Introduction to
Logic Programming
and the
Prolog Language

Robert Keller
November 2012

What is this?

- Logic programming is a computation model and the basis for certain *declarative* programming languages.
- It accommodates a different mind-set from conventional languages.
- Uses:
  - Artificial intelligence applications
  - Language understanding and translation
  - Knowledge representation
  - Databases

Declarative Language

- A declarative language allows more focus on the "what" of computing and less on the "how".
- Functional languages form one example.
- Logic languages **generalize** functional languages.

Why study this?

- Expands **expressiveness** beyond what we have seen so far.
- A good language for learning about computational paradigms based on:
  - 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.
- People who develop systems based on knowledge and reasoning.
- People who are not afraid of the unconventional.

Advantages of Logic-Programming Succinctness

- Code moves closer to concepts (and farther from machine details).
- Multiple purposes served by a single piece of code.
- Easier to ascertain **correctness**.
- Easier to evolve software.

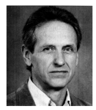
Origins

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

Prolog evolved out of research on **language translation** at the University of Aix-Marseille back in the late 1960's and early 70's.

Alain Colmerauer

Alain Colmerauer and Phillipe Roussel, both of University of Aix-Marseille, collaborated with Robert Kowalski of the University of Edinburgh to create the an early version of Prolog as we know it today.



Robert Kowalski

Kowalski contributed the theoretical framework on which Prolog is founded, while Colmerauer's research at that time provided means to formalize the Prolog language.

Standardized Prolog

The **ISO Prolog standard**: ISO/IEC 13211-1 published in 1995, aims to standardize the existing practices of the many implementations of the core elements of Prolog. It has clarified aspects of the language that were previously ambiguous and leads to portable programs. The standard is maintained by the ISO X3J17 committee.

ISO = "International Standards Organization"

http://www.complang.tuwien.ac.at/ulrich/iso-prolog/

Logic vs. Functions

- Logic programming and functional programming are two types of declarative programming.
- In functional programming, a high-level solution **function** is realized by composing simpler functions.
- In logic programming, a **predicate** expressing a **goal** that the solution is expected to fulfill is decomposed into simpler predicates, using **logic** as the means of composition.

Predicate vs. Function

- A function returns a result given some arguments.
- A predicate is true or false given some arguments.
- So how can a predicate be more general?

How can a predicate be more general?

- If we have to supply all arguments to a predicate, then maybe not so general.
- But in a logic language based on predicates, we can **leave certain arguments as variables**, and expect them to be filled in as results.

Example: Predicate vs. Function

- Function: add(3, 5) returns 8
- Predicate: add(3, 5, Z) succeeds (can be true), binds Z to 8.

Reversibility

- add(3, 5, Z) succeeds (can be true), binds Z to 8.
- add(3, Y, 8) succeeds, binds Y to 5.
- add(X, 5, 8) suceeds, binds X to 3.
- What would add(X, Y, 8) do?

What would add(X, Y, 8) do?

- Result is "non-deterministic": There are multiple ways of succeeding.
- Through **backtracking**, one binding is returned. If that doesn't suffice, another binding is tried, and so on.

Preceding Discussion Slightly Misleading

- Prolog is not usually used to solve arithmetic problems in the manner indicated.
- However, it is very good at solving **list** problems in a similar fashion.

Predicate vs. Function Preview

- Function append:
  (append '(a b c) '(d e)) ⇒ '(a b c d e)
- Predicate append:
  append([a, b, c], [d, e], Z).
  ⇒ Z = [a, b, c, d, e]

But Predicates Offer More Options

- Predicate append:
  - append([a, b, c], [d, e], Z).
    ⇒ Z = [a, b, c, d, e]
  - append([a, b, c], Y, [a, b, c, d, e]).
    ⇒ Y = [d, e]
  - append(X, Y, [a, b, c, d, e]).
    ⇒ X = [], Y = [a, b, c, d, e]
    ⇒ X = [a], Y = [b, c, d, e]
    ⇒ X = [a, b], Y = [c, d, e]
- Tying these ideas together into a **coherent logical framework** is a major contribution of logic programming.

Logic from the beginning …
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}.
  - The domains are, e.g. strings, numbers, lists, and various other structures.

