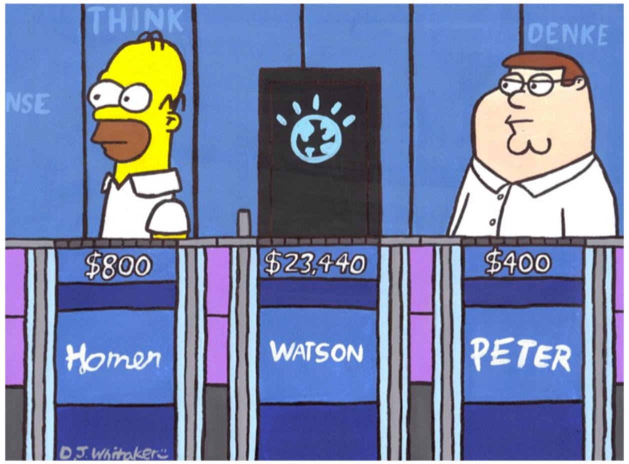
Prolog and declarative programming



Declarative programming

- Logic programming languages are fundamentally different from imperative programming languages.
  - State “this is true about the result”, not “this is how to compute the result”
- Encode a *knowledge base*
- Interact with the knowledge base through *queries*
  - The underlying inference engine will attempt to answer your queries

- It works by building constructive proofs that your queries are entailed by the knowledge base

Prolog

- *Programmation en logique* ("Programming in logic")
- Alain Colmeraeur & Philippe Roussel, 1971-1973
  - With help from theorem proving folks such as Robert Kowalski
  - Original project: Type in French statements & questions
- Computer needed NLP and deductive reasoning
- Efficiency by David Warren, 1977 (compiler, virtual machine)

- One of the foundations of early efforts in AI
  - Prolog favored by the European AI community
  - LISP favored by the American AI community

Applications of Prolog

- Intelligent database retrieval
- Natural language processing
- Expert systems
- Ontologies
- Theorem proving
- Robot planning
- Automated reasoning / problem solving
- Artificial intelligence

Prolog in one slide

- Everything in Prolog is built from *terms*.
- Three kinds of terms:
  - Constants: integers, real numbers, atoms
  - Variables
  - Compound terms
- A Prolog language system maintains a collection of facts and rules of inference
- A Prolog program is just a set of rules for this rule base
- The simplest kind of thing in the rule base is a *fact*: a term followed by a period

Helpful tutorials

- http://www.cs.toronto.edu/~sheila/324/f05/tuts/swi.pdf
- https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html#1

```
is_a( socrates, human ).

is_a( X, mortal ) :- is_a( X, human ).
```

Knowledge base	Queries

Knowledge base

```
is_a( socrates, human ).

is_a( X, mortal ) :- is_a( X, human ).
```

Queries

```
> is_a( socrates, mortal ).
true .

> is_a( socrates, human ).
true .

> is_a( Person, human ).
Person = socrates .

> is_a( Person, mortal ).
Person = socrates .
```

Knowledge base

```
factorial(0,1).

factorial(A,B) :-
          A > 0,
          C is A-1,
          factorial(C,D),
          B is A*D.
```

Queries

```
> factorial(10, What).
What=3628800
```

Knowledge base

```
borders( belgium, france ).
borders( france, spain ).
borders( germany, austria ).
borders( germany, denmark ).
borders( germany, switzerland ).
borders( netherlands, belgium ).
borders( netherlands, germany ).
borders( spain, portugal ).

route( A, B ) :-
  borders( A, B ).

route( A, B ) :-
  borders( A, Z ),
  route( Z, B ).
```

Queries

```
> route(netherlands,portugal).
true .
```

Knowledge base

```
...

route( A, B, [ go(A,B) ] ) :-
  borders( A, B ).

route( A, B, [ go(A,Z) | ZtoB ] ) :-
  borders( A, Z ),
  route( Z, B, ZtoB ).
```

Queries

```
> route(netherlands,portugal,R).
R = [go(netherlands, belgium),
     go(belgium, france),
     go(france, spain),
     go(spain, portugal)]
```

Knowledge base

```
tidy( [], [] ).

tidy( [ Toy | OtherToys ],
      [ pick_up(Toy),
        move_to(toybox),
        drop(Toy) | OtherCommands ] ) :-
  tidy( OtherToys, OtherCommands ).
```

"This is how to tidy up your room:
if there are no toys lying around,
do nothing.
if there is a toy lying around,
pick it up and put it in the toybox,
and tidy up your room."   - Patrick Winston

Queries

```
> tidy( [ teddy, ball, golly, bat ], Cmds ).

Cmds = [ pick_up(teddy),
         move_to(toybox),
         drop(teddy),
         pick_up(ball),
         move_to(toybox),
         drop(ball),
         pick_up(golly),
         move_to(toybox),
         drop(golly),
         pick_up(bat),
         move_to(toybox),
         drop(bat)]
```

Knowledge base

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X),
parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Queries

