Intro to Logic Programming and the Prolog Language

Bob Keller

November 2008

What is this?

- A type of computation model and declarative programming language.
- Useful for:
  - Language understanding and translation
  - Databases
  - Knowledge representation
  - Artificial intelligence applications

Why study this?

- Expands expressiveness over what we have seen so far.
- A good language for learning about ideas of:
  - certain forms of logic
  - non-determinism
  - backtracking

Who uses Prolog?

- People who want to have a broad set of intellectual and problem-solving tools at their disposal.

Advantages of Succinctness

- Code moves closer to concepts (and farther from machine details).
- Easier to ascertain correctness.
- Easier to modify, for purposes of software evolution.

Prolog's Origins

(see http://en.wikipedia.org/wiki/Prolog)

- Prolog was invented at the University of Montreal around 1970 by Alain Colmerauer, who since has been professor at the École Supérieure d'Ingénieurs de Luminy in Marseille, France.
- The original work was a grammar-based language for natural language translation.

Varieties of Logic

- **Proposition** Logic:
  - Propositions are symbols which may be assigned one of two values:
    - true
    - false
  - without regard to specific individuals.
- **Predicate** Logic:
  - Predicates can be viewed functions from a domain of individuals to {true, false}.

Comparison

- Propositions:
  - hmc_is_great
  - caltech_is_in_glendale
- Predicates (X and Y are variables)
  - is_great(X) is_great(hmc)
  - is_in(X, Y) is_in(caltech, glendale)

Logic of Implication (Prolog Style)

- **Logical rule:**

  Q :- P1, P2, P3.

  means that proposition or predicate application Q **is implied by** the **conjunction** of P1, P2, and P3.

  If each of P1, P2, P3 are true, then Q is true.

  (However, Q could also be true even if one of P1, P2, P3 isn't.)

Prolog Lingo

- This is a **clause**:

  Q :- P1, P2, P3.

  - Q is called the **head**.
  - P1, P2, P3 is called the **body**.

  - Each of Q, P1, P2, P3 are individually called **goals**.

Example Clause

- prepared_for_exam :-

    read_book,

    worked_problems,

    attended_lectures.

Example Clause

- prepared_for_exam :-
  knows_it_all.

Example Clause

- prepared_for_exam :-
  tutored_by_someone_prepared.

Facts

- Facts are clauses with an empty body.
- They assert the truth of something without qualification.

Examples of Possible Facts

read_book.
worked_problems.
attended_lectures.
knows_it_all.
tutored_by_someone_prepared.

Depending on the facts present, a **goal**
prepared_for_exam
maybe inferred as true or not.

Success and Failure

- A goal is presented interactively to Prolog as:

```
?- prepared_for_exam.
```

Depending on the facts, this goal may **succeed** or **fail**.

Success and Failure

- A goal **succeeds** provided one of:
  - There is a **fact** that matches the goal.
  - There is a **clause**,
    - the head of which matches the goal, and
    - all goals in the body succeed.
- [Notice the first blue bullet is really a special case of the second.]
- If a goal doesn't succeed, then it **fails**.

Success and Failure Examples

- If the facts are:

```
read_book.
worked_problems.
attended_lectures.
```

  and there is a clause:

```
prepared_for_exam :-
  read_book,
  worked_problems,
  attended_lectures.
```

  then the goal

```
?- prepared_for_exam.
```

  succeeds. If any of the facts is missing, the goal fails.

Success and Failure Examples

- If the facts are:

```
read_book.
attended_lectures.
tutored_by_someone_prepared.
```

  and there is a clause:

```
prepared_for_exam :-
    tutored_by_someone_prepared.
```

  then the goal

  ?- prepared_for_exam.

  succeeds.

How Prolog Works

- In general, there can be several goals, **all of which** need to succeed.
- Think of these goals as being kept in a **stack**.
- Success occurs when the stack is empty.
- The first goal is removed from the stack.
- Prolog searches for a clause having a matching head.
  - If none is found, then there is overall failure.
  - If a matching fact is found, then execution continues with the rest of the stack.
  - If a matching clause is found, then the clauses in the body of the clause are pushed onto the stack (with the leftmost goal now at the top) and execution continues with the new list.

Backtracking

- When a failure occurs, Prolog does not stop. It goes back to the previous clause it used, and restores the list to the way it was just before. (It “backtracks”.)
- It then tries any alternative clauses that follow that clause.
- Clauses are tried in the order listed in the program, until there are no more clauses left.
- For new each goal, searching starts afresh.

Backtracking Example

- Suppose the clauses and facts are:
  - prepared_for_exam :-
    read_book,
    worked_problems,
    attended_lectures.
  - prepared_for_exam :-
    tutored_by_someone_prepared.
  - read_book.
  - worked_problems.
  - tutored_by_someone_prepared.
- Consider the goal
  ?- prepared_for_exam.

Backtracking Example

- The initial goal stack is:
  - [prepared_for_exam]
- After a match with the first clause, the stack becomes:
  - [read_book, worked_problems, attended_lectures]
- then, as facts are matched, the stack becomes:
  - [attended_lectures]
- but here there is no matching fact or clause. Backtracking occurs, restoring the list to:
  - [prepared_for_exam]
- The next clause that matches changes the list to:
  - [tutored_by_someone_prepared]
- The first goal matches a fact, leaving:
  - []
- The original goal thus succeeds.

The **Two Compatible Interpretations** of Prolog Execution

- Logical interpretation:
  - Implications and facts.
- Procedural interpretation:
  - Goals, backtracking, etc.

Prolog's Negation

- Facts are always asserted in the positive sense.
- Negation can be tested, but not asserted.
- Negation is "negation as failure":

  \+*Goal*

  succeeds iff *Goal* fails.
- In classical logic, |- G means "G is provable".

  \+ is a representation of "is not provable".

Negation Example

- The clause

```
prepared_for_exam :-
    read_book,
    worked_problems,
    attended_lectures,
    \+ slept_during_lectures.
```

will enable

```
prepared_for_exam
```

to succeed only if

```
slept_during_lectures
```

does *not* succeed.

Predicate vs. Propositional Goals

- The goals so far have been propositional:
  - Each is either **invariably** true (succeeds) or false (fails).
- Using predicates, success or failure depends on **arguments**.
- Each fact and clause can have one or more arguments.

Exam Passing for the Full Class

- read_book(fred).
- read_book(judy).
- worked_problems(judy).
- attended_lectures(fred).
- attended_lectures(judy).
- tutored_by(fred, bob).
- tutored_by(sam, judy).

Predicate Form of Exam Passing

```
prepared_for_exam(X) :-
  read_book(X),
  worked_problems(X),
  attended_lectures(X).

prepared_for_exam(X) :-
  tutored_by(X, Y),
  prepared_for_exam(Y).
```

Extreme Case-Sensitivity

- In Prolog,
  - Variables always start with upper-case or underscore.
  - Things that start with lower-case are:
    - Predicates, propositions
    - Data items (**atoms**):
      Similar to symbols in Scheme
  - Data items can also start with upper-case **if singly-quoted**, e. g. 'John Hancock'.
  - Unlike Scheme, '-' is not considered to be just another letter.

Case Sensitivity, Arity

- read_book(fred).
- prepared_for_exam(X) :-
    tutored_by(X, Y),
    prepared_for_exam(Y).

Variables:
    X, Y

Atoms:
    fred

Predicates:
    read_book/1, prepared_for_exam/1, tutored_by/2.

/N indicates the **arity** (number of arguments) of the predicate.
    Predicate names can be overloaded.

Matching = Unification

- **Unify** means: “make the same”.
  - Two **atoms** are unifiable iff identical.
  - A **variable** can be unified with anything (even another variable), by substituting the latter thing for the variable.
  - Two predicate expressions can be unified, provided that:
    - The predicate names are identical.
    - The number of arguments is the same in both.
    - Each of the arguments can be pairwise-unified, by a **common substitution**.