Comparison

- Propositions:
  hmc_is_in_claremont
  caltech_is_in_glendale
- Predicates (X and Y are variables)
  domain = {caltech, glendale, hmc, …}
  - is_in(X, Y)
  - is_in(hmc, claremont)
  - is_in(caltech, glendale)

Logic of Implication (Prolog Style)

- :- is like → (implies) reversed
- **Logical rule:**
  Q :- P1, P2, P3.
  means that proposition, or predicate application,

  Q **is *implied by*** the **conjunction** (and) of P1, P2, and P3:

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

  If one of P1, P2, P3 is false, nothing is claimed.

Prolog Lingo: Clauses

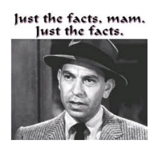
- This is a **clause**:

  q :- p1, p2, p3.

  - q is called the **head**.
  - p1, p2, p3 comprise the **body**.
  - Each of q, p1, p2, p3 are individually called **goals**.
- Does this remind you of anything?

Prolog Program

- A Prolog program will generally consist of multiple clauses.
  - q :- p, r.
  - p :- s, t.
  - s :- r.
- Some clauses might not have bodies:
  - r.
  - t.
- Not having a body is like an axiom:
  a statement that does not require proof.
  They are called **facts**.



A Prolog Program?

Example

- Things to do to prepare for an exam.
  - Attend the lectures.
  - Read the book.
  - Do the problems.
  - Be tutored by someone.

  - or, maybe you already *Know it all*.

Goal-Oriented Viewpoint

- Goal:
  prepared_for_exam.
- There can potential be multiple ways to achieving a given goal.

Example Clause

- prepared_for_exam :-
  read_book, % comma is *and*
  worked_problems, % percent is comment
  attended_lectures.

Example Clause

- prepared_for_exam :-
  tutored_by_someone_prepared.

Example Clause

- prepared_for_exam :-
  knows_it_all.

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 (provable) 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, **or**
  - There is a **clause**,
    - the **head** of which matches the goal, and
    - all goals in the body **succeed**.

recursion

- [Notice the first blue bullet is really a special case of the second, because facts have no body.]
- If a goal cannot succeed, then it **fails**.

Success and Failure Examples

- If the facts are:

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

and there is just one clause:

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

then the goal

?- prepared_for_exam.

succeeds. If any of those 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.

Two Compatible Interpretations of Prolog Execution

- **Logical** interpretation:
  - Implications and facts, logical proof.

    prepared :- read, worked, attended.

    "If you've read, worked, and attended, then you're prepared."
- **Procedural** interpretation:
  - Goals, backtracking, etc.

    "To prepare, (you can) read, work, attend."

How Prolog Works: Depth-First Search

- 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.

Goal Stack Example

- Suppose the program is:
  - p :- s, t.
  - t :- r.
  - s.
  - t.
- And the goal is
  - ?- p.

Goal Stack Example

- Goal Stack Trace (top at left)
  - p
  - s t
  - t
  - r
  - failure (no match for top of stack)

```
p :- s, t.
t :- r.
s.
t.
```

Goal Stack Example

- Suppose the program is:
  - p :- r, s, t.
  - r :- s.
  - s :- t.
  - t.
- And the goal is
  - ?- p.

Goal Stack Trace

- p
- r s t
- s s t
- t s t
- s t
- t t
- t
- success (empty stack)

```
p :- r, s, t.
r :- s.
s :- t.
t.
```

Goal Stack Quiz

- Suppose the program is:
  - p :- r, t, u.
  - r :- s.
  - s :- t.
  - u :- p.
  - t.
- And the goal is
  - ?- p.
- Does the goal succeed or fail?

Multiple Solution Possibilities

- The previous example had at most one clause with a given head.
- If there is more than one clause with a given head, and the first fails, the next can be tried, etc. until there are no more possibilities.
- However, we have to **remember where we took the previous branch**.

Goal Stack Example with Branching

- Suppose the program is:
  - p :- r, t, u.
  - r :- s.
  - s :- t.
  - u :- v.
  - u :- s.
  - t.
- And the goal is
  - ?- p.

Goal Stack Trace, with Branching

p :- r, t, u.
r :- s.
s :- t.
u :- v.
u :- s.
t.

- p
- r t u
- s t u
- t t u
- t u
- u
- v
- (failure, must branch back) Try u :- s.
- s
- t
- (success)

Backtracking

- When a failure occurs, Prolog does not necessarily stop.
  It goes **back** to the previous clause it used, and restores the goal stack 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, top-to-bottom, until there are no more clauses left.
- For new each goal on the stack,
  searching starts afresh from the top of the program.

