CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Prolog

The Prolog Programming Language

- Spectrum of Languages:
- Imperative (“how should the computer do it?”)
  - Von Neumann: Fortran, Basic, Pascal, C
    - Computing via “side-effects” (modification of variables)
  - Object-oriented: Smalltalk, Eiffel, C++, Java
    - Interactions between objects, each having an *internal state* and *functions* which manage that state
- Declarative (“what should the computer do?”)
  - Functional: Lisp, Scheme, ML, Haskell
    - Program ↔ application of functions from inputs to outputs
    - Inspired from *lambda-calculus* (Alonzo Church)
  - Logic, constraint-based: Prolog
    - Specify constraints / relationships, find values that satisfy them
    - Based on *propositional logic*

The Prolog Programming Language

- Developed in the early 1970s by:
  - Alain Colmerauer and Philippe Roussel – University of Aix-Marseille
  - Robert Cowalski – University of Edinburgh
- Prolog – Programming in Logic
- Considered the most significant logic programming language
- Various dialects
- Partially standardized in 1995
- Applications
  - Mathematical logic
  - Natural language processing
  - Symbolic equation solving
  - Many areas of artificial intelligence

The Prolog Programming Language

- General approach in logic programming:
  - Express the problem as a collection of relationships (constraints) between objects
  - The implementation will find the values that satisfy all constraints

- Problem: how many elements are in a list?
  - Imperative programming:
    - traverse the list from first element until last; at each element increment the number of elements N
  - Functional programming:
    - the number of elements N is 0 for an empty list; otherwise, it is 1 plus the number of elements in the list tail (without the first element)
  - Logic programming:
    - the proposition nr_elem (lst, N) is true if
      - list lst is empty and N is 0, or
      - list lst has a head h and a tail t, and nr_elem (t, N-1) is true

The Structure of a Prolog Program

- Programming in Prolog consists of:
  - specifying some facts about objects and their relationships
  - defining some rules about objects and their relationships
  - asking questions (queries) about objects and their relationships
- Prolog program – database (knowledge base) of clauses:
  - facts
  - rules

```
cold_outside.          % this is a fact
```

- Run a Prolog program (interpretor-based) – ask a query:

```
?- cold_outside.       % this is a query
yes                    % system answer
?-                     % wait for next query
```

The Structure of a Prolog Program

- A Prolog clause is composed of terms
- A term may be:
  - a constant
    - a number: 52 3.14
    - an atom (must start with lowercase):
      foo cold_outside 'Hi!' +
  - a variable (must start with uppercase):
Foo X
  - a structure:
    car(ford, explorer, 2003)
    % has a functor (car)
    % and components (ford, explorer, 2003)

Facts

- Express the fact that "John likes Mary"
  - have two objects (John and Mary) and a relationship (likes)

```
likes(john, mary).
```

- Such a relationship is called a predicate
  - similar to a function that returns true or false
  - number of arguments – arity of the predicate
  - specifying the fact above means that likes(john, mary) is true

- Syntactically, a predicate is defined as a structure (functor and arguments/components)

Using Prolog

- SWI-Prolog
  - free downloads for Windows, Linux, Mac
- The essentials:
  - Write your database of facts and rules into a file
  - consult('Tests/Prolog/my_file.pl').

    % to load the database in Prolog (you can also select "Consult" from the menu in Windows)
  - Run queries from Prolog prompt
  - ^C h % help
  - ^C a % abort a running query
  - ^C e % exit

- Documentation available on SWI-Prolog webpage

Queries

- Consider the database:

```
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
```

- Some queries:

```
?- likes(john, mary).
yes
?- likes(mary, john).
no
?- likes(john, money).
no
```

  - Prolog searches for a fact that unifies with the query (tries to satisfy the query goal)
  - The yes/no answers mean that the query predicate can/cannot be proven based on the database
  - If the query predicate is not in the database:

```
?- king(john, france).
no     % in most implementations
ERROR: Undefined procedure: king% in other implementations
```

Variables

- Consider the database:

```
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
```

- Ask what John likes:

```
?- likes(john, X).          % variable X is initially uninstantiated

X = flowers                 % likes(john, X) unifies with likes(john, flowers)
                            % X becomes instantiated to flowers

   % Prolog waits for further instructions:
   % - press ENTER to stop searching for more answers

yes
```

