegant programs.
- In a nutshell:
 - ① Allows us to program with side effects.
 - ② Justified under some circumstances.
 - ③ Often inefficient.

Operations on the Prolog Database

Assert

- `assert(X)` adds a clause to the database.
Not defined in `gprolog!`
- `asserta(X)` adds a clause to the *beginning* of the database.
- `assertz(X)` adds a clause to the *end* of the database.
- `assert` always succeeds, and backtracking does not undo the assertion.

- `assert` can be used in *machine learning* programs, program which learn new facts as they progress.
- In some Prolog implementations you have to specify whether a certain clause is **dynamic** (new clauses can be added to the database during execution) or **static**:

```
:- dynamic(hanoi/5).
```

This means that we can add and remove clauses with five arguments whose functor is **hanoi**.

Assert ... – Example

- Write a program that learns the addresses of places in a city.
- This program assumes a Manhattan-style city layout:
locations are given as the intersection of streets and avenues.

```
?- loc(whitehorse, Ave, St).
```

```
Ave = 8, St = 11
```

```
?- loc(airport, Ave, St).
```

```
-- this airport
```

```
what avenue? 5.
```

```
what street? 32.
```

```
Ave = 5, St = 32
```

```
?- loc(airport, Ave, St).
```

```
Ave = 5, St = 32
```

Assert ... – Example

```
location(whitehorse, 8, 11).
location(microsoft, 8, 42).
location(condomeria, 8, 43).
location(plunket, 7, 32).

% Do we know the location of X?
loc(X, Ave, Str) :- location(X, Ave, Str), !.

% if not, learn it!
loc(X, Ave, Street) :-
    nonvar(X), var(Ave), var(Street),
    write('-- this '), write(X), nl,
    write('what avenue? '), read(Ave),
    write('what street? '), read(Street),
    assert(location(X, Ave, Street)).
```


- `retract(X)` removes the first clause that matches `X`.
- `assert` and `retract` behave differently on backtracking. When we backtrack through `assert` nothing happens. When we backtrack to `retract` Prolog continues searching the database trying to find another matching clause. If one is found it is removed.
- If the argument to `retract(clause(X))` contains some uninstantiated variables they will be instantiated.
- `retract(X)` fails when no matching clause can be found.

- Backtracking does not undo the removal.

```
retractall(X) :-  
    retract(X), fail.  
retractall(X) :-  
    retract((X :- Y)),  
    fail.  
retractall(_).
```

- `clause(X, Y)` finds all clauses in the database with head `X` and body `Y`.

```
append([], X, X).  
append([A|B], C, [A|D]) :-  
    append(B, C, D).
```

```
?- clause(append(X, Y, Z), T).  
X=[], Y=_3, Z=_3, Y=true ;  
X=[_4|_5], Y=_6, Z=[_4|_7],  
    Y=append(_5, _6, _7) ;  
no
```

Clause. . .

- The goal `clause(X, Y)` instantiates `X` to the head of a goal (the left side of `:-`) and `Y` to the body.
- `X` can be just a variable (in which case it will match *all* the clauses in the database), a fully instantiated (*ground*) term, or a term which contains some uninstantiated variables.
- Note that a fact has a body `true`.

Clause...

List all the clauses whose head matches X.

```
list(X) :- clause(X, Y),  
    print(X, Y),  
    write('.') , nl, fail.  
list(_).
```

```
print(X, true) :- !, write(X).  
print(X, Y) :- write((X :- Y)).
```

```
?- list(append(X, Y, Z)).  
    append([], _4, _4).  
    append([_5|_6], _7, [_5|_8]) :-  
        append(_6, _8, _8).
```

Clausal Representation of Data Structures

- Normally we represent a data structure using a combination of Prolog lists and structures.
- A graph can for example be represented as a list of edges, where each edge is represented by a binary structure:
[edge(a,b), edge(c,b), edge(a,d), edge(c,d)]
- However, it is also possible to use *clauses* to represent data structures such as lists, trees, and graphs.
- It is usually not a good idea to do this, but sometimes it is useful, particularly when we are faced with a *static* data structure (one which does not change, or changes very little).