Stack Complications

- The **goal stack** is analogous to the stack used in recursive programming.
- In addition to keeping remaining goals, it is also necessary to keep track of where choices were made, so called **choice-points**.
- When failure occurs, the goal-stack is popped back to the last place there was a choice, with the top of stack at that time effectively restored, and the next choice option taken.
- Additional structure is needed to keep track of these choice points.

An Execution Model

- We give a functional model in Racket, that handles choices.
- It is for expository purposes only, not efficiency.

```
;; Simple Prolog execution model

(define (solve goal-stack all-rules)
  (solve1 goal-stack all-rules all-rules))

(define (solve1 goal-stack remaining-rules all-rules)
  (if (null? goal-stack)
      success
      (let (
            (found (match (first goal-stack) remaining-rules))
            )
        (if found
            (if (solve (append found (rest goal-stack)) all-rules)
                success
                (solve1 goal-stack (rest remaining-rules) all-rules))
            fail))))

(define (match goal rules)
  (if (null? rules)
      #f
      (if (equal? goal (head (first rules)))
          (body (first rules))
          (match goal (rest rules)))))

(define (head rule) (first rule))
(define (body rule) (rest rule))
```

Prolog's Version of Negation

- Facts can ***only*** be asserted in the positive sense.
- Negation can be tested, but not asserted.
- Negation is "negation as failure":
  \+*Goal*
  succeeds iff *Goal* fails.
- Etymology: In classical logic, |— *G* means "*G* is provable".
  \+ is a "smiley" version of "is not provable".

A Negation Example

- The clause

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

will enable

```
prepared_for_exam
```

to succeed by this clause 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

Let's say the class contains {bob, fred, judy, sam}, and the facts are:

- 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 Racket/Scheme
  - Data items can also start with upper-case **if singly-quoted**, e. g. 'John Hancock'.
  - Unlike Racket/Scheme, dash '-' is **not** considered to be just another letter. (It is an infix "functor".)

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 ← Y Z ← f(Y) - OR - Y ← X Z ← f(X)
p(a, f(Y))	p(Y, Z)	Yes	Y ← a Z ← f(a)
p(g(Z), f(Y))	p(Y, Z)	Yes (but only in Prolog)	Y ← g(f(g(f(.. Z ← f(g(f(g..

Quiz: 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

```
% This is with SWI-Prolog 10.5.2

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

?- p(X, X) = p(fred, bob).
false.

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

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

```
% Here a graph is represented by a list of pairs of nodes.

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) prevent the Prolog compiler from complaining about "singleton variables" in clauses (which usually signals a user error).

```
Warning: … filename … line containing first clause …
        Singleton variables: [X1,X2]
```

```
p(X1) :- q(X2).    % Did the user intend X1 and X2 to be the same?

p(_X1) :- q(_X2). % If not, do it this way
```

Throw-Away Variable

_ by itself is extremely "wild":

It does not even need to unify with other instances of the same variable, as do other variables.

Throw-Away Examples

```
?- _X = 1, _X = 2.   % Each _X is the same var

false.

?- _ = 1, _ = 2.     % Each _ is a separate var
true.
```

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 ⊆ names **x** dorms (as a set of pairs)

takes ⊆ names **x** depts **x** numbers (as a set of 3-tuples)

tutors ⊆ 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

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 backtracking, but never before.
- After backtracking, they may be **re-bound**.
- Amazingly, 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).
```

```
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).
```

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).
```