Variables

- Same database:

```
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
```

- If we want all answers:

```
?- likes(john, X).            % variable X is initially uninstantiated
X = flowers ;                 % likes(john, X) unifies with likes(john, flowers)
                                 % X becomes instantiated to flowers
  % a marker it placed in database where the
  % unifier was found → likes(john, flowers)
                    % - press ; to continue searching
                    % X becomes uninstantiated again
 % search is resumed from the place marker
X = mary ;                    % X becomes instantiated to mary
no                               % no more answers
```

Variables

- Same database:

```
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
```

- Ask who likes Mary:

```
?- likes(X, mary).

X = john ;

X = paul ;

no
```

Conjunctions

- Consider the database:

```
likes(mary, chocolate).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- Ask if John and Mary like each other
  - need to ask if John likes Mary and Mary likes John

```
?- likes(john, mary), likes(mary, john).      % comma between two goals represents
   % their conjunction ("and")
no
```

- How does Prolog try to satisfy a goal (or conjunction of goals)?
  - backtracking (backward-chaining)

Backtracking

- Same database:

  likes(mary, chocolate).
  likes(mary, wine).
  likes(john, wine).
  likes(john, mary).

- Ask if there is something that both Mary and John like:

  ?- likes(mary, X), likes(john, X).
  % scope of X is the entire clause

1. Try to satisfy first goal likes(mary, X), where X is uninstantiated
   - Succeeds in unifying with likes(mary, chocolate)
   - X becomes instantiated to chocolate
   - Mark the place in database where first goal succeeded

2. Try to satisfy second goal likes(john, chocolate) – it fails

Backtracking (cont.)

- Same database:

  likes(mary, chocolate).
  likes(mary, wine).
  likes(john, wine).
  likes(john, mary).

- Ask if there is something that both Mary and John like:

  ?- likes(mary, X), likes(john, X).
  % scope of X is the entire clause

3. Try to re-satisfy first goal likes(mary, X), with X again uninstantiated
   - Start from the marker of first goal – succeeds in unifying with likes(mary, wine)
   - X becomes instantiated to wine
   - Mark the place in database where first goal succeeded

4. Try to satisfy second goal likes(john, wine)
   - Succeeds in unifying with likes(john, wine)
   - Mark the place in database where second goal succeeded
   - Now the query is satisfied:

X = wine ; % ask for more answers – try to re-satisfy second goal likes(john, wine) from its marker – it fails

% try to re-satisfy first goal likes(mary, X) from

no its marker – it fails → no more answers

Rules

- Rules – express a predicate that depends on other predicates

- John likes anyone who likes wine:

  likes(john, X) :- likes(X, wine).

- John likes any female who likes wine:

  likes(john, X) :- female(X), likes(X, wine).

- A bird is an animal which has feathers:

  bird(X) :- animal(X), has_feathers(X).

- A rule has a head (bird(X)) and a body (animal(X), has_feathers(X))
- :- is read "if"
- Scope of variables:
  - the scope of a variable is only the clause in which it appears
  - all X should represent the same object in second rule above
  - X in likes(john, X) and X in bird(X) do not have anything in common

Backtracking with Rules

- Consider the database:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward, victoria, albert). % victoria and albert are parents of edward
parents(alice, victoria, albert).
sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).

?- sister_of(alice, edward).
```

1. Query unifies with sister_of(X, Y), so X is alice, and Y is edward
   Now try to satisfy goals in the body, one by one

2. Try to satisfy female(alice) - succeeds
   Place a marker for this goal (at 3rd clause)

Backtracking with Rules (cont.)

- Consider the database:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward, victoria, albert). % victoria and albert are parents of edward
parents(alice, victoria, albert).
sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).

