AN ADVENTURE GAME

- Consider the following knowledge base:

```
location(egg,duck_pen).
location(ducks,duck_pen).
location(fox,woods).
location(you,house).

connect(yard,house).
connect(yard,woods).

is_closed(gate).
connect(duck_pen,yard) :- is_open(gate).
```

- We want to move around, be able to open and close the gate, pick the egg, and so on.
- In order to do this we need to modify the knowledge base dynamically.

MODIFYING THE KNOWLEDGE BASE

- Adding a fact to the knowledge base:
  - `assert` adds the fact given as argument somewere
  - `asserta` adds the fact given as argument at the beginning of the knowledge base
  - `assertz` adds the fact given as argument at the end
  - all variants succeed only once
- Removing a fact from the knowledge base:
  - `retract` removes one instance that unifies with the argument
  - removes one more instance at each redo attempt
  - fails when no removal is possible

CONVINCING THE KNOWLEDGE BASE TO ALLOW CHANGES

- All the industrial grade PROLOG implementations compile the knowledge base as to increase the speed of retrieving facts from it.
  - SWI PROLOG is one such an example
- In these variants, you need to specify which facts are changeable at run time.
  - These predicates will be stored separately, in an un-optimized fashion
  - The `dynamic` declaration must precede the predicate definition

```
:-  dynamic(you_have/1),
    dynamic(location/2),
    dynamic(is_closed/1),
    dynamic(is_open/1).
```

DON'T DO IT...

- ...unless strictly necessary
- self-modifying code is notoriously hard to get right and debug
- searching the knowledge base looses a great deal of efficiency
  - knowledge base search is a crucial process, even the slightest slow down has dramatic effects

Adventure game (cont'd)

- Moving around:

```
goto(X) :-
        location(you,L),
        (connect(L,X); connect(X,L)),
        retract(location(you,L)),
        assert(location(you,X)),
        write(' You are in the '),
        write(X), nl.
goto(X) :- write(' You cannot get there '), nl.
```

- Picking up the egg:

```
pick(egg) :-
            location(you,duck_pen),
            not you_have(egg),
            assert(you_have(egg)),
            write(' You picked the egg '), nl.
pick(X) :- write(' There is nothing to pick '), nl.
```

Adventure game (cont'd)

- Opening the gate:

```
open(gate) :-
            location(you,yard),
            is_closed(gate),
            retract(is_closed(gate)),
            assert(is_open(gate)),
            write(' Opened. '), nl.
open(X) :- write(' You cannot open that '), nl.
```

- How the other creatures react:

```
ducks :-
       is_opened(gate),
       retract(location(ducks,duck_pen)),
       assert(location(ducks,yard)).
ducks.

fox :-
     location(ducks,yard),
     location(you,house),
     write(' The fox has taken a duck '), nl.
fox.
```

ADVENTURE GAME (CONT'D)

- The main loop:

```
go :- done.
go :-
    write('>>'),
    read(X),
    call(X),
    go.

done :-
      location(you,house),
      you_have(egg),
      ducks, fox,
      write(' Thanks for getting the egg. '), nl.
```

Sample interaction

```
?- go.
>>goto(yard).
 You are in the yard
>>goto(duck_pen).
 You cannot get there from here
>>pick(egg).
 There is nothing to pick
>>open(gate).
 Opened.
>>goto(duck_pen).
 You are in the duck_pen
>>pick(egg).
 You picked the egg
>>goto(house).
 You cannot get there from here
>>goto(yard).
 You are in the yard
>>goto(house).
 You are in the house
 The fox has taken a duck
 Thanks for getting the egg.
yes
```

FOL: The weakest link

- Resolution or modus ponens are exact
  - there is no possibility of mistake if the rules are followed exactly.
- These methods of inference (also known as deductive methods) require that information be complete, precise, and consistent.
- By contrast, the real world requires common sense reasoning in the face of incomplete, inexact, and potentially inconsistent information.

INCOMPLETE FACTS

- A logic is monotonic if the truth of a sentence does not change when more facts are added. FOL is monotonic.
- A logic is non-monotonic if the truth of a proposition may change when new information (facts) is added or old information is deleted.

  "It rained last night if the grass is wet and the sprinkler was not on last evening. I am looking right now and see that the grass is wet."
  Did it rain last night?