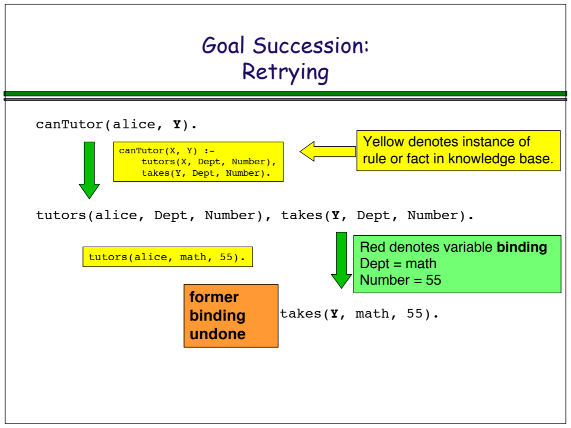
```
tutors(alice, math, 55).
```

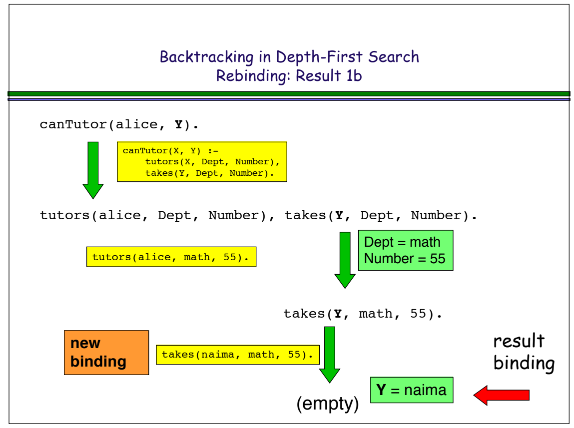
Red denotes variable **binding**
Dept = math
Number = 55

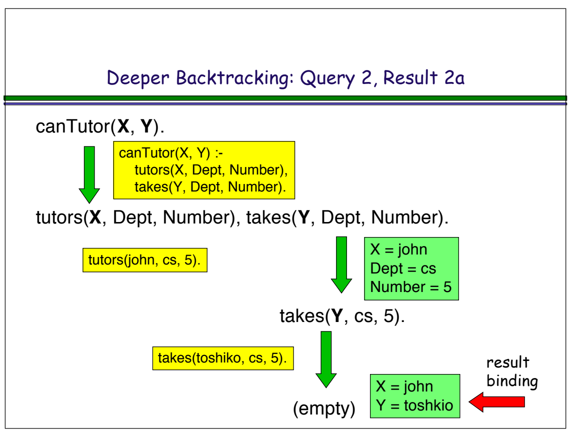
undo former binding; try for another result

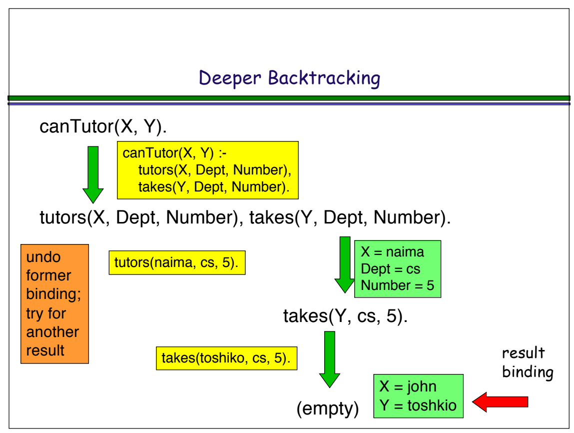
```
takes(Y, math, 55).
```

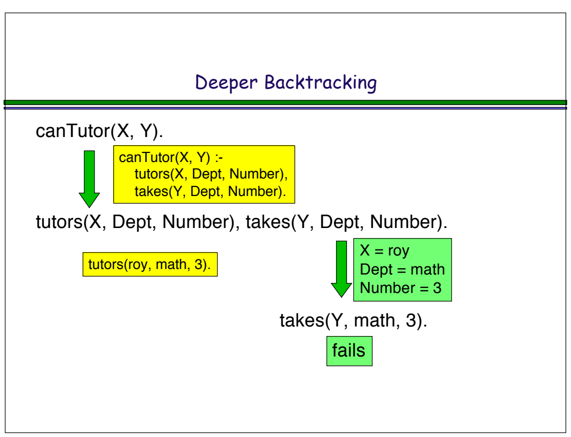
```
takes(roy, math, 55).
```

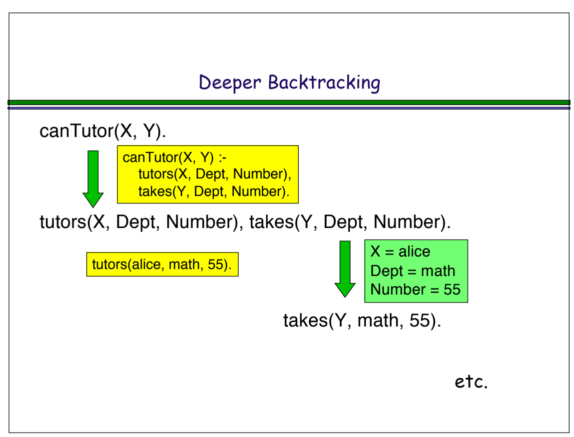

Goal Succession:
Retrying
canTutor(alice, Y).
canTutor(X, Y) :-
tutors(X, Dept, Number),
takes(Y, Dept, Number).
Yellow denotes instance of rule or fact in knowledge base.
tutors(alice, Dept, Number), takes(Y, Dept, Number).
tutors(alice, math, 55).
Red denotes variable binding
Dept = math
Number = 55
former binding undone
takes(Y, math, 55).