Prolog Unification Examples

Term 1	Term 2	Unifiable?	Substitution
fred	bob	No	
fred	X	Yes	X ← fred
X	bob	Yes	X ← bob
X	Y	Yes	X ← Y
p(X, Y)	p(fred, bob)	Yes	X ← fred Y ← bob
p(X, bob)	p(fred, Y)	Yes	X ← fred Y ← bob
p(X, bob)	p(fred, X)	No	

More Unification Examples

Term 1	Term 2	Unifiable?	Substitution
p(X, X)	p(fred, bob)	No	
p(X, f(Y))	p(Y, Z)	Yes	$X \leftarrow Y$ $Z \leftarrow f(Y)$ - OR - $Y \leftarrow X$ $Z \leftarrow f(X)$
p(a, f(Y))	p(Y, Z)	Yes	$Y \leftarrow a$ $Z \leftarrow f(a)$
p(g(Z), f(Y))	p(Y, Z)	Yes (but only in Prolog)	$Y \leftarrow g(f(g(f(..$ $Z \leftarrow f(g(f(g..$

Quiz 1: Complete the Table

Term 1	Term 2	Unifiable?	Substitution
p(X, fred)	p(fred, X)		
p(X, Y)	p(Y, Z)		
p(b, f(b))	p(f(X), b)		
p(a, X)	p(a, f(a))		
p(X, f(Z), Z)	p(a, X, a)		
p(X, Y, Z)	p(f(Y), g(Z), a)		

Use = in Command to Check Unifiability

```
?- p(a, f(Y)) = p(Y, Z).
Y = a,
Z = f(a)
Yes

?- p(X, X) = p(fred, bob).
No

?-  p(g(Z), f(Y)) = p(Y, Z).
Z = f(g(**)),
Y = g(f(**))
Yes
```

Unification and Clause Searching

- The variables of a clause are purely **local** to the clause. Variables do **not** connect one clause to another.
- When **searching for a match**, the variables in a clause are first **renamed** uniformly across the clause so that they are distinct from any variables that might be in the goal.
- Any **substitution applied to a goal** is applied to all remaining goals in the list.

Example

- **Clauses:**
  - prepared_for_exam(bob).
  - tutored_by(fred, bob).
  - prepared_for_exam(X) :-
    tutored_by(X, Y),
    prepared_for_exam(Y).
- **Goals:**
  - [prepared_for_exam(fred)]
- **Substitution:**
  - X1 ← fred
- **New goals:**
  [tutored_by(fred, Y1), prepared_for_exam(Y1)]
- **Substitution:**
  - Y1 ← bob
- **New goals:**
  [prepared_for_exam(bob)]

Recursive Example

```
child(Parent, Child, Graph) :-
    member([Parent, Child], Graph).

isDescendant(Ancestor, Desc, Graph) :-
    child(Ancestor, Desc, Graph).

isDescendant(Ancestor, Desc, Graph) :-
    child(Ancestor, Child, Graph),
    isDescendant(Child, Desc, Graph).
```

```
?- isDescendant(e, a, [[a,b], [b, c], [c,d], [b,e], [a,f]]).

Yes
```

Prolog's Data Types

- Atoms: x, abc, y99, this_is_too
- Numbers: 789, 15.3e-27
- Terms: f(x, 789)
- Lists (special type of term):
  - [red, green, blue]
  - [[red, 10], [green, 20], [blue, 50]]

Throw-Away Variable

- Variables beginning with _ (including _ itself) are “throw away“ or “don’t care”.
- They match anything.
- They do not need to unify with other instances of the same variable.

Database Applications

- Data are stored as predicate facts (aka “relations”).
- Queries are goals.
- Substitutions (resulting from unifications) are results

Relational Database Example

lives

name	dorm
John	East
Naima	South
Alice	West
Toshiko	East
Roy	North
Albert	South

takes

name	dept	number
John	CS	60
Naima	CS	60
Alice	CS	5
Toshiko	CS	5
Albert	CS	60
Roy	Math	55
Naima	Math	55
Alice	Math	70
Toshiko	Math	80
Albert	Math	55

tutors

name	dept	number
John	CS	5
Naima	CS	5
Roy	Math	3
Alice	Math	55
Albert	Math	4

Three relations:

lives $\subseteq$ names **x** dorms

takes $\subseteq$ names **x** depts **x** numbers

tutors $\subseteq$ names **x** depts **x** numbers

Relational Database Example

Sample Queries:

Who lives in South dorm?

lives(X, 'South')

Who lives in East dorm and takes CS 60?

lives(X, 'East'), takes(X, 'CS', 60)

Who takes a CS course?

takes(X, 'CS', _)

lives

name	dorm
John	East
Naima	South
Alice	West
Toshiko	East
Roy	North
Albert	South

takes

name	dept	number
John	CS	60
Naima	CS	60
Alice	CS	5
Toshiko	CS	5
Albert	CS	60
Roy	Math	55
Naima	Math	55
Alice	Math	70
Toshiko	Math	80
Albert	Math	55

tutors

name	dept	number
John	CS	5
Naima	CS	5
Roy	Math	3
Alice	Math	55
Albert	Math	4

Quiz 2

Express as Prolog Queries:

Who takes a CS course and tutors a Math course?

What tutors live in West dorm?

Who lives in East dorm that is not a tutor?

lives

name	dorm
John	East
Naima	South
Alice	West
Toshiko	East
Roy	North
Albert	South

takes

name	dept	number
John	CS	60
Naima	CS	60
Alice	CS	5
Toshiko	CS	5
Albert	CS	60
Roy	Math	55
Naima	Math	55
Alice	Math	70
Toshiko	Math	80
Albert	Math	55

tutors

name	dept	number
John	CS	5
Naima	CS	5
Roy	Math	3
Alice	Math	55
Albert	Math	4

Previous Example Prolog KB

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).

% lives(N, D) means that person named N lives in dorm D

lives(john,    east).
lives(naima,   south).
lives(alice,   west).
lives(toshiko, east).
lives(roy,     north).
lives(albert,  south).

% takes(N, D, C) means that person named N takes course C in department D

takes(john,    cs,   60).
takes(naima,   cs,   60).
takes(alice,   cs,   60).
takes(toshiko, cs,    5).
takes(albert,  cs,   60).
takes(roy,     math, 55).
takes(naima,   math, 55).
takes(alice,   math, 70).
takes(toshiko, math, 80).
takes(albert,  math, 55).

% tutors(N, D, C) means that person named N tutors course C in department D

tutors(john,   cs,    5).
tutors(naima,  cs,    5).
tutors(roy,    math,  3).
tutors(alice,  math, 55).
tutors(albert, math,  4).
```

Solving Goals with Variables

- Variables get **bound** during matching.
- They get **unbound** during back-tracking, but never before.
- They may then be **re-bound**.
- Somehow this all can be viewed declaratively.

Goal Succession: Depth-First Execution in Prolog: Query 1

```
canTutor(alice, Y).
```

variable, since starts with upper-case

Goal Succession:
Depth-First Execution in Prolog: Chaining

```
canTutor(alice, Y).
```

Yellow denotes instance of rule or fact in knowledge base.

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

Same variable in original goal

Goal Succession: Depth-First Execution in Prolog: Binding

```
canTutor(alice, Y).
```

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

Yellow denotes instance of rule or fact in knowledge base.

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

```
tutors(alice, math, 55).
```

Green denotes variable **binding.**
Dept = math
Number = 55

```
takes(Y, math, 55)
```

Goal Succession: Depth-First Execution in Prolog: Result 1a

```
canTutor(alice, Y).
```

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

Yellow denotes instance of rule or fact in knowledge base.

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

```
tutors(alice, math, 55).
```

Green denotes variable **binding**
Dept = math
Number = 55

