Predicate Logic

Predicate Logic

- Proposition logic (discused previously) deals with propositions (self-contained things that can be true or false). Individuals are not treated as entities.
- **Predicate** logic constructs truth values out of predicates that apply to **individuals**.
- A predicate **with all argument individuals specified** is like a proposition, in that it has a true or false value.

Example

- Suppose *knows* is a 2-ary predicate.
- Then knows(x, y) is true or false for any substitution of individuals for x and y.
  - knows("John", "Sally") is true or false
  - knows("Sally", "Jane") is true or false
  - etc.

Quantifiers

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

$(\forall x)$ p(x) : for all x, p(x)

universal quantifier

$(\exists x)$ p(x) : 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.

Interpretation of Formulas with Quantifiers

- An **interpretation** of the symbols in a formula assumes a set of individuals over which we are quantifying, called the **domain**.
- The domain may be finite or infinite.
- An interpretation also assumes a **specific predicate** for each predicate symbol.

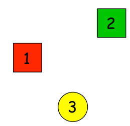
Interpretations Provide Meaning

- A predicate logic formula can be true in some interpretations but not in all.
- Some formulas are true in all interpretations.
- Some are never true.

Fun with Specific Interpretations

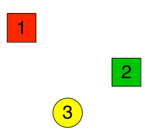
triangle(x) means x is a triangle
square(x) means x is a square
circle(x) means x is a circle
red(x) means x is red, etc.

above(x, y) means x is above y
left(x, y) means x is to the left of y



Which are true in the interpretation shown?
red(3)
square(2)
above(2, 1)
left(3, 1)
left(1, 3) □ (red(2)   yellow(3))

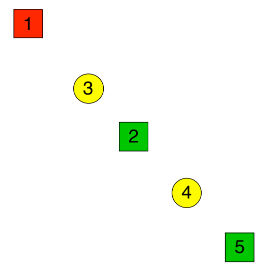
Specific Interpretations that quantify over individuals



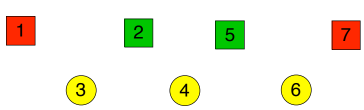
Which are true in the interpretation shown?
(□x) (square(x)   circle(x))
(□x, y) ((square(x) □ circle(y)) □ above(x, y))
(□x, y) ((square(y) □ circle(x) □ left(x, y))
(□x) (left(x, x))
(□x) square(x) □ (red(x)   green(x))

Give Formulas that Characterize the Interpretation

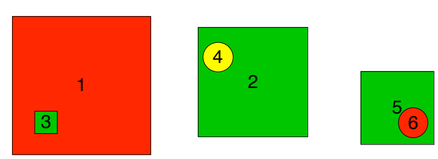
(without referring to specific individuals)



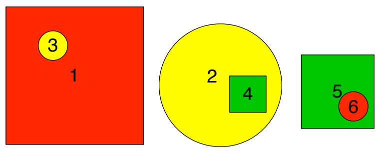
Give Formulas that Characterize the Interpretation



Give Formulas that Characterize the Interpretation



Give Formulas that Characterize the Interpretation



Formulas Valid for the Interpretation Natural Numbers

- (□x) ((x = 0) v (□y) (x = S(y))
- (□x) □(S(x) = 0)
- (□x, y) S(x) = S(y) □ x = y
- (p(0) □ (□x) (p(x) □ p(S(x)))) □ (□x) p (x)
- Here
  - S represents the *successor* function (S(x) = x+1)
  - 0 is an "individual" symbol
  - p is any predicate
- These form a variant on the Peano axioms (Guiseppe Peano, 1889).

Universally **Valid** Formulas are true regardless of interpretation

- We always assume the domain is *non-empty*.
- (□x) p(x) □ (□x) p(x)
- (□x) (p(x) □ q(x)) □ ((□x) p(x)) □ ((□x) q(x))
- ((□x) p(x) v (□x) q(x)) □ (□x) (p(x) v q(x))
- (□x) (p(x) □ q(x)) □ ((□x) p(x)) □ ((□x) q(x))
- (□x) p(x) □ □(□x)□p(x)
- (□x)(□y) p(x, y) □ (□y)(□x) p(x, y)
- not valid: (□x)(□y) p(x, y) □ (□y)(□x) p(x, y)

Uses of Predicate Logic

- Querying databases
- Program specification
- Program verification
- Advanced properties of logic circuits (time dependence, etc.)
- We look at the first three in this course.