?- sister_of(alice, edward).
```

3. Try to satisfy parents(alice, M, F) – succeeds, M is victoria, and F is albert
   Place a marker for this goal (at 6th clause)
4. Try to satisfy parents(edward, victoria, albert) – succeeds
   Place a marker for this goal (at 5th clause)
   Now the entire query succeeds:

```
yes
```

Rules – Shared Variables

- Same database:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward, victoria, albert).        % victoria and albert are parents of edward
parents(alice, victoria, albert).

sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).        % clause 7

?- sister_of(alice, X).

X = edward
```

- X in the query sister_of(alice, X) is not the same as X in clause 7
- When unifying query with head of clause 7:
  - X in clause 7 will instantiate with alice
  - Y in clause 7 and X in query remain uninstantiated, but they are shared (co-refererences)
    - when one becomes instantiated, the other will be instantiated to the same object

Anonymous Variables

```
male(albert).
male(edward).
male(jimmy).
female(alice).
female(victoria).
parents(edward, victoria, albert).        % victoria and albert are parents of edward
parents(alice, victoria, albert).
parents(jimmy, lisa, albert).
sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).
```

- Is Alice anyone's sister?

```
?- sister_of(alice, _).        % _ is a placeholder for an anonymous variable
   % don't need to know who it is
yes
```

- Do Alice and Jimmy have the same father?

```
?- parents(alice, _, X), parents(jimmy, _, X).
X = albert ;                   % anonymous variables do not co-refer with any other      % variable or anonymous
   variable
no
```

Structures

- Structures can also appear in predicates:

```
owns(john, book(wuthering_heights, author(emily, bronte))).
```

- Can have queries about components of structures – ask if John owns a book by any of the Bronte sisters:

```
?- owns(john, book(X, author(Y, bronte))).

X = wuthering_heights
Y = emily
yes
```

- Or, with anonymous variables:

```
?- owns(john, book(_, author(_, bronte))).
yes
```

- Why do predicates and structures have the same syntax?
  - convenient to represent a Prolog program as a set of structures
  - can add/remove clauses dynamically, at run-time

Unification

- Prolog rules for unification:
- An uninstantiated variable unifies with any object
  - if the object has a value (is a constant or an instantiated variable), the first variable becomes instantiated to that object
  - if the object is an uninstantiated variable, the two variables will remain uninstantiated, but will co-refer
- A constant unifies only with itself or with an uninstantiated variable
- A structure will unify with another structure if they have the same functor and arity (number of arguments), and the corresponding arguments also unify recursively; exactly the same rule also applies to predicates

The Occurs Check

- Consider the following query:

  ?- p(X) = X.

- Should p(X) and X unify?
  - Conceptually, no – for any constant a, it's obvious that p(a) is not the same as a
  - However, checking for such occurrences would be potentially time-consuming
  - Therefore, this occurs check is not performed in Prolog
    - An uninstantiated variable unifies with anything
  - The answer is:

    X = p(p(p(p(p(p(p(p(p(p(...))))))))))

    yes

Programming with Matching

- Example (horizontal and vertical lines):

```
vertical(line(point(X,Y), point(X,Z))).
horizontal(line(point(X,Y), point(Z,Y))).

?- vertical(line(point(1, 1), point(1, 3))).
yes

?- horizontal(line(point(1, 1), point(2, Y))).
Y = 1 ;
no

?- horizontal(line(point(2, 3), P)).
P = point(_G405,3) ;          % any x-coordinate will do
no
```

Equality

- Equality - defined in terms of "unifiability"
- The goal =(X, Y) which can also be written as X = Y, succeeds if and only if X and Y can unify
- Examples:

```
?- a = a.
yes                 % constant unifies with itself

?- a = b.
no% but not with another constant

?- foo(a, b) = foo(a, b).
yes                 % structures are recursively identical

?- X = a.
 X = a ;            % variable unifies with constant, also becomes      instantiated
 no                 % only once
```

Equality

- More examples:

```
?- foo(a, b) = foo(X, b).
 X = a ;                    % arguments must unify
 no                         % only one possibility

?- foo(a, b) = X.
 X = foo(a, b) ;            % variable unifies with anything
 no                         % only once

?- X = Y.
 X = _G204                  % variables remain uninstantiated, but will co-refer
 Y = _G204 ;                % _G204 is some implementation tag that represents
    their (shared) location
 no
```

Arithmetic

- The usual arithmetic operators are available
  - however, they are functors, not functions
  - +(2, 3) which can also be written as 2 + 3, is a two-argument structure, not a function call – it will not unify with 5:

```
?- X = 2 + 3.
X = 2 + 3
```

```
?- (2 + 3) = 5.
no
```