```
takes(Y, math, 55).
```

```
takes(roy, math, 55).
```

Y = roy

result variable binding

(empty)

Goal Succession:
Undoing Binding on Failure

```
canTutor(alice, Y).
```

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

Yellow denotes instance of rule or fact in knowledge base.

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

```
tutors(alice, math, 55).
```

Red denotes variable **binding**
Dept = math
Number = 55

```
takes(Y, math, 55).
```

undo former binding; try for another result

```
takes(roy, math, 55).
```

Goal Succession: Retrying

```
canTutor(alice, Y).
```

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

Yellow denotes instance of rule or fact in knowledge base.

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

```
tutors(alice, math, 55).
```

Red denotes variable **binding**
Dept = math
Number = 55

former binding undone

```
takes(Y, math, 55).
```

Backtracking in Depth-First Search
Rebinding: Result 1b

```
canTutor(alice, Y).
```

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

```
tutors(alice, Dept, Number), takes(Y, Dept, Number).
```

```
tutors(alice, math, 55).
```

Dept = math
Number = 55

```
takes(Y, math, 55).
```

new binding

```
takes(naima, math, 55).
```

(empty)

Y = naima

result binding

Deeper Backtracking: Query 2, Result 2a

canTutor(**X**, **Y**).

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

tutors(**X**, Dept, Number), takes(**Y**, Dept, Number).

tutors(john, cs, 5).

X = john
Dept = cs
Number = 5

takes(**Y**, cs, 5).

takes(toshiko, cs, 5).

(empty)

X = john
Y = toshkio

result binding

Deeper Backtracking

canTutor(X, Y).

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

tutors(X, Dept, Number), takes(Y, Dept, Number).

undo former binding; try for another result

tutors(naima, cs, 5).

X = naima
Dept = cs
Number = 5

takes(Y, cs, 5).

takes(toshiko, cs, 5).

(empty)

X = john
Y = toshkio

result binding

Deeper Backtracking

canTutor(X, Y).

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

tutors(X, Dept, Number), takes(Y, Dept, Number).

tutors(roy, math, 3).

X = roy
Dept = math
Number = 3

takes(Y, math, 3).

fails

Deeper Backtracking

canTutor(X, Y).

```
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).
```

tutors(X, Dept, Number), takes(Y, Dept, Number).

```
tutors(alice, math, 55).
```

X = alice
Dept = math
Number = 55

takes(Y, math, 55).

etc.

Summary of Backtracking

- Given a goal, Prolog tries rules **in order of occurrence** ("top-to-bottom"), using the first rule, the consequent of which **matches** the goal.
- If the rule has sub-goals, the sub-goals are satisfied **in order of occurrence** ("left-to-right"), resulting in bindings at each stage.
- If a goal sub-goal fails completely, Prolog **retries** to satisfy it using the next available option (e.g. the next rule).

Rule and Sub-Goal Ordering

Suppose the goal is `knows(john, Y, R).`

This rule is tried first.

```
knows(X, Y, living) :-
    lives(X, Z),
    lives(Y, Z).
```

This sub-goal is satisfied first, which binds Z.

This sub-goal is satisfied next.

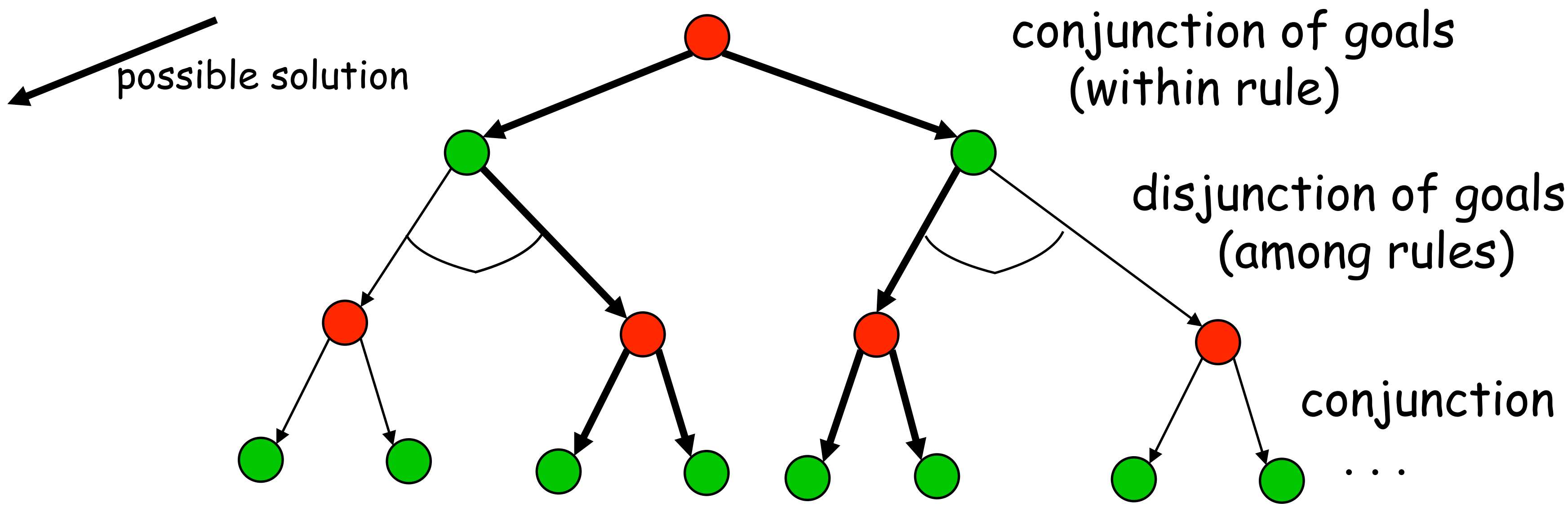
This rule is tried after the first rule is **exhausted**.

```
knows(X, Y, tutoring) :-
    canTutor(Y, X).
```

In effect, we have **disjunction** (*or*) **among** rules,
and **conjunction** (*and*) **within** rules.
Remember that Prolog execution is **depth-first search**.

And-Or Trees

- In AI, problem-solving trees are typically “And-Or” trees.
- This applies to Prolog’s goals.



“Logical Variables” in Prolog

- A variable in Prolog is like an **object** that can have one of two states:
  - unbound
  - bound, to some Prolog term, e.g. an individual
- Once the variable is bound, it only gets re-bound in **backtracking**, which results in removing the former binding first.

Lists in Prolog

- Unlike Scheme lists, Prolog lists use square brackets and require comma separators:
  - [a, b, 123]
  - [[foo, bar], []]

- There is no explicit cons, first, rest. Rather unification is used for this purpose:

  - [First | Rest] = [1, 2, 3, 4]
    unifies with First $\leftarrow$ 1, Rest $\leftarrow$ [2, 3, 4]

  - [First | Rest] does not unify with []

Lists in Prolog

- Second, third, etc. are also not necessary:

  [First, Second, Third | More] = [1, 2, 3, 4, 5]

  unifies with:

  First $\leftarrow$ 1, Second $\leftarrow$ 2, Third $\leftarrow$ 2,
  More $\leftarrow$ [4, 5]

Movie Database (used in hw8)

- % movie([Title, Year], Director, Categories), e.g.

  ```
  movie(['Being John Malkovich', 1999], 'Spike Jonze',
          [comedy, fantasy]).
  ```

- % actress(Name, [Birth City, State], Year), e.g.

  ```
  actress('Drew Barrymore', ['Culver City', 'California'], 1975).
  ```

- % actor(Name, [Birth City, State], Year), e.g.

  ```
  actor('Ben Affleck', ['Berkeley', 'California'], 1972).
  ```

- % plays(Player, Part, [Title, Year]), e.g.