Querying Databases

- For now, restrict to
  "**relational**" **databases**
- View each table in database as a relation (=□predicate)
- Query the database:
  Predicate logic expression selects those combinations for which the expression evaluates to true

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: names x dorms

takes: names x depts x numbers

tutors: names x depts x numbers

Relational Database Example

Sample Queries:

Who lives in South dorm?

x: lives(x, South)

Who lives in East dorm and takes CS 5?

x: lives(x, East) □ takes(x, CS, 5)

Who takes a CS course?

x: (□y) takes(x, CS, y)

Relational Database Example

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?

Predominant Database Languages

- SQL (sometimes pronounced "sequel")
  - **Structured Query Language**
  - The standard query language used in most commercial relational database systems (Oracle, Informix, Sybase, etc.)
  - Invented by Don Chamberlin, HMC '66
- Prolog (sometimes pronounced "prolog")
  - **Programming in Logic**
  - A complete programming language
  - Used in AI and rapid prototyping

Prolog Tutorial

- Relations can be expressed in two ways:
  - Enumeration
  - Rules
  - Combinations of both are possible
- **Highly case-sensitive**
  - **Predicates and constants *begin with* lower-case**, unless quoted with single quotes '….'
  - **Variables *begin with* uppper-case!**
    - _ is considered upper-case
    - _ alone does not match others
  - "…" is not used for quoting; it means something else.

Prolog Tutorial

Enumeration of the *lives* relation in Prolog:

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

called "clauses"

Enumeration of the *tutors* relation in Prolog:

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

Prolog Tutorial

- Typical developmental execution (as□pposed to complete application) scenario:
  - Knowledge Base (= Database+Rules) is loaded ("consulted").
  - Queries are posed based on loaded database.

Prolog on Turing
(text in red is typed by user)

```
turing ~:1> prolog +l tutors.pl
Quintus Prolog Release 3.2 (Sun 4, SunOS 5.3)
Copyright (C) 1994, Quintus Corporation.  All rights reserved.
Licensed to Harvey Mudd College, CS Dept.

| ?- lives(john, X).              % Where does john live?

X = east
| ?- lives(X, east).              % Who lives in east?

X = john ;
X = toshiko ;

no
```

variable, since starts with upper-case

individual, since starts with upper-case

Prolog Commenting

- % indicates comment from here to end of line
- /* . . . */ is comment for arbitrary number of lines
- // is integer division

Prolog Rule Syntax

- A clause such as
  lives(john,east).
  is called a **unit clause** or **fact**; it refers to one piece of information.
- Non-unit clauses typically are in the form of reverse **implications**:
  Consequent :- *Antecedent*.
  which stand for
  Antecedent **implies** *Consequent*.

Consequent :- *Antecedent*.

- Can be read as any of the following:
  - *Antecedent* **implies** *Consequent*
  - *Consequent* **is implied by** *Antecedent*
  - *Consequent* **if** *Antecedent*
  - *Consequent* **provided** *Antecedent*
- These are called the ***logical interpretation*** of a clause (as distinguished from *procedural* interpretation).

Non-Unit Clause Examples

```
livesInEast(X) :- lives(X, east).
```

variable

individual

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

Comma in this context is *and*.

Variables occurring in the **consequent** are implicitly *universally* (i.e. □) quantified.

Existential Variables

A variable occurring in the antecedent but *not* in the consequent is implicitly existentially (i.e. □) quantified.

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

Means "(For all X, Y),
if there is ***some*** Z such that
lives(X, Z) and lives(Y, Z),
then knows(X, Y)."

Multiple Clauses for One Predicate

Expressing two different ways for X to know Y:

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

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

Logic of Passing an Exam

- There are two ways for a person X to pass an exam:
  - X is adequately prepared, or
  - the exam is extremely easy

```
pass_exam(X) :- prepared_for_exam(X).

pass_exam(X) :- easy_exam, person(X).
```

This is like a **proposition variable**. It is equivalent to a predicate with **no arguments**, and **parens are not used**.

Preparing for an Exam (1)

- There are three ways for X to be prepared:
  - X knows it all

```
prepared_for_exam(X) :-
    knows_it_all(X).
```

Preparing for an Exam (2)

- X read the book, attended classes (without sleeping), and worked the problems:

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

Note: \+ is Prolog's version of *not*.