`rained :- grass_is_wet,`
`           \+ sprinkler_was_on.`
`grass_is_wet.`
	`?- rained.`
	`Yes`
	`?- assert(sprinkler_was_on).`
	`Yes`
	`?- rained.`
	`No`
	`?- retract(sprinkler_was_on).`
	`Yes`
	`?- rained.`
	`Yes`

- Similar to the closed world assumption but more precise
- We specify particular predicates that are "as false as possible"
  - Meaning that they are false for all the objects except for those for which we know them to be true

    $$bird(X) \wedge \neg abnormal(X) \rightarrow flies(X)$$

    provided that $abnormal$ is circumscribed
  - We draw the conclusion that $flies(tweety)$ out of $bird(tweety)$ provided that we do not know that $abnormal(tweety)$ holds
- Implemented in Prolog by the `not` predicate (more or less)

- Default logic adds a new inference rule: if α is true and β is not known to be false then γ:

$$\frac{\alpha \quad : \quad \beta}{\gamma}$$

e.g.,

$$\frac{\texttt{grass\_is\_wet} \quad : \quad \neg\texttt{sprinkler\_was\_on}}{\texttt{rained}}$$

- Nonmonotonic logic adds a new operator $\mathbb{M}$:

$$\alpha \wedge \mathbb{M}\beta \rightarrow \gamma$$

stands for "if α is true and β is not known to be false then γ." e.g.,

$$\texttt{grass\_is\_wet} \wedge \mathbb{M}\neg\texttt{sprinkler\_was\_on} \rightarrow \texttt{rained}$$

$$\begin{array}{c} american(X) \wedge adult(X) \wedge \\ \mathbb{M}(\exists A: \ (car(A) \wedge owns(X, A))) \end{array} \rightarrow (\exists A: \ (car(A) \wedge owns(X, A)))$$

Nonmonotonic logic in Prolog

- Prolog implements nonmonotonic logic using `not/1` (as seen earlier)
  - The only difference is that the “not known to be false” part must have a negated formulation
  - Direct nonmonotonic statements can be formulated using `!/0`, though the formulation is a bit more verbose

    Typically, vehicles have four wheels. Trucks are vehicles. They have 18 wheels.

    ```
    vehicle(a).
    vehicle(b).
    vehicle(X) :- truck(X).
    truck(c).

    wheels(X,18) :- vehicle(X), truck(X),!.
    wheels(X,4) :- vehicle(X).
    ```

More dynamically generated stuff

- The name of a structure can never be a variable (even if that variable is actually bound):

```
?- write(p(b)).
p(b)
Yes

?- P = p, write(P(b)).
ERROR: Syntax error: Operator expected
```

MORE DYNAMICALLY GENERATED STUFF

- The name of a structure can never be a variable (even if that variable is actually bound):

```
?- write(p(b)).
p(b)
Yes

?- P = p, write(P(b)).
ERROR: Syntax error: Operator expected
```

- We can go around this limitation using the predicate `=../2` (pronounced "univ") that builds a structure for us out of a list of objects (the first object will be the name of the structure, the rest the arguments)

```
?- X =.. [a, b, c].        ?- a(b, c) =.. L.        ?- a(b, c) =.. [a|L].
X = a(b, c).               L = [a, b, c].           L = [b, c].

?- P = p, S =.. [P,b], write(S).
p(b)
P = p,
S = p(b).
```

Sidestepping Prolog's search strategy

- Find all the values that satisfy a predicate: `findall(Object,Goal,List)` produces a `List` of all `Object` that satisfy `Goal`
  - `Object` is usually just a variable, but can be any structure using the variables from `Goal`
  - `findall/3` succeeds exactly once (even if there is no way to satisfy the goal, case in which we get back the empty list)

```
?- findall(X,parent(X,peter),L).
L = [adam, eve].

?- findall(son(Y,X),parent(X,Y),L).
L = [son(peter, adam), son(peter, eve),
     son(paul, adam), son(paul, mary)].

?- findall(son(Y,X),parent(margaret,Y),L).
L = [].
```

```
parent(adam,peter).
parent(eve,peter).
parent(adam,paul).
parent(mary,paul).
```

SIDESTEPPING PROLOG'S SEARCH STRATEGY (CONT'D)

- `findall/3` is useful in sidestepping the default search strategy; interesting example: breath-first search

```
bsearch(Initial,Final,Result) :-
    bsearch([pair(Initial,[])],Final,[Initial],Result).

bsearch([pair(Final,Moves)|_],Final,_,Result) :- reverse(Moves,Result).

bsearch([pair(Final,_)|Rest],Final,Visited,Result) :-  % other solutions...
    bsearch(Rest,Final,Visited,Result),!. % no backtracking needed since
                                           % we perform the search by hand!

bsearch([pair(Current,Moves)|Rest],Final,Visited,Result) :-
    findall(pair(B,[M|Moves]),
            (move(Current,B,M),not(member(B,Visited))),
            Bag), % Fails on redo! This effectively eliminates
                  % Prolog's own search strategy on Current
                  % (however, Prolog's backtracking is used as a while
                  % loop in order to expand the states from Rest).
    append(Rest,Bag,L),
    bsearch(L,Final,[Current|Visited],Result).
```