  ```
  plays('Ben Affleck', 'Rafe',   ['Pearl Harbor', 2001]).
  ```

Numeric Aspects

- Numbers can be compared like any other goal:
  - 2 < 3   succeeds
  - 5 =< -5   fails

- Numeric comparisons (caution):
  - <   >   =<   >=   =\=   =:=

Numeric Operators: Different!!

- 2+3 is not 5
  It is an unevaluated term, effectively
  +(2, 3)

- The 'is' operator causes evaluation:
  X is 2+3
  binds X to 5.

Numeric relations will also cause evaluation

- 2 < 3+4 ok as a goal.
  - *is* is not needed here; Evaluation is forced.
- Most numeric functions are *not reversible* as regular goals are.
  - 5 is X+4 **won't** solve for X if it isn't bound.
- Arguments to arithmetic functions must be already bound.

Non-deterministic Programming

- One interpretation of “non-deterministic”:
  - Find *all* solutions by finding one solution.
  - Solutions can here be for the overall problem or a sub-problem.

Example of ND Programming

- member(X, [X | _]).
- member(X, [_ | L]) :- member(X, L).

- We think about this as either:
  - checking
  - generating

Generating vs. Testing

Consider:

```
member(X, [X | _]).

member(X, [_ | L]) :- member(X, L).
```

This predicate can be viewed as a member **tester**.

It can also be viewed as a member **generator**.

Generating vs. Testing

test

```
?- member(3, [1, 2, 3, 4, 5]).

yes

?- member(6, [1, 2, 3, 4, 5]).

no
```

generate

```
?- member(X, [1, 2, 3, 4, 5]).

X = 1 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

no
```

Exercise

- Extend member to have a 3rd argument: the **residue** left after the first element is removed from the list:

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

X = 1,
R = [2, 3, 4] ;

X = 2,
R = [1, 3, 4] ;

X = 3,
R = [1, 2, 4] ;

X = 4,
R = [1, 2, 3] ;

No
```

Generating with **append**

```
append ([], M, M).

append ([A | L], M, [A | N]) :-
       append (L, M, N).
```

functional

```
?- append ([1, 2, 3], [4, 5], Z).

Z = [1,2,3,4,5] ;

no
```

relational

```
?- append (X, Y, [1, 2, 3, 4, 5]).

X = [ ],
Y = [1,2,3,4,5] ;

X = [1],
Y = [2,3,4,5] ;

X = [1,2],
Y = [3,4,5] ;
…
X = [1,2,3,4,5],
Y = [ ] ;

no
```

Using a Generator as a "for" loop

```
?- for(I, 5, 8).

I = 5 ;

I = 6 ;

I = 7 ;

I = 8 ;

No
```

Definition:

```
for(M, M, N) :- M =< N.
for(I, M, N) :-
        M < N,
        M1 is M+1,
        for(I, M1, N).
```

Caution: Won't work in reverse, due to *is*.

Generating an Infinite Set

```
?- for(I, 5).

I = 5 ;

I = 6 ;

I = 7 ;

I = 8 ;

.
.
.
```

Definition:

```
for(M, M).
for(I, M) :-
        M1 is M+1,
        for(I, M1).
```

Caution: Won't work in reverse, due to *is*.

Exercise: Generate all Pairs in N x N

```
?- pair(I, J).

I = 0
J = 0 ;

I = 0
J = 1 ;

I = 1
J = 0 ;

I = 0
J = 2 ;

I = 1
J = 1 ;
.
.
.
```

Example of ND Programming

- permutation(X, Y) is true if list Y is a permutation of list X.

- An attempt:
  permutation(X, Y) :- sort(X, Z), sort(Y, Z).

- This is logical, but doesn't work;
  the built-in sort is **uni-directional**.

Permutation

- ```
  permutation([], []).
  ```
- ```
  permutation(L, [A | M]) :-
          member(A, L, Residue),
          permutation(Residue, M).
  ```

generalized *member*

- ```
  member(A, [A | X], X).
  ```
- ```
  member(A, [B | X], [B | Y]) :-member(A, X, Y).
  ```

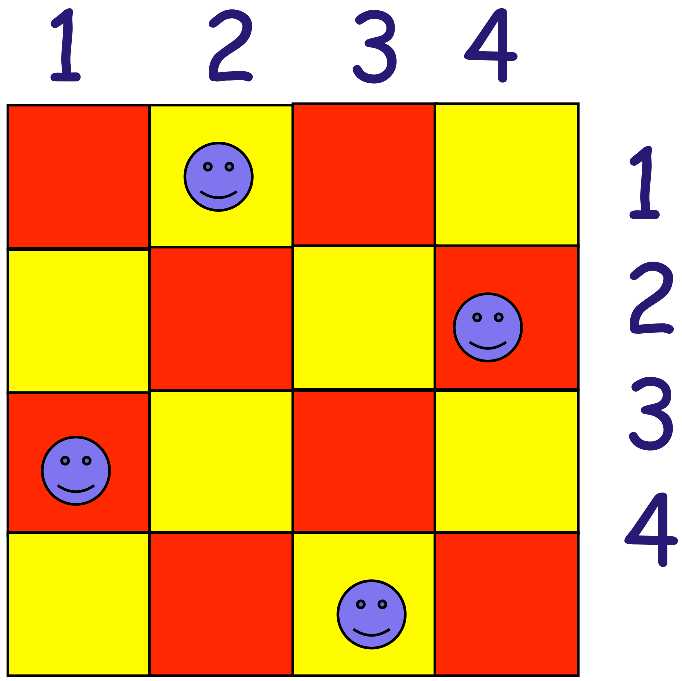
slowsort (joke)

```
% slowsort(X, Y) is true when Y is a sorted permutation of X.

slowsort(X, Y) :- permutation(X, Y), sorted(Y).

% sorted(Y) is true when Y is a list of elements in non-decreasing order

sorted([]).

sorted([_]).

sorted([A, B | X]) :- A @=< B, sorted([B | X]).
```

N-Queens Problem (NDP)

- Two queens on a chessboard are "attacking" if they are in a common row, column, or diagonal.
- Given a board size N, find a solution (or all solutions) for placing N queens so that no two are attacking.

Example, N = 4



State representation:

[[1, 3], [2, 1], [3, 4], [4, 2]]

Solving Queens

- Given a list of columns and unoccupied rows:

  If columns is empty, succeed.

  For the first column:

  If there is a row where the queen is not being attacked, place it and recurse.

  If no such row, fail (backtrack).

Queens (1/3)

```
queens(N, S) :-
    range(1, N, Range),
    solve(Range, Range, [], Solution),
    reverse(Solution, S).

% solve(Rows, Cols, Acc, Solution)

solve([], _, Acc, Acc).

solve([Col | Cols], Rows, Acc, Sol) :-
    member(Row, Rows, Residue),
    noAttack([Col, Row], Acc),
    solve(Cols, Residue, [[Col, Row] | Acc], Sol).
```

Queens (2/3)

```
% noAttack([Col, Row], Pairs) succeeds if pair [Col], Row] does not
% attack the other Pairs, assuming
% the other Pairs don't attack each other.

noAttack(_, []).
noAttack([Col1, Row1], [[Col2, Row2] | Pairs]) :-
    Col1 =\= Col2,
    Row1 =\= Row2,
    abs(Col1-Col2) =\= abs(Row1-Row2),
    noAttack([Col1, Row1], Pairs).
```

Queens (3/3)

```
member(A, [A | X], X).
member(A, [B | X], [B | Y]) :- member(A, X, Y).


range(M, N, []) :- M > N.
range(M, N, [M | L]) :- M =< N, M1 is M+1, range(M1, N, L).


reverse(L, M) :- reverse(L, [], M).

reverse([], M, M).
reverse([A | L], M, R) :- reverse(L, [A | M], R).
```

The "24" Game (homework problem)

- Given:
  - A set of positive integers: [2, 4, 5, 7]
  - A set of reusable operators: [+, -, *]
  - A target: 24
- Construct an expression (as a syntax tree) showing how to make the target from the integers.