Clauses as Data Structures – Lists

```
list(c).
```

```
list(h).
```

```
list(r).
```

```
list(i).
```

```
list(s).
```

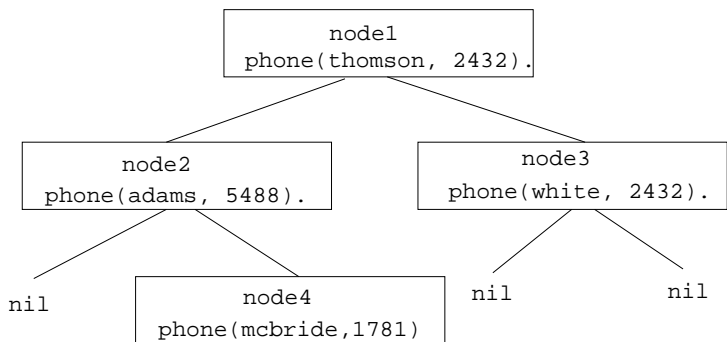
```
process_list :- list(X), process_item(X), fail.
```

```
process_list.
```

Clauses as Data Structures – Trees

```
t(node1, node2, phone(thompson, 2432), node3).  
t(node2, nil, phone(adams, 5488), node4).  
t(node3, nil, phone(white, 2432), nil).  
t(node4, nil, phone(mcbride, 1781), nil).
```


Clauses as Data Structures – Trees...



```
inorder(nil).  
inorder(Node) :-  
    t(Node, Left, P, Right),  
    inorder(Left),  
    write(P), nl,  
    inorder(Right).
```

```
?- inorder(node1).  
    phone(adams,5488)  
    phone(mcbride,1781)  
    phone(thompson,2432)  
    phone(white,2432)
```

Clausal Representation. . .

- In general it is a bad idea to represent data in this way.
- Inserting and removing data has to be done using `assert` and `retract`, which are fairly expensive operations.
- However, in Prolog implementations which support *clause indexing*, storing data in clauses gives us a way to access information *directly*, rather than through sequential search.
- The reason for this is that *indexing* uses hash tables to access clauses.

Switches

Switches

- From *Prolog by Example*, Coelho & Cotta.
- In some cases it is a good idea to use global data rather than passing it around as a parameter.
- Assume we want to be able to switch between short and long error messages. Instead of extending every clause by an extra parameter (clumsy and inefficient) we use a global switch.
- The first clause in `turnon` will fire if the switch is already turned on.
- The first clause in `turnoff` fails if `Switch` was already off.
- The first clause in `flip` fails if `Switch` was turned off, in which case the second clause fires and the switch is turned on.

Switches...

```
turnon(Switch) :-  
    call(Switch), !.  
turnon(Switch) :-  
    assert(Switch).  
  
turnoff(Switch) :-  
    retract(Switch).  
turnoff(_).  
  
flip(Switch) :-  
    retract(Switch), !.  
flip(Switch) :-  
    assert(Switch).
```

Switches...

```
turnon(terse_mess).
```

```
.....
```

```
flip(terse_mess).
```

```
message(C) :-
```

```
    terse_mes, write ('Error!'), nl, !.
```

```
message(C) :-
```

```
    write ('We are sorry to...'),
```

```
    write ('error has occurred near the symbol '),
```

```
    write(C), write(' Please accept our...'),
```

```
    nl, !.
```

Memoization

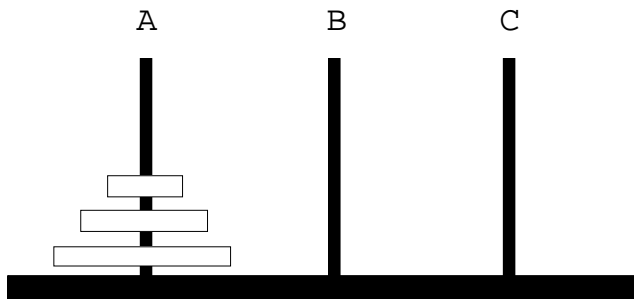
Memoization

- Many recursive programs are extremely inefficient because they solve the same subproblem several times.
- In **dynamic programming** the idea is simply to store the results of a computation in a table, and when we try to solve the same problem again we retrieve the value from the table rather than computing the value once more.
- There is a variation of dynamic programming known as **memoization**.