Backtracking in Depth-First Search
Rebinding: Result 1b
canTutor(alice, Y).
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).
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
X = john
Y = toshkio
(empty)


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
X = john
Y = toshkio
(empty)


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, binding Z.

This sub-goal is satisfied next.

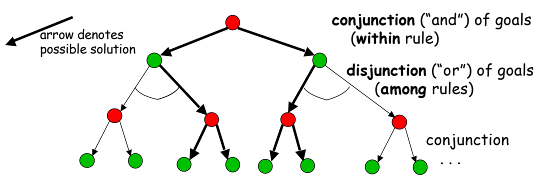
This rule is tried after the first rule is **exhausted**.

```
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

- Ideally, variables are understood as pure logic. However, a deeper operational understanding is sometimes necessary.
- 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 first **unbinding the previous binding**.

Creation of Logical Variables

- Logical variables may get created dynamically and implicitly whenever goals are introduced.
- A top-level goal may contain logical variables.
- The use of a clause, the head of which matches a goal, may create new goals containing logical variables.

Logical Variable Binding is Transitive

- **One logical variable can be bound to another**, which means that they are effectively the same, **as long as this binding is in effect**.
- Arbitrarily-long chains of bindings can exist.

Example

- X = b, Y = X.

  X is first bound to b,
  then Y is bound to X.

  Both X and Y effectively have the same value, b.

Example

- X = Y, Y = b.

  X is first bound to Y,
  then Y is bound to b.

  Both X and Y effectively have the same value, b.

Example

- Suppose we have some clauses for predicates p and q:
  - p(b).
  - p(X) :- q(X, Y), p(Y).
  - q(c, b).
- What solutions will be produced for the top-level goal:
  ?- p(Z).

Example continued

```
p(b).
p(X) :- q(X, Y), p(Y).
q(c, b).
```

?- p(Z).

Z is a logical variable

Z = b will be the first solution.

On backtracking, Z will be re-bound to X in the second clause. A new logical variable for Y is introduced.

Z and Y will then have values c, b.

Z = c will be the second solution.

Lists in Prolog

- Unlike Racket/Scheme lists, Prolog lists use square brackets and require comma separators:
  - [a, b, 123]
  - [[foo, bar], []]
- There is no need for explicit *cons*, *first*, *rest*. Rather unification is used for this purpose:
  - [First | Rest] = [1, 2, 3, 4]
    unifies with First ← 1, Rest ← [2, 3, 4]
  - [First | Rest] does not unify with []

Lists in Prolog

- second, third, etc. **functions** are also not necessary:

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

  unifies with:
  First ← 1, Second ← 2, Third ← 2,
  More ← [4, 5]

Logical Variables in Lists and Other Structures

- Because they are "objects", variables can occur in lists in either bound or unbound states.
- Suppose L = [a, X, b].
  If X is unbound, it acts as a "place-holder", which can subsequently be bound by unification, e.g.
  L = [_, c, _] will unify X with C.

Movie Database

- % 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 by <.
- 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.

Functional Programming in Prolog

- Most concepts you know from Racket/Scheme are applicable.
- Syntax is different.
- Also, "higher order" functions are not built-in. We must code them.

Example: range function

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

```
?- range(1, 10, L).
   L = [1,2,3,4,5,6,7,8,9,10]
```

Note: This range is not "reversible", due to the use of is.

Example: max computes the maximum of a non-empty list of numbers

```
max([X | L], M) :- max_helper(L, X, M).

max_helper([], M, M).

max_helper([X | L], M, N) :-
   X > M,
   max_helper(L, X, N).

max_helper([X | L], M, N) :-
   X =< M,
   max_helper(L, M, N).