Approach to 24: Non-Deterministic Programming

- If there is only one integer in the set:
  - There is no choice. Either it is the same as the goal or not.
- Otherwise:
  - Split the integers into two non-empty subsets.
  - Compute a tree that could be constructed from each of the subsets.
  - Choose an operator for the root joining the two trees.
  - Check to see whether the **overall** tree meets the given target.

Strong recommendations

- Don't **evaluate** a tree until the **final** tree is built.
- Don't try to optimize by evaluating sub-trees during the solution search (at least not for this assignment).

Prolog Uninterpreted Expressions

- Prolog has a built-in an infix operator precedence parser:
  - 3+4*5 is really:

    +(3, *(4, 5))

How can you be sure? Try unifying:

```
?- 3+4*5 = +(3, *(4, 5)).

Yes
```

Evaluating an Expression

- The *is* operator will evaluate an expression. = (unification) will not:

```
?- X is 3+4*5.

X = 23


?- X is +(3, *(4, 5)).

X = 23


?- 23 = 3+4*5.

No
```

```
?- X = +(3, *(4, 5)), Y is X.

X = 3+4*5,
Y = 23
```

Composing/Decomposing an Expression

- Infix operator =.. (called "univ") will build an expression from an operator and arguments, or take an expression apart:

```
?- X =.. [+, 3, 4].       % compose

X = 3+4


?- 3+4 =.. Y.             % decompose

Y = [+, 3, 4]
```

Splitting a List

- Only concerned with lists of 2 or more elements (why?)

```
?- split([1, 2, 3, 4], X, Y).

X = [1, 2, 3]
Y = [4] ;
        .
        .
        .
X = [1, 4]
Y = [2, 3]
```

Splitting a List

- The tricky part is making sure that you get **every** way of splitting.
- Ideally you get back each way of splitting only once.

Base Case

- A list of exactly two elements is easy to split.

How to split a list of > 2 elements

- Remove an element E from the list
  (using an extended 'member' predicate).
- Recursively split the remaining elements into two.
- Add E back to the first list.
- Exploit symmetry by using an auxiliary predicate and calling it twice from the interface predicate:
  - split(X, L, R) :- split2(X, L, R).
  - split(X, L, R) :- split2(X, R, L).

The solve24/4 predicate

- solve24(Ops, Values, Result, Exp) :- . . .
  - Ops is the set of operators.
  - Values is the list of values to be combined.
  - Result is the value of the expression , which will need to **unify** with the target if one is specified.
  - Exp is the expression itself.

Example

```
?- setof(Exp, solve([+, *, -], [2, 3, 4, 5], 24, Exp), Ans).

Ans =
                                [2* (3+ (4+5)),
                                 2* (3+ (5+4)),
                                 2* (4+ (3+5)),
                                 2* (4+ (5+3)),
                                 2* (5+ (3+4)),
                                 2* (5+ (4+3)),
                                 2* (3+4+5),
                                 2* (3+5+4),
                                 2* (4+3+5),
                                 2* (4+5+3),
                                        .
                                        .
                                        .
```

The “Zebra” Problem

(aka “Einstein’s Riddle”?)

Five people of different nationalities, with different occupations, live in consecutive houses on a street. These houses are painted different colors. Each person has a different pet and a different favorite drink.
Given:

1. The English person lives in the red house.
2. The Spanish person owns a dog.
3. The green house is on the right side of the white house.
4. The Italian drinks tea.
5. The Norwegian lives in the first house on the left.
6. The photographer breeds snails.
7. The Norwegian’s house is next to the blue one.
8. The Japanese person is a painter.
9. The fox is in a house next to that of the physician.
10. The diplomat lives in the yellow house.
11. The owner of the green house drinks coffee.
12. The violinist drinks orange juice.
13. The horse is in a house next to that of the diplomat.
14. Milk is drunk in the middle house.

Determine: ***who owns the zebra?; who drinks water?.***

Analysis

- There are 5 “houses”, which can be represented as a parenthesized structure:
  - (Nationality, Colo, Occupation, Pet, Drink)
- There is a list of 5 houses:
  - L = [_, _, _, _, _]
- Note that order is important in the list (left-to-right).
- Each clue places a constraint on the list.
- The questions to be answered are:
  - Find X where member((X, _, _, zebra, _), L).
  - Find Y where member((Y, _, _, _, water), L).

Translating Clues

1. The English person lives in the red house.

   ```
   clue1(L) :- nationality(english, H), color(red, H),          house(H, L).
   ```

2. The Spanish person owns a dog.

   ```
   clue2(L) :- nationality(spanish, H), pet(dog, H),            house(H, L).
   ```

3. The green house is on the right side of the white house.

   ```
   clue3(L) :- color(green, G), color(white, W),          rightof(G, W, L).
   ```

Helpers:

```
house(X, L) :- member(X, L).

rightof(X, Y, L) :- leftof(Y, X, L).

    leftof(X, Y, [X, Y | _]).              % assume "immediate" left of
leftof(X, Y, [_ | L]) :- leftof(X, Y, L).
```

Unifying with House Structures

```
nationality( N,  (N, _, _, _, _)).
color(        C,  (_, C, _, _, _)).
occupation( O,  (_, _, O, _, _)).
pet(          P,  (_, _, _, P, _)).
drink(        D,  (_, _, _, _, D)).
```

Using the Clues Together:

```
clues(L) :-
    clue14(L),        % strategic placement
    clue1(L),
    clue2(L),
    clue3(L),
    clue4(L),
    clue5(L),
    clue6(L),
    clue7(L),
    clue8(L),
    clue9(L),
    clue10(L),
    clue11(L),
    clue12(L),
    clue13(L),
    true.
```

```
solution(Z, W, L) :-
    clues(L),
    pet(zebra, H1),   nationality(Z, H1), house(H1, L),
    drink(water, H2), nationality(W, H2), house(H2, L).
```

Tester, with Uniqueness Test
(using if-then-else P -> Q; R)

```
test :-
   solution(Z, W, L)
     -> (
          solution(_, _, M), L \== M

          -> write('The solution is not unique.'),
             write('One solution is: '), nl, pprint(L), nl,
             write('Another solution is: '), nl, pprint(M)

           ; write('The solution is unique:'), nl,
             pprint(L), nl,
             write('The '), write(Z), write(' owns the zebra.'), nl,
             write('The '), write(W), write(' drinks water.'), nl
        )

     ; write('There is no solution'), nl.
```

Quiz 3: Translate the rest of the clues.

1. The English person lives in the red house.
   clue1(L) :- nationality(english, H), color(red, H), house(H, L).

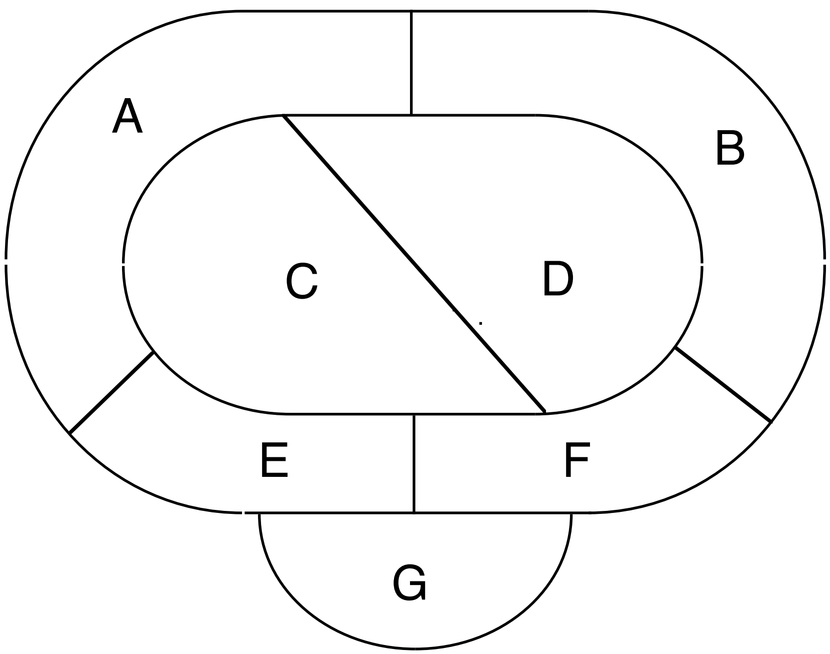
2. The Spanish person owns a dog.
   clue2(L) :- nationality(spanish, H), pet(dog, H), house(H, L).