Preparing for an Exam (3)

- X was tutored by Y, who was prepared for the exam:

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

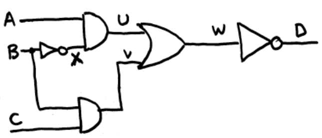
Who passes the exam?

```
person(mary).
person(john).
person(tom).
person(sally).
person(fred).

read_book(fred).
read_book(mary).

worked_problems(fred).
worked_problems(mary).
```

```
attended_lectures(fred).
attended_lectures(mary).

slept_during_lectures(fred).

knows_it_all(tom).

tutored_by(john, mary).
tutored_by(sally, john).
```

Simulating a Logic Circuit

```
circuit.pl        Tue Mar 11 22:39:03 1997        1

% an example logic circuit

circuit(A, B, C, D) :-
  not(B, X),
  and(A, X, U),
  or(U, V, W),
  and(B, C, V),
  not(W, D).

% definitions of logic elements

and(0, 0, 0).
and(0, 1, 0).
and(1, 0, 0).
and(1, 1, 1).

or(0, 0, 0).
or(0, 1, 1).
or(1, 0, 1).
or(1, 1, 1).

not(0, 1).
not(1, 0).
```



Goal-Oriented (Procedural) Interpretation of Prolog

- The execution of Prolog is actually a form of **depth-first search**.
- A Prolog **query** is composed of a list of **goals** (the individual predicate expressions).
- Prolog tries to solve these goals by **finding individuals** that **satisfy** the predicates, as determined by the knowledge base.
- During the solving process, a goal is replaced by other goals, according to the **rules**, until there are no unsolved goals left.

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).
```

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**).

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).

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

undo former binding; try for another result

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

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**).

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).

result binding

Y = naima

(empty)

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).

result binding

(empty)

X = john
Y = toshkio

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).

result binding

(empty)

X = john
Y = toshkio

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.

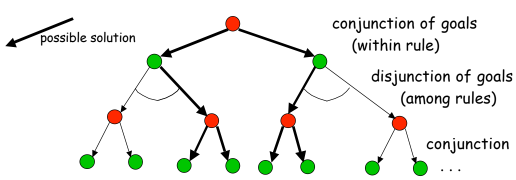
This rule is tried after the first rule is **exhausted**.

```
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

- The rex list notation was derived from Prolog's list notation; the two are pretty much the same.
- A list can contain logical variables, which are already bound, or may get bound **later**.
- The process of binding is known as **unification** (which means "make the same").

Unification (1)

- Unification causes two logical variables to denote the same thing.
- The symbol for unification in Prolog is =.
- Examples:
  - X = a   Variable unified with constant
  - X = [a, b, c]   Variable unified with a list
  - [X, b, c] = [a | Y]   X = a, Y = [b, c]
  - [X, b] = [a, c]   NOT UNIFIABLE

Unification (2)

- Unification takes place **implicitly** when a rule is used for a goal:

```
knows(john, U, R).

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

```
X = john
Y = U
R = living
```

- The rule is usable iff the goal and the rule consequent variables unify.

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).

The *is* operator in Prolog

- Unlike a functional language, expressions that look like arithmetic are **not** automatically evaluated.
- Example: X - 1 is kept as its **syntax tree**
  -(X,1).
- The *is* operator is used to force evaluation.
- Example: Y is X-1 means:
  unify Y with the arithmetic **value** of X-1
- All variables in the right-hand side must be bound ***before*** evaluation.

The *is* operator in Prolog

Y = 5, X = Y + 3.	% X = 5 + 3 (without evaluation)
X = Y + 3.	% X = Y + 3
Y = 5, X is Y + 3.	% X = 8
X is Y + 3.	% is an error; Y is unbound

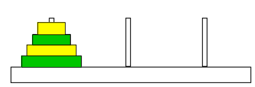
Other arithmetic operators that force evaluation (of both sides)

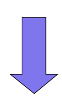
<	e.g. 2+3 < 5+1
>	
=<	(not <=, which "looks like an arrow")
>=	(not =>, which "looks like an arrow")
=\=	e.g. 3+2 =\= 5+1
=:=	e.g. 3+2 =:= 4+1

Other comparison operators

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

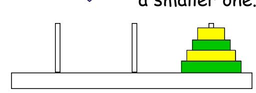
Example: Towers of Hanoi





Move only one disk at a time.

Never place a larger disk on a smaller one.



Solving Towers of Hanoi

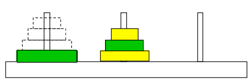
- Some approaches:
  - Pre-programmed solution
    - Recursive solution is easy in most languages
  - Let prolog find solution using depth-first search
    - Trickier, but shows off Prolog's capabilities
    - May not find shortest solution
  - Program breadth-first search in Prolog
    - Still trickier

Pre-Programmed Towers of Hanoi (1)

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

Pre-Programmed Towers of Hanoi (2)

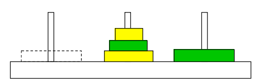
- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack Other (the stack other than ***From*** and ***To***)



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.

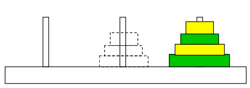
Pre-Programmed Towers of Hanoi (3)

- 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***



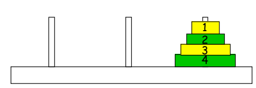
Pre-Programmed Towers of Hanoi (4)

- 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 list
% 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 appending).

% 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).
```

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).