test :- max([3, 7, 2, 9, 1, -5], M), write(M), nl.
```

Note: Tail recursion applies here.

Note: This max is not "reversible", due to the use of >.

Using McCarthy's Transformation

- In some cases, either think, or write out, the program as imperative, then convert to logic.

Example: extended max computes the maximum of a list of numbers and the **first location** of that maximum

```
max([X | L], M) :- max_helper(L, X, M).

max_helper([], M, M).

max_helper([X | L], M, N) :-
   X > M,
   max_helper(L, X, N).

max_helper([X | L], M, N) :-
   X =< M,
   max_helper(L, M, N).

test :- max([3, 7, 2, 9, 1, -5], M), write(M), nl.
```

Note: Tail recursion applies here.

Note: This max is not "reversible", due to the use of >.

Example: extended max computes the maximum of a list of numbers and the **first location** of that maximum

Imperative version:

```
L is the original list
M will be the maximum value
I will be the first location of M

L = … non-empty list …
N = first(L);
K = 0;  // location of N
L = rest(L);
J = 1;  // location of first element of L
while( L is non-empty )
        {
        if( first(L) > N )
           { N = first(L); K = J;}
        J = J+1;
        L = rest(L);
        }
M = N;
I = K;
```

Example: extended max computes the maximum of a list of numbers and the **first location** of that maximum

Imperative Version:

```
L is the original list
M will be the maximum value
I will be the first location of M

L = … non-empty list …
N = first(L);
K = 0;  // location of N
L = rest(L);
J = 1;  // location of first element of L
while( L is non-empty )
        {
        if( first(L) > N )
           { N = first(L); K = J;}
        J = J+1;
        L = rest(L);
        }
M = N;
I = K;
```

Logic Version:

```
max([X | L], M, I) :- max_helper(L, 1, X, 0, M, I).

max_helper([], _, N, I, N, I).

max_helper([X | L], J, N, _K, M, I) :-
   X > N,
   J1 is J+1,
   max_helper(L, J1, X, J, M, I).

max_helper([X | L], J, N, K, M, I) :-
   X =< N,
   J1 is J+1,
   max_helper(L, J1, N, K, M, I).
```

Using -> … ; … (if … then … else …) for greater clarity and efficiency

Original Version:

```
max([X | L], M, I) :-
            max_helper(L, 1, X, 0, M, I).

max_helper([], _, N, I, N, I).

max_helper([X | L], J, N, _K, M, I) :-
   X > N,
   J1 is J+1,
   max_helper(L, J1, X, J, M, I).

max_helper([X | L], J, N, K, M, I) :-
   X =< N,
   J1 is J+1,
   max_helper(L, J1, N, K, M, I).
```

Revised Version:

```
max([X | L], M, I) :- max_helper(L, 1, X, 0, M, I).

max_helper([], _, N, I, N, I).

max_helper([X | L], J, N, K, M, I) :-
   J1 is J+1,
   (        % Note outer (…) needed for correctness.
   X > N ->
      max_helper(L, J1, X, J, M, I)
   ;
      max_helper(L, J1, N, K, M, I)
   ).
```

Note: Tail recursive.

call: Predicates as Arguments

- **call**(P, A1, A2, …, An) allows predicate P to be a variable.
- Its meaning is ***as if***

  P(A1, A2, …, An)

- The latter syntax is not allowed, however.

call example

```
p(0, 1).
p(1, 2).
p(2, 0).

test(X) :- call(p, X, Y), write(Y), nl.
```

```
?- test(0).
1
true.

?- test(1).
2
true.

?- test(2).
0
true
```

map example: map applied to 2-ary predicates

```
map(_P, [], []).

map(P, [A | X], [B | Y]) :-
     call(P, A, B),
     map(P, X, Y).

% Example use
p(X, Y) :- Y is X+1.

test :- map(p, [1, 2, 3], Z), write(Z), nl.
```

More on Using Logical Variables

- length(X, N) is true when X is a list of length N.
- length([a, b, c], 3) succeeds
- length([a, b], 3) fails
- length(X, 3) succeeds with X being a list of three generated logical variables

```
?- length(X, 3).
X = [_G263, _G266, _G269].
```

More on Using Logical Variables

- member(A, L) succeeds when A is a member of list L.
- member(c, [a, b, c]) succeeds.
- member(c, [a, b, d]) fails.
- member(c, [a, b, X]) succeds with X = c.
- member(c, X) succeeds an infinite number of ways.