3. The green house is on the right side of the white house.
   clue3(L) :- color(green, G), color(white, W), rightof(G, W, L).

7. The Italian drinks tea.
8. The Norwegian lives in the first house on the left.
9. The photographer breeds snails.
10. The Norwegian's house is next to the blue one.
11. The Japanese person is a painter.
12. The fox is in a house next to that of the physician.
13. The diplomat lives in the yellow house.
14. The owner of the green house drinks coffee.
15. The violinist drinks orange juice.
16. The horse is in a house next to that of the diplomat.
17. Milk is drunk in the middle house.

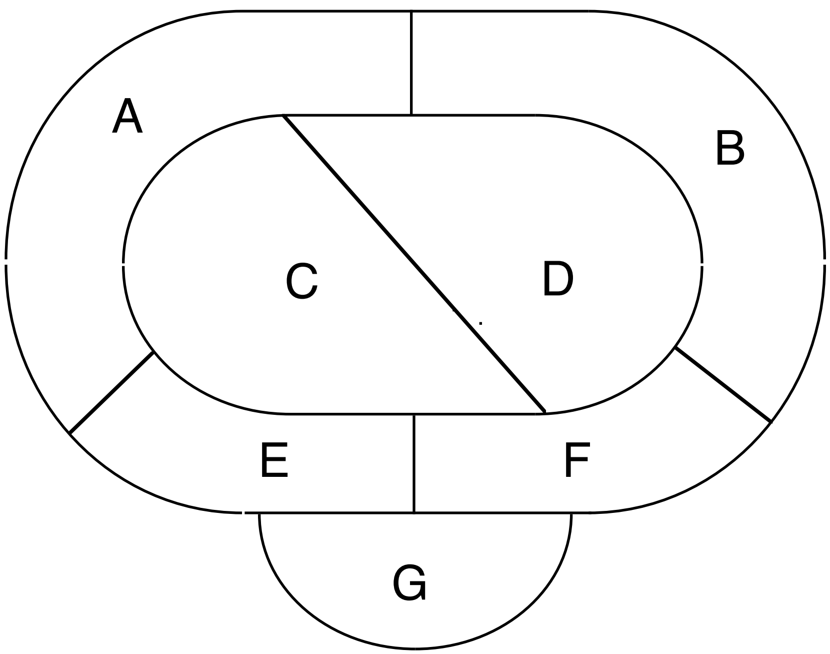
Generator/Test Example: Map Coloring

A map

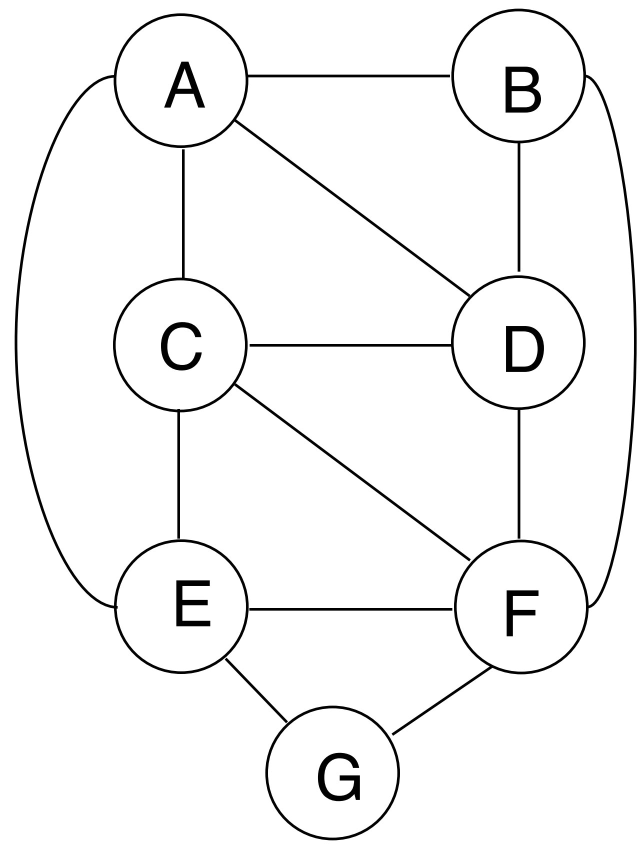


Map Coloring (2)

A map



Corresponding graph

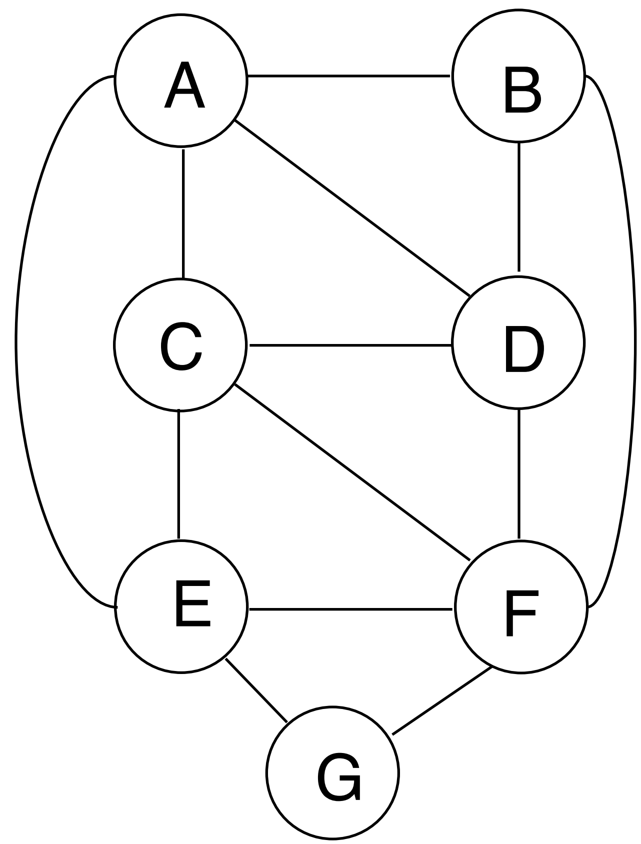


Map Coloring (3)

Prolog Clause

```
map([A, B, C, D, E, F, G]) :-
    next(A, B),
    next(A, C),
    next(A, D),
    next(A, E),
    next(B, D),
    next(B, F),
    next(C, D),
    next(C, E),
    next(C, F),
    next(D, F),
    next(E, F),
    next(E, G),
    next(F, G).
```

Graph



Map Coloring (4): Color Constraints

```
next(X, Y) :- color(X), color(Y), X \== Y.
```

means individuals are **not equal**

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

colors to be used

These and the preceding clause are the entire program.

Sudoku

- Sudoku is basically a graph-coloring problem.
- Adjacency is no longer binary. Any two squares in the same row, column, or sub-square are considered "adjacent".

Sudograph (pseudo-graph?)

- G =
  - List of Nodes (logical variables)
  - List of constraints
    - Each constraint is a list of variables
    - No two variables in the same constraint can have the same node values.