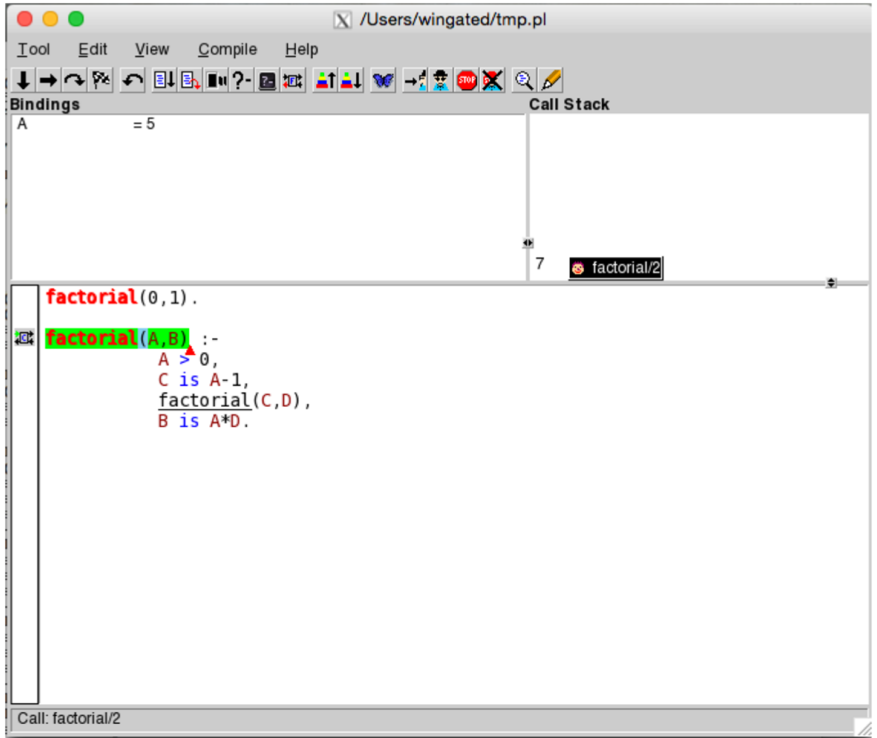
```
> sibling(sally, erica).
true

> parent_child(Parent, erica),
  parent_child(Parent, sally).
Parent = tom

> parent_child(Parent, Child).
Parent = tom,
Child = sally
Parent = tom,
Child = erica
Parent = mike,
Child = tom
Parent = trude,
Child = sally.
```

Debugging

- Can use trace. or notrace. to trace execution. (see tutorials)
- Also check out spy (like a breakpoint)




```
1 ?- ['/Users/wingated/tmp.pl'].
true.

2 ?- trace.
true.

[trace] 2 ?- factorial(5,What).
   Call: (7) factorial(5, _G929) ? creep
   Call: (8) 5>0 ? creep
   Exit: (8) 5>0 ? creep
   Call: (8) _G1007 is 5+ -1 ? creep
   Exit: (8) 4 is 5+ -1 ? creep
   Call: (8) factorial(4, _G1008) ? creep
   Call: (9) 4>0 ? creep
   Exit: (9) 4>0 ? creep
   Call: (9) _G1010 is 4+ -1 ? creep
   Exit: (9) 3 is 4+ -1 ? creep
   Call: (9) factorial(3, _G1011) ? creep
   Call: (10) 3>0 ? creep
   Exit: (10) 3>0 ? creep
   Call: (10) _G1013 is 3+ -1 ? creep
   Exit: (10) 2 is 3+ -1 ? creep
   Call: (10) factorial(2, _G1014) ? creep
   Call: (11) 2>0 ? creep
   Exit: (11) 2>0 ? creep
   Call: (11) _G1016 is 2+ -1 ? creep
   Exit: (11) 1 is 2+ -1 ? creep
   Call: (11) factorial(1, _G1017) ? creep
   Call: (12) 1>0 ? creep
   Exit: (12) 1>0 ? creep
   Call: (12) _G1019 is 1+ -1 ? creep
   Exit: (12) 0 is 1+ -1 ? creep
   Call: (12) factorial(0, _G1020) ? creep
   Exit: (12) factorial(0, 1) ? creep
   Call: (12) _G1022 is 1*1 ? creep
   Exit: (12) 1 is 1*1 ? creep
   Exit: (11) factorial(1, 1) ? creep
   Call: (11) _G1025 is 2*1 ? creep
   Exit: (11) 2 is 2*1 ? creep
   Exit: (10) factorial(2, 2) ? creep
   Call: (10) _G1028 is 3*2 ? creep
   Exit: (10) 6 is 3*2 ? creep
   Exit: (9) factorial(3, 6) ? creep
   Call: (9) _G1031 is 4*6 ? creep
   Exit: (9) 24 is 4*6 ? creep
   Exit: (8) factorial(4, 24) ? creep
   Call: (8) _G929 is 5*24 ? creep
   Exit: (8) 120 is 5*24 ? creep
   Exit: (7) factorial(5, 120) ? creep
What = 120 .

[trace] 3 ?- |
```