Knowledge Base

Prolog

Prolog Knowledge Base

- Knowledge base = database
  - set of facts/rules in the program
- Can add/subtract facts and rules at run time
- Adding facts/rules
  - assert, asserta, assertz
- Subtracting facts/rules
  - retract, retractall

Asserting a Fact

- Just tell prolog that it's so

```
?- raining.
ERROR: Undefined procedure: raining/0

?- assert(raining).
true.

?- raining.
true.
```

- Prolog didn't know about raining/0

Assertion Order

- **assert/1** puts the fact/rule in the database
  - doesn't specify where the entry is added in the database
  - (SWI-Prolog puts it at the end)

- The order of clauses for a predicate is very important as Prolog attempts to match clause heads in the order they were consulted
  - **asserta/1** puts fact/rule in front
  - **assertz/1** puts fact/rule at end

Assertion Order

```
?- assert(num(1)).
?- assertz(num(2)).
?- asserta(num(0)).
?- assertz(num(3)).
?- num(X).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3
```

Adds num(1) to KB
Adds num(2) to end of KB
Adds num(0) to front of KB
Adds num(3) to end of KB

Assertion Order: Exercise

```
?- asserta(what(1)).
?- assertz(what(2)).
?- asserta(what(3)).
?- assertz(what(4)).
?- asserta(what(5)).
     ?- what(X).
     X = ? ;
     X = ? ;
     X = ? ;
     ...
```

Asserting Rules

- Rules can also be asserted:
  - Rules are enclosed in brackets and *without the final full-stop*

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

Example database manipulation

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

?- mother(sue,tom).
true.
```

- To see the the rule/fact in knowledge base:

```
?- listing(mother/2).
```

Monotonic vs. Nonmonotonic logic

- Standard logic is monotonic: once something is true, it is true forever
- Logic isn't good to fit reality: reality may change!

- Prolog uses nonmonotonic logic
- Facts and rules can be changed at any time
  - such facts and rules are said to be **dynamic**

Marking clauses as “Dynamic”

- Standard Prolog allow to assert and retract clauses without any restrictions
- SWI-Prolog and some others require to mark variable clauses as **“dynamic”** to be manipulated during runtime

```
:- dynamic raining/0.
```

- The “:-” at the beginning is mandatory

Exercise

- Write a predicate that asks the user for a person's parents & asserts those facts

```
?- add_parents(mark).
Who is mark's father?  bob.
Who is mark's mother?  mary.
Yes
?- father(mark, Dad), mother(mark, Mom).
Dad = bob, Mom = mary
```

Solution

```
askParents(Person):-askforWho(Person,father,Dad),
                    askforWho(Person,mother,Mom),
                    assert(father(Person,Dad)),
                    assert(mother(Person,Mom)).

askforWho(Person,Role,Name):- write('Who is '),
                              write(Person),
                              write('\'s '),
                              write(Role),
                              write('?  '),
                              read(Name).
```

Retraction

- Tell Prolog to remove a fact/rule

```
?- raining.
Yes
?- retract(raining).
Yes
?- raining.
No
```

Retraction Order

- From first to last

```
?- retract(num(0)), retract(num(1)).
true;
true.
```

- retract fails if no clause matches

Retracting All Clauses

- rectractall/1 retracts multiple clauses
  - all clauses with *head* matching the argument

```
?- num(N).
N = 2;
N = 3
?- retractall(num(N)).
Yes
?- num(N).
false.
```

Retracting Rules

- As for asserting rules
  - use parentheses if body is compound
  - body may be a variable/partly instantiated

```
?- retract((mother(X,Y):-son(Y,X),female(X))).
true.
?- mother(sue,tom).
false.
```

Asserting and Retracting

- Used for AI programs that learn
  - create a new rule & add it to the database
  - forget an old rule

- Can also be used for efficiency
  - asserta solutions previously found
  - found before general code called

Naïve Fibonacci

```
fib(1, 1).
fib(2, 1).
fib(N, F):- N > 2,
            N1 is N - 1, fib(N1, F1),
            N2 is N - 2, fib(N2, F2),
            F is F1 + F2.
```

Trace fib(5,F)