Sudograph setup

```
test(Nodes, Constraints, Colors):-
    Nodes = [A, B, C, D, E, F, G],
    Constraints = [[A, B, C],
                   [B, C, D],
                   [C, D, E],
                   [D, E, F],
                   [E, F, G],
                   [G, A, B]],
    Colors = [red, blue, green, yellow],
    sudographSolver(Nodes, Constraints, Colors).
```

```
?- test(Nodes, Constraints, Colors).
Colors: [red, blue, green, yellow]
Nodes:  [red, blue, green, red, blue, green, yellow]
Constraints:
       [red, blue, green]
       [blue, green, red]
       [green, red, blue]
       [red, blue, green]
       [blue, green, yellow]
       [yellow, red, blue]
```

More on Predicate Logic:

Quantifiers

Quantifiers

- In addition to truth function operators of proposition logic, predicate logic introduces **quantifiers** for expressing variation over individuals:

$$(\forall x)\ p(x) : \text{for all } x,\ p(x)$$

universal quantifier

$$(\exists x)\ p(x) : \text{for some } x,\ p(x)$$

existential quantifier

Order of Quantifiers

- $(\forall x)\ (\exists y)$ knows(x, y):
  Everyone knows someone.

- $(\exists x)\ (\forall y)$ knows(x, y):
  Someone knows everyone.

- $(\exists x)\ (\forall y)\ \neg$knows(x, y)
  Someone knows no one.

- $(\exists x)\ (\exists y)$ knows(x, y) $\wedge\ x \neq y$
  Someone knows someone other than him/herself.

Quantifiers in Prolog

- In most formulas, quantifiers are implicit:
  - If a variable appears in the head, it is for-all quantified in the rule.
  - If a variable appears in the body, but not the head, it is there-exists quantified.

- Examples:
  - p(X, Y) :- q(X), r(X, Y) says:

    (∀x) (∀y) [if (q(x) and r(X, Y)) then p(X, Y)].

  - p(X) :- q(X), r(X, Y) says:

    (∀x) [if (∃y) (q(x) and r(X, Y)) then p(X)].

Quantifiers in Prolog

- The ∃ can be made explicit:

- Examples:
  - p(X) :- q(X), r(X, Y) says:

    (∀x) [if (∃y) (q(x) and r(X, Y)) then p(X)].

  - p(X) :- Y^(q(X), r(X, Y)) says the same thing.

  - ^ is an "infix" version of ∃.

Where it Really Matters: setof

- Consider
  - setof(X, p(X, Y), Z).
- How is Y quantified? If you want it to be ∃, the usual case, use:
  setof(X, Y^p(X, Y), Z).
- If you leave it off, it is a free variable, and may become bound in solving, **in which case all other solutions would use the same** Y.
- You won't get all solutions for all Y in this case.
- Typical use of the unquantified version:
  - r(X, Z) :- setof(X, p(X, Y), Z).
  - Here there is a set of Z for **each** possible X.

Special handling of _ in Prolog

- The variable _ is special.
- It is called a “throw-away” or “don't care” variable.
- _ unifies with anything, but different instances of _ within the same clause are *not* unified, unlike other variables.

Other variables beginning with _

- A variable that occurs only once in a clause is called a "**singleton variable**".
- Often singleton variables are the result of a typing error, and certain **compilers will warn about them**.
- **To prevent the warning**, when this is the intention, use a variable that begins with _, such as _Name rather than Name.

== in Prolog is not unification

- == is literal equality
- a == a succeeds
- a == b fails
- X == a fails if X is unbound (unlike =)
- X = a, X == a succeeds (X becomes bound)
- X == Y fails if either is unbound

\== in Prolog is literal inequality

- a \== a fails
- a \== b succeeds
- X \== Y succeeds if either is unbound

- There is no \= (not-unifiable) operator.
- Instead use \+ X = Y (it is not the case that X = Y).

Other comparison operators

- @<   compare arbitrary terms (e.g. lists)
- @>   in lexicographic order
- @=<
- @>=

Some Reversible Arithmetic can be Simulated with Lists

Number N is represented as a list of N 1's

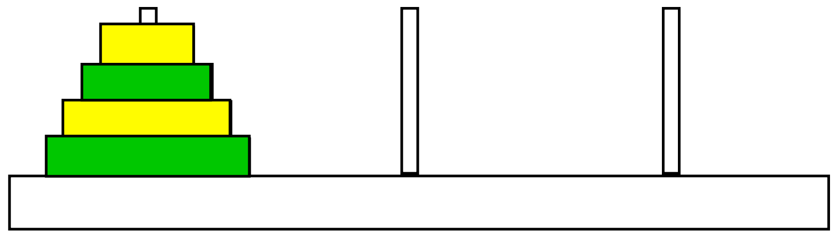
```
sum([ ], Y, Y).

sum([1 | X], Y, [1 | Z]) :- sum(X, Y, Z).
```

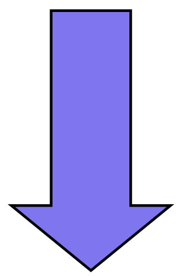
The following doesn't quite work for all inverses. A problem arises in factoring 0.

```
prod([ ], Y, []).

prod([1 | X], Y, Z) :- prod(X, Y, Z1),
                       sum(Z1, Y, Z).
```

```
| ?- sum([1,1,1], [1,1], Z).

Z = [1,1,1,1,1]

| ?- sum(X, Y, [1,1,1,1,1]).

X = [],
Y = [1,1,1,1,1] ;

X = [1],
Y = [1,1,1,1] ;

X = [1,1],
Y = [1,1,1] ;
…                        (abridged)
X = [1,1,1,1,1],
Y = [] ;

no
```

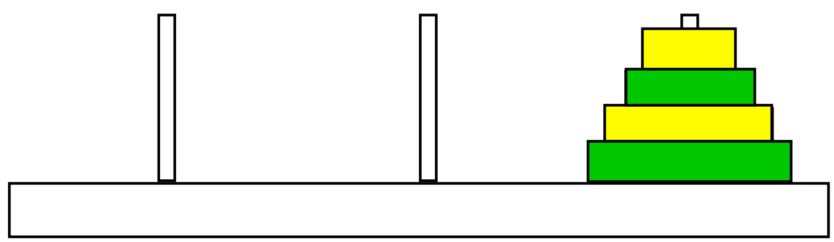
Example: Towers of Hanoi





Move only one disk at a time.

Never place a larger disk on a smaller one.



Solving Towers of Hanoi

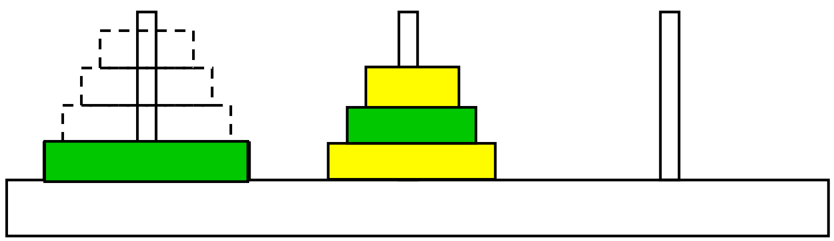
- Some approaches:
  - Pre-programmed solution
    - Recursive solution is easy in most languages
  - Let prolog find solution using depth-first search
    - Trickier, but shows off Prolog's capabilities
    - May not find shortest solution
  - Program breadth-first search in Prolog
    - Still trickier
  - Program iterative-deepening search
    - Easier than breadth-first

Pre-Programmed Towers of Hanoi (1)

- To move N disks
  from stack ***From***
  to stack ***To***:

Pre-Programmed Towers of Hanoi (2)

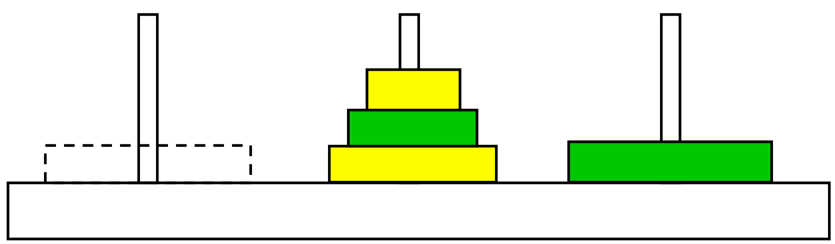
- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack Other (the stack other than ***From*** and ***To***)



A key point throughout is that the N-1 disk moves can be done without violating the constraint that a larger disk not be put atop a smaller one.