  - To actually compute the value of an expression – use the infix predicate is:
    - succeeds if it can unify its first argument (must be a variable) with the arithmetic value of its second argument

```
?- X is 1+2.
X = 3                    % evaluates the arithmetic expression
```

Arithmetic

- More examples:

```
?- 1+2 is 4-1.
no% first argument is not a variable

?- X is Y.
ERROR                 % second argument must be instantiated

?- Y is 1+2, X is Y.
 X = 3
 Y = 3                % Y is instantiated by the time it is needed
```

Arithmetic

- Predicates for comparing numbers:

X =:= Y	same number
X =\= Y	different numbers
X < Y	
X > Y	
X =< Y	less than or equal (not the usual <= notation)
X >= Y	

- Both arguments must be instantiated
- All these predicates also evaluate expressions:

```
?- 3 =:= 2+1.
yes

?- 3*2 < 7+1.
yes
```

Arithmetic

- Example (reigns of Princes of Wales in 9th and 10th centuries):

```
reigns(rhodri, 844, 878).
reigns(anarawd, 878, 916).
reigns(hywel_dda, 916, 950).
reigns(lago_ap_idwal, 950, 979).
reigns(hywel_ap_ieuaf, 979, 985).
reigns(cadwallon, 985, 986).
reigns(maredudd, 986, 999).

prince(X, Y) :- reigns(X, A, B), Y >= A, Y =< B.        % X was a prince during year Y


?- prince(cadwallon, 986).
yes

?- prince(X, 979).
X = lago-ap_idwal ;
X = hywel_ap_ieuaf ;
no
```

Arithmetic

- Example (compute the population density):

```
pop(india, 548).        % in millions (old data)
pop(china, 800).
pop(brazil, 108).

area(india, 1).         % in millions of square miles
area(china, 4).
area(brazil, 3).

density(X, Y) :- pop(X, P), area(X, A), Y is P/A.


?- density(china, X).
X = 200
yes

?- density(turkey, X).
no
```

Lists

- List – non-homogeneous collection of elements
  - Defined recursively

    .(a, .(b, .(c, [])))

    . is a functor (similar to cons) that builds a list from head and tail

    [] is the empty list

  - Shorthand notation:

    [a, b, c]

  - Can also specify a list by its first elements (not only head), and the rest (tail):

    [a | [b, c]]

    [a, b | [c]]

    [a, b, c | []]

Lists

- Instantiations with lists:

```
p([1, 2, 3]).
p([the, cat, sat, [on, the, mat]]).

?- p([X|Y]).

X = 1
Y = [2, 3] ;

X = the
Y = [cat, sat, [on, the, mat]] ;

no


?- p([_, _, _, [_|X]]).

X = [the, mat] ;

no
```

Lists

- Instantiations with lists:

List 1	List 2	Instantiations
[X, Y, Z]	[john, likes, fish]	X = john Y = likes Z = fish
[cat]	[X|Y]	X = cat Y = []
[X, Y|Z]	[mary, likes, wine]	X = mary Y = likes Z = [wine]
[[the, Y]|Z]	[[X, hare], [is, here]]	X = the Y = hare Z = [[is, here]]

Lists

- Instantiations with lists (cont.):

List 1	List 2	Instantiations
[golden|T]	[golden, norfolk]	T = [norfolk]
[black, horse]	[horse, X]	(none)
[white|Q]	[P|horse]	P = white Q = horse

- Proper and improper lists:

```
?- [white|[horse]] = .(white, .(horse, [])).
yes                          % proper list

?- [white|horse] = .(white, .(horse, [])).
no

?- [white|horse] = .(white, horse).
yes                          % improper list (tail is not a list)
```

Recursion

- Membership in a list:

1. Do not need to check the empty list – if member(X, [ ]) does not appear in database => it is false

2. Check the first element (head):

```
member(X, [H|T]) :- X = H.
```

or better:

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

or even better:

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

3. Check the rest of the list (tail):

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

<u>Note:</u> each recursive instance of a goal (member) is a new "copy" (with its own place marker in the database)