Memoization – Towers of Hanoi

- I'm sure you've heard of the Towers of Hanoi problem. It is one first year computer science students are tortured with to no end.
- The problem is to move a number of disks from a peg A to a peg B , using a peg C as intermediate storage. Additionally, we are only allowed to put smaller disks onto larger disks.
- A recursive solution of the problem to move N disks from A to B is as follows:
 - 1 Move $N - 1$ disks from A to C .
 - 2 Move the remaining (largest) disk from A to B .
 - 3 Move the $N - 1$ disks from C to B .

Memoization – Towers of Hanoi...



Memoization – Towers of Hanoi...

```
:- op(100, xfx, to).

hanoi(1, A, B, C, [A to B]).
hanoi(N, A, B, C, Ms) :-
    N > 1,
    N1 is N-1,
    hanoi(N1, A, C, B, M1),
    hanoi(N1, C, B, A, M2),
    append(M1, [A to B|M2], Ms).

go(N, Moves) :-
    hanoi(N, a, b, c, Moves).
```

Memoization – Towers of Hanoi...

?- go(2,M).

M = [a to c, a to b, c to b]

?- go(3,M).

M = [a to b, a to c, b to c,
a to b, c to a, c to b,
a to b]

?- go(4,M).

M = [a to c, a to b, c to b,
a to c, b to a, b to c,
a to c, a to b, c to b,
c to a, b to a, c to b,
a to c, a to b, c to b]

Memoization – Towers of Hanoi...

```
hanoi(1, A, B, C, [A to B]).  
hanoi(N, A, B, C, Ms) :-  
    N > 1, R is N-1,  
    lemma(hanoi(R, A, C, B, M1)),  
    hanoi(N1, C, B, A, M2),  
    append(M1, [A to B|M2], Ms).
```

```
lemma(P) :- call(P),  
            asserta((P :- !)).
```

```
go(N, Pegs, Moves) :-  
    hanoi(N, A, B, C, Moves),  
    Pegs=[A, B, C].
```

Memoization – Towers of Hanoi...

```
hanoi(1, _3, _5, _4, [_3 to _5]) :- !.  
hanoi(2, _3, _4, _5,  
      [_3 to _5, _3 to _4, _5 to _4]) :- !.  
hanoi(3, _3, _5, _4,  
      [_3 to _5, _3 to _4, _5 to _4,  
       _3 to _5, _4 to _3, _4 to _5,  
       _3 to _5]) :- !.
```

Example – Gensym

Example – Gensym

- From *Programming in Prolog*, Clocksin & Mellish.
- If we want to store data between different top-level queries, then using the database is our only option.
- In the following example we want to generate new atoms.
- In order to make this work, `gensym` has to store the number of atoms with a given prefix that it has generated so far. The clause `current_num(Root, Num)` is used for this purpose. There is one `current_num` clause for each kind of atom that we generate.

Example – Gensym. . .

```
gensym(Root, Atom) :-  
    get_num(Root, Num),  
    name(Root, Name1),  
    int_name(Num, Name2),  
    append(Name1, Name2, Name),  
    name(Atom, Name).  
  
get_num(Root, Num) :-  
    retract(current_num(Root, Num1)),  
    !, Num is Num1 + 1,  
    asserta(current_num(Root, Num)).  
get_num(Root, 1) :-  
    asserta(current_num(Root, 1)).
```

Example – Gensym...

```
int_name(Int, List) :- int_name(Int, [], List).
int_name(I, Sofar, [C|Sofar]) :-
    I<10, !, C is I+48.
int_name(I, Sofar, List) :-
    Tophalf is I/10, Bothalf is I mod 10,
    C is Bothalf + 48,
    int_name(Tophalf, [C|Sofar], List).
```

```
?- gensym(chris, A).
```

```
A = chris1
```

```
?- gensym(chris, A).
```

```
A = chris2
```

```
?- gensym(chris, A).
```

```
A = chris3
```

Readings and References

- Read Clocksin-Mellish, Chapter 6.