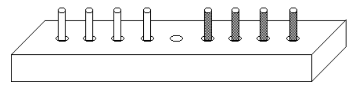
```
towers(InitialState, Moves) :- towers(InitialState, [ ], Moves).


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

towers(Before, Seen, [Move | Moves]) :-
    nonMember(Before, Seen),        only consider if Before not already seen
    move(Move, Before, After),
    towers(After, [Before | Seen], Moves).    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.)

Bi-Directional Execution (1)

Consider:

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

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

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

Bi-Directional Execution (2)

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
```

Generating with **append**

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

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

functional

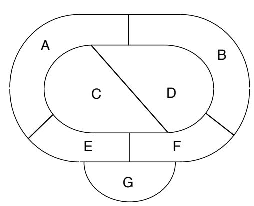
```
| ?- append ([1, 2, 3], [4, 5], Z).

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

no
```

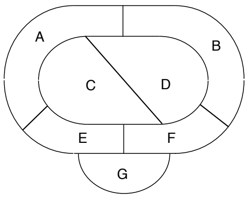
Generator/Test Example: Map Coloring

A map

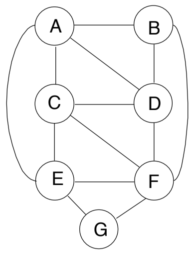


Map Coloring (2)

A map



Corresponding graph

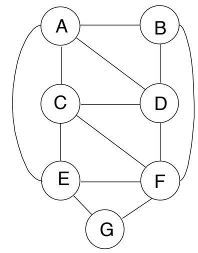


Map Coloring (3)

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.
```

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

colors to be used

means individuals are **not equal**

These and the preceding clause are the entire program.

Constraint Solving with Generate & Test: "Zebra" problem (1 of 2)

There are five consecutive houses, each of a different color and inhabited by men of different nationalities. They each own a different pet, have a different favorite drink and drive a different car.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Porsche driver owns snails.
7. The Masserati is driven by the man who lives in the yellow house.

"Zebra" problem (2 of 2)

8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who drives a Saab lives in the house next to the man with the fox.
11. The Masserati is driven by the man in the house next to the house where the horse is kept.
12. The Honda driver drinks orange juice.
13. The Japanese drives a Jaguar.
14. The Norwegian lives next to the blue house.

The problem is: Who owns the Zebra? Who drinks water?

A Prolog Solution to "Zebra"

```
left_right(L,R,[L,R,_,_,_]).
left_right(L,R,[_,L,R,_,_]).
left_right(L,R,[_,_,L,R,_]).
left_right(L,R,[_,_,_,L,R]).

next_to(X,Y,L) :- left_right(X,Y,L).
next_to(X,Y,L) :- left_right(Y,X,L).
```

```
zebra(S) :-
   S = [[_,norwegian,_,_,_],_,[_,_,_,milk,_],_,_],
   next_to([_,norwegian,_,_,_],[blue,_,_,_,_],S),
   member([green,_,_,coffee,_],S),
   left_right([ivory,_,_,_,_],[green,_,_,_,_],S),
   member([red,englishman,_,_,_],S),
   member([_,ukranian,_,tea,_],S),
   member([yellow,_,masserati,_,_],S),
   member([_,_,honda,orange_juice,_],S),
   member([_,japanese,jaguar,_,_],S),
   member([_,spaniard,_,_,dog],S),
   next_to([_,_,masserati,_,_],[_,_,_,_,horse],S),
   member([_,_,porsche,_,snails],S),
   next_to([_,_,saab,_,_],[_,_,_,_,fox],S).
```

Builtin Arithmetic Not Reversible

- Consider a clause
  p(X, Y) :- Y is X+1.
- A possible use of this clause is:
  | ?- p(5, Z).
  Z = 6
- This clause cannot be used in reverse because the is predicate is not reversible:
  | ?- p(X, 6).
  will not give X = 5; it will fail.

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
```

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 generating or checking a solution.
- We can use it to generate all solutions.

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).

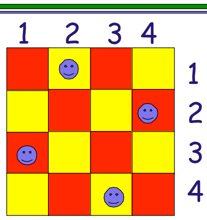
slowsort (joke)

```
% 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

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).
```

Prolog's Origins

(see http://max.cs.kzoo.edu/~acarra/prolog.html)

- 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.

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.)