Recursion

- Using the member predicate:

- Check for membership:

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

- Enumerate elements of a list:

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

Recursion

- Using the member predicate (cont.):

- Search in a dictionary (list of pairs):

```
?- member([3,Y], [[1,a],[2,m],[3,z],[4,v],[3,p]]).
Y = z ;
Y = p ;
no
```

- Find elements in a list that satisfy some constraint (their square is < 100):

```
?- member(X, [23,9,19,45,6]), X*X < 100.
X = 9 ;
X = 6 ;
no
```

Recursion

- Check if a list is sorted:

```
sorted([]).                    % empty list is sorted
sorted([ _ ]).                 % list with one element is sorted
sorted([A,B|T]) :- A =< B, sorted([B|T]).
```

- Count the number of elements in a list:

```
nrelem([], 0).                 % empty list has 0 elements
nrelem([ _|T], N) :- nrelem(T, X), N is X+1.
```

  - What if we wrote:

```
nrelem([ _|T], N) :- nrelem(T, X), X is N-1.
```

  - Although logically correct, it does not work – after satisfying nrelem(T, X), X is instantiated and N is not; the is operator requires:
    - left operand must be uninstantiated
    - right operand must be an expression that can be evaluated

Recursion

- Append two lists into a third list:

```
append([], L, L).
append([H|T], L, [H|L1]) :- append(T, L, L1).
```

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

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

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

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

- In general, Prolog does not distinguish between "input" and "output" arguments
- Difference between functional and logic languages:
  - Functional languages – apply functions to input arguments in order to generate results
  - Logic languages – search for values for which a predicate is true

Recursion

- Append (cont.):

```
?- append(X, Y, [a,b,c,d]).

X = []          Y = [a,b,c,d] ;
X = [a]         Y = [b,c,d] ;
X = [a,b]       Y = [c,d] ;
X = [a,b,c]     Y = [d] ;
X = [a,b,c,d]   Y = [] ;
no
```

- Can also use append to split a list into 2 sublists

Recursion

- Check if list P is a prefix of list L:

```
prefix(P, L) :- append(P, _, L).

?- prefix(X, [a,b,c,d]).

X = [] ;

X = [a] ;

X = [a,b] ;

X = [a,b,c] ;

X = [a,b,c,d] ;

no
```

Recursion

- Similarly, check if list S is a suffix of list L:

```
suffix(S, L) :- append(_, S, L).
```

- Check if SubL is a sublist of list L:

(sublists of [a,b,c] are :
[ ], [a], [b], [c], [a,b], [b,c], [a,b,c])

```
sublist(SubL, L) :- suffix(S, L), prefix(SubL, S).
% SubL is a sublist of L if there is some suffix S of L of which SubL is a
  prefix
```

Recursion

- Given a list of a's and a list of b's, check if they have the same length:

```
a2b([], []).
a2b([a|T1], [b|T2]) :- a2b(T1, T2).
```

```
?- a2b([a,a,a], [b,b,b]).
yes

?- a2b([a,a,a,a], [b,b,b]).
no

?- a2b([a,c,a,a], [b,b,5,4]).
no

?- a2b([a,a,a,a], X).
X = [b,b,b,b] ;
no
```

```
?- a2b(X, Y).

X = []
Y = [] ;

X = [a]
Y = [b] ;

X = [a, a]
Y = [b, b] ;

X = [a, a, a]
Y = [b, b, b] ;

...
```

Announcements

- Readings
  - Prolog resources
- Homework
  - HW 7 out – due May 7
  - Submission
    - Submit in your code in Canvas as one “hw7.pl” file containing all your predicates.
    - The file must be able to load (with “consult”) and be tested in the interpreter.
      - Consequently, the file must be in a plain text format; do <u>not</u> submit Word, PDF, RTF, JPG or any such types of files.
      - Also make sure that any auxiliary information (such as your name or question numbers) is commented out.