```
fib(5, F0)
   fib(4, F1)
      fib(3, F2)
         fib(2, F3) →  F3 = 1
         fib(1, F4) →  F4 = 1
      fib(2, F5) →  F5 = 1
   fib(3, F2)
      fib(2, F3) →  F3 = 1
      fib(1, F4) →  F4 = 1
```

fib(3, F) gets calculated again

extra work done

much worse as #s get bigger

Assertional Fibonacci

```
fibon(1, 1).
fibon(2, 1).
fibon(N, F) :-
   N > 2,
   N1 is N - 1, fibon(N1, F1),
   N2 is N - 2, fibon(N2, F2),
   F is F1 + F2,
   asserta(fibon(N, F)).  % remember the result
                          % at the beginning
```

Trace fibon(5,F)

```
fibon(5, F0)
  fibon(4, F1)
   fibon(3, F2)
      fibon(2, F3) → F3 = 1
      fibon(1, F4) → F4 = 1
      asserta(fibon(3,2))
   fibon(2, F5) → F5 = 1
   asserta(fibon(4,3))
  fibon(3,F6) → F6 = 2
  asserta(fib2(5,5))
```

→ Saves work from calculating fib(3)

→ Matches asserted fact – no need to recalculate

Collecting all solutions

Collecting all solutions

- Generate all of the solutions to a given goal

```
?- member(X, [1,2,3,4]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
no
```

- It is useful to have **all the generated objects** available together—for example collected into a list
- The built-in predicates **bagof**, **setof**, and **findall** serve this purpose

Meta-predicates

- findall/3, setof/3, and bagof/3 are all ***meta-predicates***

findall(X,P,L)
setof(X,P,L)
bagof(X,P,L)

All produce a list L of all the objects X such that goal P is satisfied

- They all repeatedly call the goal P, instantiating the variable X within P and adding it to the list L
- They succeed when there are no more solutions

findall/3

- **findall/3** is the most straightforward of the three, and the most commonly used:

```
?- findall(X, member(X, [1,2,3,4]), Results).
Results = [1,2,3,4]
yes
```

- Solutions are listed in the result in the same order in which Prolog finds them
- If there are duplicated solutions, all are included. If there are infinitely-many solutions, it will never terminate!

findall/3

- The findall/3 can be used in more sophisticated ways
- The second argument, which is the goal, might be a compound goal:

```
?- findall(X,(member(X,[1,2,3,4]),X > 2), R).
R = [3,4]?
Yes
```

- The first argument can be a term of any complexity:

```
?- findall(X/Y,(member(X,[1,2,3,4]),Y is X*X),R).
R = [1/1, 2/4, 3/9, 4/16]?
yes
```

setof/3

- **setof/3** works very much like findall/3, except that:
  - It produces the ***set*** of all results, with any duplicates and the results ***sorted***
  - If any variables are used in the goal, which do not appear in the first argument, setof/3 will return a separate result for each possible instantiation of that variable:

```
age(peter,7).
age(ann,5).
age(pat,8).
age(tom,5).
age(ann,5).
```

Knowledge base

```
?-setof(Child,age(Child,Age),R).
Age = 5,
R = [ann,tom];
Age = 7,
R = [peter];
Age = 8,
R = [pat];
no
```

setof/3

- A ***nested*** call to setof/3 collects together the individual results:

```
?- setof(Age/Children,setof(Child,age(Child,Age),
   Children), AllRes).
   AllRes = [5/[ann,tom],7/[peter],8/[pat]].
```

- If the variable that appear in the first argument is not important:

```
?- setof(Child, Age^age(Child,Age), R).
R = [ann,pat,peter,tom].
```

  - This reads: 'Find the set of all children, such that the Child has an Age (whatever it might be), and put the results in R'

bagof/3

- **bagof/3** is very much like setof/3 except:
  - that the list of results might contain duplicates
  - and isn't sorted

```
?- bagof(Child, age(Child,Age),Results).
Age = 5, Results = [tom,ann,ann]
   Age = 7, Results = [peter]
   Age = 8, Results = [pat].
```

- bagof/3 is different to findall/3 as it will generate separate results for all the variables in the goal that do not appear in the first argument

```
?- findall(Child, age(Child,Age),Results).
Results = [peter,pat,tom,ann,ann].
```