

PROLOG

- install either
 - GNU Prolog (gprolog): <http://www.gprolog.org/>
 - SWI-Prolog (swipl): <https://www.swi-prolog.org/>
 - these are generally command-line programs, but there are graphical IDEs
- tutorial
 - <https://people.engr.tamu.edu/ioerger/prolog.txt>

Prolog Syntax

- Definite clauses (fact and conjunctive rules)
- facts: predicates with args, *followed by a period*.
 - `color(apple,red). meat(hamburger). in(london,england). college_of(csceDept,engineering).`
 - predicate names and constants must start with lower case
- rules:
 - write them backwards, using `:-` for $\leftarrow$ (read it as “if”)
 - use commas for ‘and’
 - drop $\forall$; variable must start with upper case
 - $\forall x,m,st \text{ graduated}(x,m) \wedge \text{medicalSchool}(m) \wedge \text{passedBoards}(x,st) \rightarrow \text{doctor}(x)$
 - `doctor(X) :- graduated(X,M),medicalSchool(M),passedBoards(X,State) .`
 - "X is a doctor IF X graduated from a medical school and passed board exams in some state"

Using Prolog

- run it from command-line, get interactive prompt
 > swipl
 Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
 1 ?-
- load .pl files
 1 ?- ['examples.pl']. // shorthand for consult('examples.pl').
- type in queries (see next slide)
- quitting
 2 ?- halt.
- if you trigger an error warning, type 'a' to abort back to prompt
 3 ?- foo(_).
 ERROR: Undefined procedure: foo/1
 Exception: (8) foo(_8282) ? **a**
 % Execution Aborted
 4 ?-

Using Prolog

- make queries
 - solutions are variable bindings, not just T/F – this is how Prolog *computes*
 - get additional solutions by typing ‘;’
 - 4 ?- color(X). // equiv. to: " $\exists x$ color(x) ?"
 - X = red ;
 - X = green ;
 - X = blue.
 - you can also make queries with multiple goals, with commas:
 - lawyer(X),licensedIn(X,alabama).
 - X = atticusFinch ...
 - teachesAt(Faculty,tamu),degree(Faculty,phd),field(Faculty,math).
 - Faculty=stephen_fulling ;
 - Faculty=boris_hanin...

Prolog does Back-chaining (with unification)

```
animal(X) :- mammal(X).
animal(X) :- bird(X).
animal(X) :- fish(X).
mammal(X) :- dog(X).
mammal(X) :- cat(X).
bird(X) :- canary(X).
bird(X) :- penguin(X).
bird(X) :- owl(X).
```

```
dog(fido).
dog(snoopy).
cat(garfield).
canary(tweety).
canary(woodstock).
penguin(opus).
owl(hedwig).
person(john).
state(rhode_island).
```

```
4 ?- animal(X).
X = fido ;
X = snoopy ;
X = garfield ;
X = tweety ;
X = woodstock ;
X = opus ;
X = hedwig ;
```

goal-stack:

```
animal(X)
  mammal(X) // try first rule, choice-point
    dog(X).
      X=fido (solution 1)
      X=snoopy
      no more solutions, so back-track
    cat(X).
      X=garfield
  bird(X)
    canary(X).
      X=tweety
    penguin(X).
```

Note - you can ask Prolog to display tracing info during a query by typing 'trace.'
Then type the query. To get out of it, type 'nodebug.'

Prolog files (.pl)

- rules can span across multiple lines
- order matters! (for back-chaining)
- group your facts or rules of same predicate name together
 - otherwise, it might give you a warning, which is harmless
- comments are indicated by '%'
- if ';' isn't working right, try this:
 `set_prolog_flag(tty_control,false).`

Colonel West example in Prolog

colonel west.pl:

% from AIMA

criminal(X) :- american(X), weapon(Y), sells(west,Y,Z), hostile(Z).

weapon(Y) :- missile(Y).

hostile(Z) :- enemy(Z,america).

sells(west,m1,nono).

missile(m1).

enemy(nono,america).

query:

?- criminal(A).

A = west.

- there is a lot of other stuff in Prolog
 - numerics: there are predicates for doing math (+, *, log...), and operators for comparison (<, =, etc)
 - negation: (we will talk about this later)
 - lists: special notation for using lists as terms, ([Head|Rest])
 - 'cut' (!): operator for controlling execution flow
 - '_': anonymous variables
 - format(): for printing out strings
 - this always evaluates to True as an antecedent, but prints out as side-effect of execution.
- ```
message(M,Name) :- format("~w from ~w", [M,Name]) .
?- message("hello", "joe") .
hello from joe
```

# Doing Math in Prolog

- suppose you want to write a function for 'doubling' numbers

- write a predicate with 2 args, to be used as 'input' and 'output'
- in the body, use 'is' to bind a variable to a computed value
- this will get unified and returned when the predicate succeeds

```
double(X,Y) :- Y is 2*X.
```

```
?- double(5,A) .
```

```
A = 10
```

- other functions are usually available, like sin, exp, sqrt

```
tan(Theta,Z) :- C is cos(Theta), S is sin(Theta), Z is S/C. % in radians
```

- can you write a conversion function: **radians(Deg,Rad) :- ...?**

- comparison operators act like regular antecedents, i.e. tests that are T or F.