Ordering

- In propositional logic, order does not matter:

    A and B <=> B and A
    A or B <=> B or A

- However, in Prolog order matters:
  - Subgoals are tried from left to right
  - Clauses are tried from top to bottom

```
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).
descend(X, Y) :- child(X, Y).
descend(X, Y) :- child(X, Z), descend(Z, Y).

?- descend(martha, rose).
yes
```

Ordering

- Same database:

child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y) :- child(X, Y).
~~descend(X, Y) :- child(X, Z), descend(Z, Y).~~

- Now change the order of subgoals in the last rule:

descend(X, Y) :- descend(Z, Y), child(X, Z).

- Same query:

?- descend(martha, rose).
yes % still works, but is less efficient

Ordering

- Same database:

child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

~~descend(X, Y) :- child(X, Y).~~
~~descend(X, Y) :- child(X, Z), descend(Z, Y).~~

- Now change:
  - the order of the two subgoals in the last rule
  - the order of the two rules

```
descend(X, Y) :- descend(Z, Y), child(X, Z).
descend(X, Y) :- child(X, Y).
```

- Same query:

```
?- descend(martha, rose).
ERROR: Out of local stack        % infinite loop
```

Ordering

- Consider again the member predicate:

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

?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

- Processing:

member(X, [1, 2, 3]	member(1, [1, 2, 3])	X = 1
member(X, [2, 3])	member(2, [2, 3])	X = 2
member(X, [3])	member(3, [3])	X = 3
member(X, [])		<fail>

<nothing else to try>

Ordering

- Member predicate - now change the order of the rules:

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

?- member(X, [1,2,3]).
X = 3 ;
X = 2 ;
X = 1 ;
no
```

- Processing:

member(X, [1, 2, 3])	member(X, [2, 3])	member(X, [3])	member(X, [])	<fail>
		member(3, [3])		X = 3
	member(2, [2, 3])			X = 2
member(1, [1, 2, 3])				X = 1
				<nothing else to try>

Accumulators

- Recall the following predicate (counts the number of elements in a list):

```
nrelem([], 0).
nrelem([_|T], N) :- nrelem(T, X), N is X+1.
```

- Is it tail recursive?
  - No – there is still work to do (N is X+1) after the recursive "call" (nrelem(T, X))

- A tail recursive version – update an accumulator (the number found so far):

```
nrelemTR(L, N) :- nrelemAcc(L, 0, N).        % initialize accumulator with 0

nrelemAcc([], A, A).
nrelemAcc([_|T], A, N) :- Anew is A+1, nrelemAcc(T, Anew, N).
```

Imperative Control Flow

- Control over the way Prolog performs backtracking
- The ! predicate (cut):

```
p(X) :- a(X).
p(X) :- b(X), c(X), !, d(X), e(X).
p(X) :- f(X).

q(X) :- p(X), foo.
```

- How ! works:
  - ! always succeeds
  - once ! is encountered, it commits Prolog to all choices made since that rule was chosen
  - if something later fails (for example d, e or foo)
    - no more backtracking (try to re-satisfy) over b and c
    - no attempt to choose another rule for p
    - however, normal backtracking over d and e

Imperative Control Flow

- Example - without cut:

```
p(X) :- a(X).
p(X) :- b(X), c(X), d(X), e(X).
p(X) :- f(X).

a(1). b(1). c(1).
b(2). c(2). d(2). e(2).
f(3).

?- p(X).

X = 1 ;
X = 2 ;
X = 3 ;
no
```

Imperative Control Flow

- Same example - with cut:

```
p(X) :- a(X).
p(X) :- b(X), c(X), !, d(X), e(X).
p(X) :- f(X).

a(1). b(1). c(1).
b(2). c(2). d(2). e(2).
f(3).
```

```
?- p(X).