```
?- member(c, X).
X = [c |_G328] ;
X = [_G327, c |_G331] ;
X = [_G327, _G330, c |_G334] ;
X = [_G327, _G330, _G333, c |_G337] ;
X = [_G327, _G330, _G333, _G336, c |_G340]
```

Generate-Test Programming

- One way to solve a constraint-based problem:
  - Generate a trial solution
  - Test to see if it really is a solution
  - Repeat the above until a solution is found.
- A challenge is to get the generation to cover **all** possibilities.

Generate-Test Programming

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
```

A simple constraint problem

- I want a list L that:
  - Contains exactly three elements
  - All elements are members of a list M
  - One element occurs exactly twice in the list
- Simple solution:

```
solve([X, X, Y], M) :- ok(X, Y, M).
solve([X, Y, X], M) :- ok(X, Y, M).
solve([Y, X, X], M) :- ok(X, Y, M).
ok(X, Y, M) :- member(X, M), member(Y, M), X \= Y.
```

Example Execution

```
?- solve(L, [1, 2, 3]).
L = [1, 1, 2] ;
L = [1, 1, 3] ;
L = [2, 2, 1] ;
L = [2, 2, 3] ;
L = [3, 3, 1] ;
L = [3, 3, 2] ;
L = [1, 2, 1] ;
L = [1, 3, 1] ;
L = [2, 1, 2] ;
L = [2, 3, 2] ;
L = [3, 1, 3] ;
L = [3, 2, 3] ;
L = [2, 1, 1] ;
L = [3, 1, 1] ;
L = [1, 2, 2] ;
L = [3, 2, 2] ;
L = [1, 3, 3] ;
L = [2, 3, 3] ;
false.
```

What Happens

?- solve(L, [1, 2, 3]).
Clause solve([X, X, Y], M) :- ok(X, Y, M).
binds L to [X, X, Y], creating goal

ok(X, Y, [1, 2, 3])
which is solved to get X = 1, Y = 2
which gives L = [1, 1, 2]

backtracking gives X = 1, Y = 3
which gives L = [1, 1, 3]

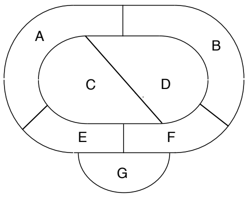
backtracking gives X = 2, Y = 1
which gives L = [2, 2, 1]

backtracking gives X = 2, Y = 3
which gives L = [2, 2, 3]

etc. eventually other clauses for solve are used.

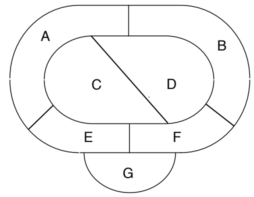
Generate/Test Example: Map Coloring

A map

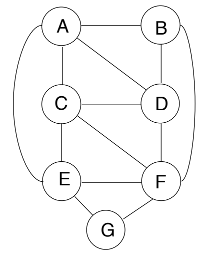


Map Coloring (2)

A map



Corresponding graph

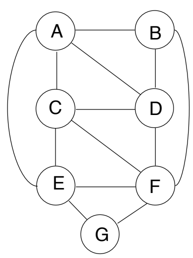


Map Coloring (3)

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.

Version where the colors are presented as a list

- next(X, Y, Colors) :-
  member(X, Colors, Residue),
  member(Y, Residue).
- The generalized member ensures that the second color is not a duplicate (assuming no duplicates in the original list).

Version where the colors are presented as a list

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

Sudoku

- Sudoku is basically a graph-coloring problem, except that we have a "hypergraph" rather than an undirected graph.
- Adjacency is **no longer binary**. Any 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

```
testSudograph(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).
```

```
?- testSudograph(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]
```

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)

```
testZebra :-
   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).
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.

Optimizing Generate-Test

- It is more to generate fewer possibilities.

  ok(X, Y, M) :- member(X, M), member(Y, M), X \= Y.

  generates pairs where X = Y, then rejects them.

- The following revision avoids generating pairs that are going to be rejected anyway.

  ok(X, Y, M) :- member(X, M, R), member(Y, R) , X \= Y

- Above, R is the "residue" of removing X from M.
- If M is known to contain no duplicates, the final check is not necessary.

Exercise: Extended member Predicate

- 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 ;
.
.
.
```

Non-deterministic Programming

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

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

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([]).

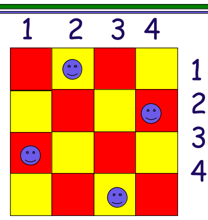
sorted([_]).