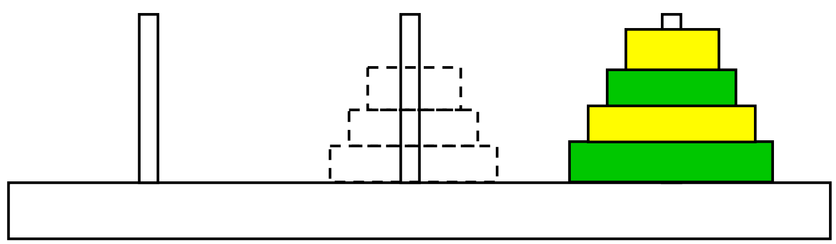
Pre-Programmed Towers of Hanoi (3)

- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack ***Other*** (the stack other than ***From*** and ***To***)
  - Move 1 disk from stack ***From*** to stack ***To***



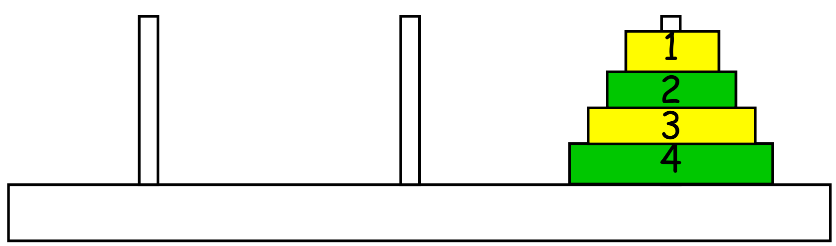
Pre-Programmed Towers of Hanoi (4)

- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack ***Other*** (the stack other than ***From*** and ***To***)
  - Move 1 disk from stack ***From*** to stack ***To***
  - Move N-1 disks from stack ***Other*** to stack ***To***



Data Representation

- Number the disks 1, 2, 3, . . . smallest to largest.
- Use numeric value to detect size constraint.



Pre-Programmed Towers of Hanoi(5)

```
% towers(N, From, To, Moves) means that Moves is the list of
% moves to move N disks from stack From to stack To

towers(N, From, To, Moves) :-
    towers(N, From, To, [ ], ReversedMoves),
    reverse(ReversedMoves, Moves).


% towers(N, From, To, Acc, Moves) means that Moves is the reverse of the lis
% of moves to move N disks from stack From to stack To, with
% Acc being the reverse of the accumulated moves going in (to avoid appendi

% towers(N, From, To, Acc, Moves).

towers(0, _, _, Acc, Acc).

towers(N, From, To, Acc, Moves) :-
    other(From, To, Other),
    N1 is N - 1,
    towers(N1, From, Other, Acc, Moves1),
    towers(N1, Other, To, [ [From, To] | Moves1], Moves).
```

```
other(1, 2, 3).
other(1, 3, 2).
other(2, 1, 3).
other(2, 3, 1).
other(3, 1, 2).
other(3, 2, 1).
```

Depth-First
Towers of Hanoi

Depth-First Towers of Hanoi (1)

Does not require a human to solve the puzzle first

First characterize the possible moves.

This is a move from stack 1 to stack 2:

from / to stack 1 before stack 2 after

```
move([1, 2], [[F1 | R1], S2, S3], [R1, [F1 | S2], S3]) :-
        ok(F1, S2).
```

provided that it is ok to move disk F1 onto stack S2

Depth-First Towers of Hanoi (2)

All the possible moves in six rules:

```
move([1, 2], [[F1 | R1], S2, S3], [R1, [F1 | S2], S3]) :- ok(F1, S2).
move([1, 3], [[F1 | R1], S2, S3], [R1, S2, [F1 | S3]]) :- ok(F1, S3).
move([2, 1], [S1, [F2 | R2], S3], [[F2 | S1], R2, S3]) :- ok(F2, S1).
move([2, 3], [S1, [F2 | R2], S3], [S1, R2, [F2 | S3]]) :- ok(F2, S3).
move([3, 1], [S1, S2, [F3 | R3]], [[F3 | S1], S2, R3]) :- ok(F3, S1).
move([3, 2], [S1, S2, [F3 | R3]], [S1, [F3 | S2], R3]) :- ok(F3, S2).
```

from / to   state before   state after   condition

Depth-First Towers of Hanoi (3)

When is it ok to move a disk onto a stack?
Assume the disks are represented by numbers 1, 2, 3, …
with smaller numbers representing smaller disks.

```
ok(_, [ ]).
```

empty target stack

```
ok(A, [B | _]) :- smaller(A, B).


smaller(A, B) :- A < B.
```

Depth-First Towers of Hanoi (4)

`towers([S1, S2, S3], Moves)` will mean that Moves is a valid move sequence that results in S1 and S2 being empty (so all disks are on S3).

`towers([S1, S2, S3], Seen, Moves)` means the same, except that Seen will be a list of all previous states (to prevent infinite looping).

```
towers(InitialState, Moves) :- towers(InitialState, [ ], Moves).
```

```
towers([[ ], [ ], _], _, [ ]). % final state, no more moves
```

```
towers(Before, Seen, [Move | Moves]) :-
    nonMember(Before, Seen),
    move(Move, Before, After),
    towers(After, [Before | Seen], Moves).
```

only consider if Before not already seen

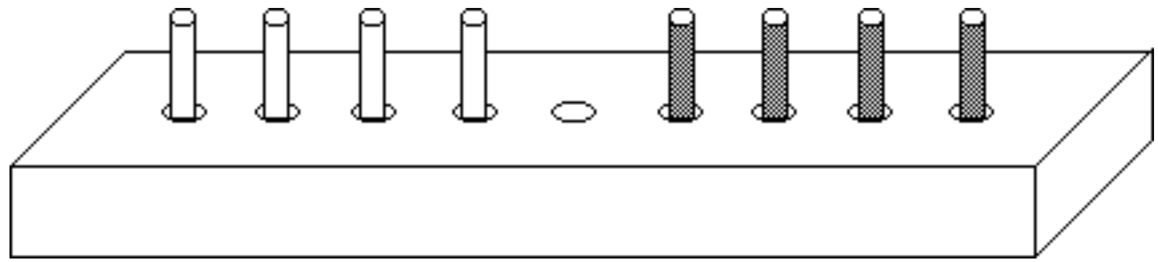
recurse

Depth-First Towers of Hanoi (5)

Auxiliary Predicates:

```
nonMember(X, L) :- \+ member(X, L).

member(X, [X | _]).

member(X, [_ | L]) :- member(X, L).
```

Exercise



Reverse the pegs by moving peg "forward" or jumping forward over a peg of either color.

Work out a depth-first solution in Prolog.

(You don't have to check for cycles, because there can't be any.)

Prolog Perspective

- A complete programming language
- Not a complete logic language
  - Restricted to "Horn Clauses"
  - Restricted form of negation
  - Quantifiers not completely general
  - Builtin arithmetic not reversible
- More powerful logic systems exist, e.g.
  - Otter (see CS 80 or 151)

Contemporary Extensions of Prolog

- Constraint logic programming
- Inductive logic programming
- Lambda-prolog
- Goedel
- Parallel prologs
- Prolog++
- ... (The list is quite long.)