- see [http://www.gprolog.org/manual/html\\_node/gprolog030.html](http://www.gprolog.org/manual/html_node/gprolog030.html)

```
large_frog(X) :- frog(X), length(X,W), W > 10. % large frogs are over 10 cm long
```

```
odd(A) :- B is A mod 2, B==1.
```

```
even(A) :- B is A mod 2, B\==1. % '\==' is inequality operator in gprolog
```

- can define mathematical functions in prolog
- typically defined as relations with args for input AND output

```
factorial(1,1). % base case
factorial(N,F) :- % rule
 N>1,
 N1 is N-1,
 factorial(N1,F1),
 F is N * F1.
```

```
?- factorial(10,N).
N = 3628800.
```

execution trace:

```
factorial(10,N) calls
factorial(9,N) calls
factorial(8,N) calls
...
factorial(2,N) calls
factorial(1,N) which returns
factorial(1,1).
factorial(2,2).
factorial(3,6).
factorial(4,24)...
factorial(10,3628800).
```

You can use this idea to calculate square roots by Newton-Raphson iteration. Write Prolog rules to define `sqrt(A,B)`.

# Paradigms for Programming in Prolog (Use Cases)

- 1. Expressing FOL sentences that define concepts
  - examples
    - `criminal(X) :- ... weapon(W) :- ... hostile(C) :- ...`
    - `check(Board,Player) :-` % in the sense of chess
    - `loan_at_risk_of_default(L) :-`
    - `invasive_surgery(P) :-`
    - `can_graduate(P) :-`
    - `grandmother(A,B) :- mother(A,C),mother(C,B).` % if there exists a C in between
    - `safe(Row,Col) :-` % from wumpus world
    - ...

```
criminal(X) :- american(X),weapon(Y),hostile(Z),sells(X,Y,Z).
sells(west,C,nono) :- owns(nono,C),missile(C).
weapon(D) :- missile(D).
hostile(E) :- enemy(E,america).
```

# Paradigms for Programming in Prolog (Use Cases)

- 2. Datalog

- predicates encode facts like tuples in a database
- rules query them like 'joins'
- rules can also define higher concepts, and search for combinations of facts that satisfy them
- example: define 'outpatient\_procedure(X)' based on body parts or equipment used, and then search database for all outpatient procedures performed

```
state(al).
state(ak).
state(ca).
state(co).
...
ocean(atlantic).
ocean(pacific).
island(Hi).
```

```
adjacent(ca,pacific).
adjacent(fl,atlantic).
adjacent(ny,atlantic).
adjacent(tx,atlantic).
adjacent(hi,pacific).
```

```
...
```

```
eastCoast(S) :- state(S),adjacent(S,atlantic).
westCoast(S) :- state(S),adjacent(S,pacific).
coastal(S) :- state(S),ocean(O),adjacent(S,O).
```

# Paradigms for Programming in Prolog (Use Cases)

- 3. Calculating mathematical functions
  - include multiple args for 'input' values (bound when called) and 'output' (bound when return)
  - `double(5,A).` => `A=10`
  - `factorial(5,F).` => `F=120`

# Paradigms for Programming in Prolog (Use Cases)

- 4. Enumerating Combinations of things

- generate all 3-bit strings (assigning values 0/1 to vars A-C)

```
bits3(A,B,C) :- bit(A),bit(B),bit(C).
```

```
bit(0).
```

```
bit(1).
```

- think about how back-tracking works by trying A=0, B=0, C=0 first (since bit(A) unifies with bit(0) hence A is bound to 0...),
- then changes C from 0 to 1 for second solution, then backtracks and flips B to 1 and sets C to 0 again...

```
?- bits3(A,B,C).
```

```
A = 0, B = 0, C = 0 ;
```

```
A = 0, B = 0, C = 1 ;
```

```
A = 0, B = 1, C = 0 ;
```

```
A = 0, B = 1, C = 1 ;
```

```
A = 1, B = 0, C = 0 ;
```

```
A = 1, B = 0, C = 1 ;
```

```
A = 1, B = 1, C = 0 ;
```

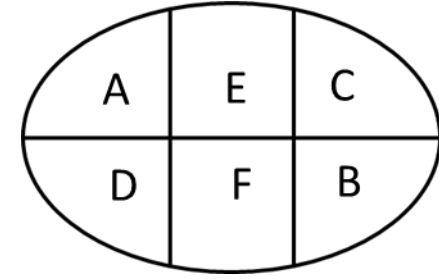
```
A = 1, B = 1, C = 1.
```

type semi-colon to  
get all 8 solutions

# Paradigms for Programming in Prolog (Use Cases)

- 5. solving Constraint Satisfaction Problems
  - generate possible solution combinatorially; then check to see if they satisfy constraints (generate-and-test paradigm)
  - example: map-coloring (see next slide)
  - try implementing cryptarithmic problems like SEND+MORE=MONEY
    - hint: generate all combinations of digit assignments, then check for correctness
  - try solving the 5-queens problem
    - hint: generate all possible locations for 5 queens, and eliminate any that have attacks

# Using Prolog to solve the map-color CSP



```
color(red) . color(green) . color(blue) .
```

```
mapcolor(A,B,C,D,E,F) :-
```

```
 color(A) , color(B) , color(C) , color(D) , color(E) , color(F) ,
```

```
 A \== D, A \== E, D \== F, E \== F, E \== C, F \== B, B \== C.
```

% generate all  
possible colorings

% apply adjacency  
constraints

```
?- map_color(A,B,C,D,E,F) .
```

```
A = red, B = red, C = green, F = green, D = blue, E = blue ;
```

```
A = red, B = red, C = blue, F = blue, D = gggreen, E = green ;
```

```
...
```