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

N-Queens Problem:
ND Programming with Generate-Test Optimization

- 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



Solution representation:
[[1, 3], [2, 1], [3, 4], [4, 2]]

Strategy

- An unoptimized generate-test would generate all possible boards, then test whether a board satisfies the constraints.
- Optimizations:
  - **Generate**: Generate only boards with a queen in each column.
  - **Test**: Add one column at time, reducing the test part to testing conflicts introduced by the newly-added column.

Solving Queens

Given a next column and a list of unoccupied rows:

- If rows is empty, succeed (all rows used).
- For the next unassigned column:
  If there is a row where the queen is not being attacked, place it and recurse.

  If no such row, fail (and backtrack).

Queens Top-Level

```
queens(N, Solution) :-
   range(1, N, Rows),
   queens(Rows, 1, Solution).

range(M, N, []) :- M > N.

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

Queens Recursion

```
queens([], _, []).                % basis

queens(Rows, Col, [[Col, Row] | Rest]) :-
   NextCol is 1+Col,
   member(Row, Rows, RemainingRows),
   queens(RemainingRows, NextCol, Rest),
   nonAttacking(Rest, Col, Row).
```

Queens Non-Attacking

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

nonAttacking([[Row1, Col1] | Pairs], Row, Col) :-
   Row1 - Row =\= Col1 - Col,
   Row - Row1 =\= Col1 - Col,
   nonAttacking(Pairs, Row, Col).

% This checks diagonals.
% Why don't we need to check row and column attacks?
```

The "42" 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 42: 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.

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

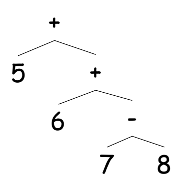
- Remove an element E from the list (using an extended 'member' predicate).
- Recursively split the remaining elements into two.
- Add E back to the first list.
- Exploit symmetry by using an auxiliary predicate and calling it twice from the interface predicate:
  - split(X, L, R) :- split2(X, L, R).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

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).

Evaluating a Tree

- Assume we are representing trees as lists:
  - A number is a tree
  - If T1 and T2 are trees, then [Op, T1, T2] is a tree.
- Example:
  - [+, 5, [+, 6, [-, 7, 8]]]



A Tree Evaluator

- eval(Tree, Value).
- eval(N, N) :- number(N).
- eval([Op, T1, T2], N) :- …

Expressions as Lists Not Strictly Necessary

- An alternate model (not used fall 2010) is to use the expressions themselves as trees.
- Expressions are not inherently evaluated in Prolog, so their parts can be recovered.

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]
```

Example for solve42 alternate

```
?- setof(Exp, solve42a([+, *, -], [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),
                      .
                      .
                      .
```

More on Predicate Logic:

Quantifiers

Quantifiers

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

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

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.

== 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

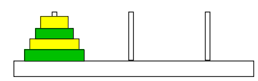
```
sum([ ], Y, Y).

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

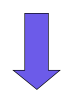
The following doesn't quite work for all inverses. A problem arises in factoring 0.

```
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
```

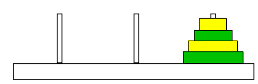
Example: Towers of Hanoi





Move only one disk at a time.

Never place a larger disk on a smaller one.



Solving Towers of Hanoi

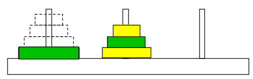
- Some approaches:
  - Pre-programmed solution
    - Recursive solution is easy in most languages
  - Let prolog find solution using depth-first search
    - Trickier, but shows off Prolog's capabilities
    - May not find shortest solution
  - Program breadth-first search in Prolog
    - Still trickier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

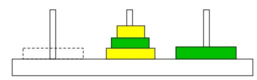
- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack Other (the stack other than ***From*** and ***To***)



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.

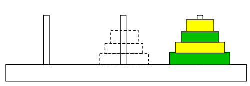
Pre-Programmed Towers of Hanoi (3)

- 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***



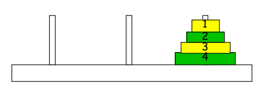
Pre-Programmed Towers of Hanoi (4)

- 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
% 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 appe

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

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

empty target stack

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

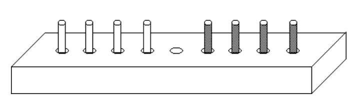
recurse

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.)