X = 1 ;
no
```

  - What happens?

  - First rule is chosen, new goal is a(X), then a(x) matches with a(1), first solution is X = 1.
  - Look for more solutions
  - Second rule is chosen, new goal is b(X),c(X),!,d(X),e(X), then b(X) matches with b(1), now X is 1, then c(1) succeeds, then ! succeeds, then d(1) fails
  - Because of the !:
    - no attempt to resatisfy b and c
    - no attempt to choose another rule for p

Imperative Control Flow

- Common uses for the cut:

- Tell Prolog that it has found the right rule for a particular goal:
  - "if you got this far, you have picked the correct rule for this goal"

- Tell Prolog to fail a goal immediately, without trying for alternative solutions:
  - "if you got here, no solutions will be found – you should stop trying to satisfying this goal"

- Tell Prolog to stop searching for other solutions (the one found is the only one, or we don't care whether there are others):
  - "if you got here, there is no point in looking for alternatives"

Imperative Control Flow

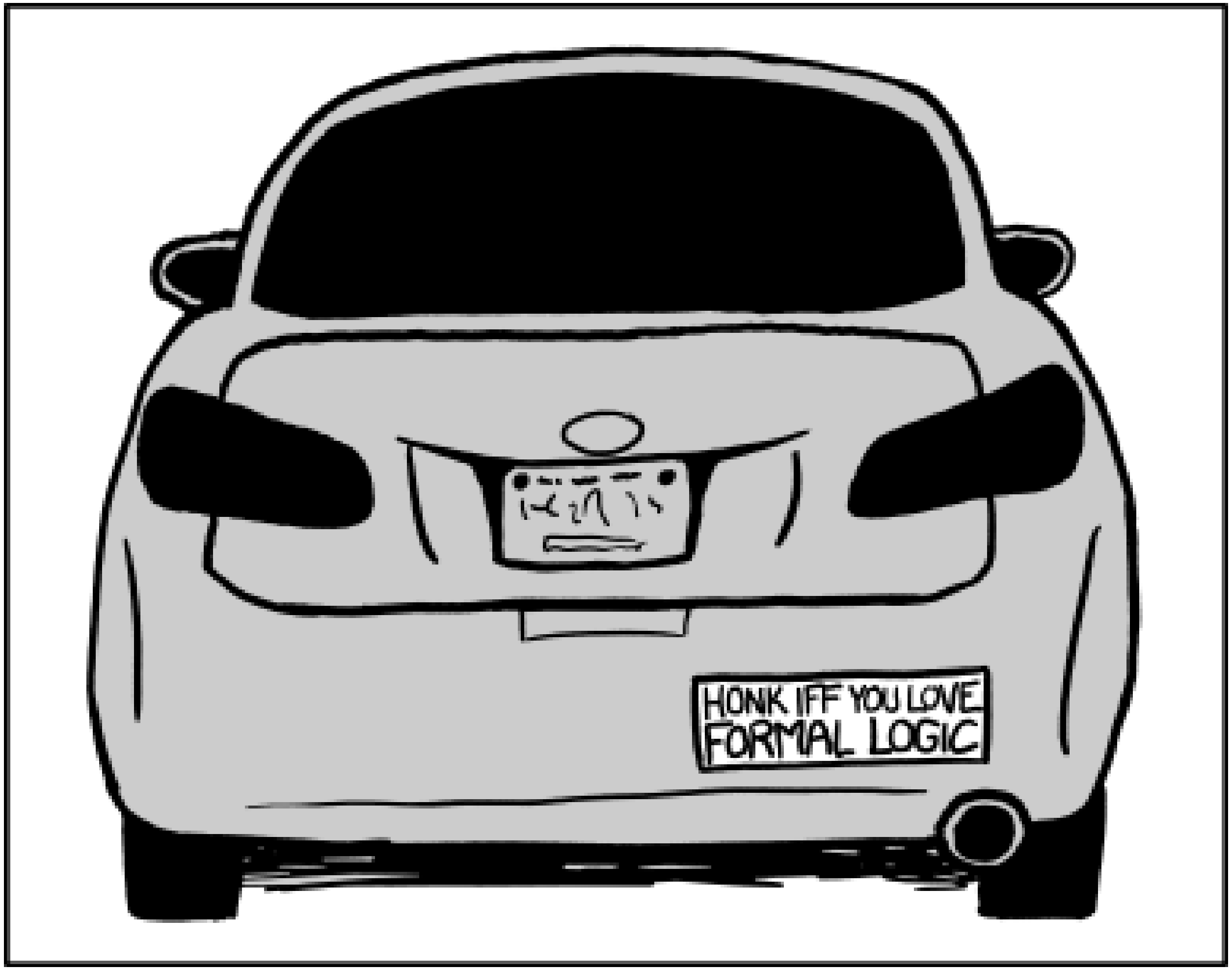
- Use cut to achieve the effect of if a then b else c:

```
p :- a, !, b.
p :- c.
```

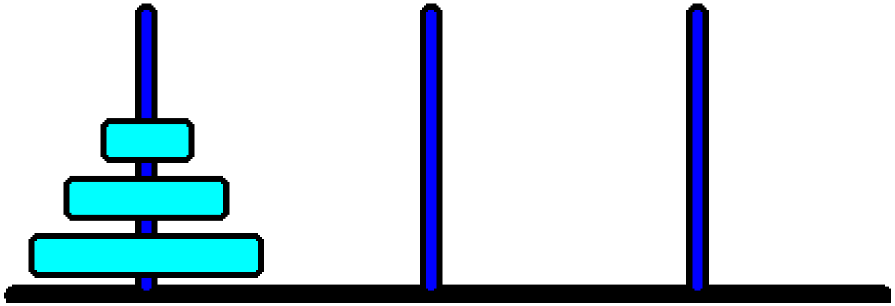
- Use cut to achieve the effect of a loop:

```
natural(1).                    % generate all natural numbers
natural(N) :- natural(M), N is M+1.

my_loop(N) :- natural(Ind),
                              write(Ind), nl,      % body of loop (nl prints a new line)
        Ind >= N, !.

?- my_loop(3).
1
2
3
yes
```

Formal Logic



Towers of Hanoi



- Goal – move N disks from left peg to right peg using the center peg as an auxiliary holding peg
- Only one disk can be moved at a time
- At no time can a larger disk be placed upon a smaller disk

- Recursive solution:
  - move N-1 disks from left to center (smaller version of same problem)
  - move the remaining 1 disk from left to right (elementary problem)
  - move the N-1 disks from center to right (smaller version of same problem)

Towers of Hanoi

- Predicate move(N, X, Y, Z) – moves N disks from X to Y using Z as auxiliary

```
move(1, X, Y, _) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.

move(N, X, Y, Z) :-
    N > 1,
    M is N-1,
    move(M, X, Z, Y),
    move(1, X, Y, _),
    move(M, Z, Y, X).
```

```
?- move(2, left, right, center).
Move top disk from left to center
Move top disk from left to right
Move top disk from center to right
yes

?- move(3, left, right, center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right
yes
```

Tic-Tac-Toe

- Board layout:

1	2	3
4	5	6
7	8	9

- A program for the x player:

```
% Ordered lines:
ordered_line(1,2,3).     ordered_line(4,5,6).
ordered_line(7,8,9).     ordered_line(1,4,7).
ordered_line(2,5,8).     ordered_line(3,6,9).
ordered_line(1,5,9).     ordered_line(3,5,7).

% Check if three cells are in line:
line(A,B,C) :- ordered_line(A,B,C).
line(A,B,C) :- ordered_line(A,C,B).
line(A,B,C) :- ordered_line(B,A,C).
line(A,B,C) :- ordered_line(B,C,A).
line(A,B,C) :- ordered_line(C,A,B).
line(A,B,C) :- ordered_line(C,B,A).
```

Tic-Tac-Toe

```
% How to make a move:
move(A) :- good(A), empty(A).

% Define full and empty cell:
full(A) :- x(A).
full(A) :- o(A).
empty(A) :- not(full(A)).

% Strategy (order is essential) - what is a good move:
good(A) :- win(A).
good(A) :- block_win(A).
good(A) :- split(A).
good(A) :- block_split(A).
good(A) :- build(A).
```

Tic-Tac-Toe

```
% Strategy
win(A) :- x(B), x(C), line(A,B,C).
block_win(A) :- o(B), o(C), line(A,B,C).
split(A) :- x(B), x(C), not(B=C), line(A,B,D), line(A,C,E), empty(D), empty(E).
```

A x
x o o

% place an x on A to create a split and win next move

```
block_split(A) :- o(B), o(C), not(B=C), line(A,B,D), line(A,C,E), empty(D), empty(E).
build(A) :- x(B), line(A,B,C), empty(C).

% Otherwise, choose any empty cell
% Priority is – center, corners, sides
good(5).
good(1). good(3). good(7). good(9).
good(2). good(4). good(6). good(8).
```

Tic-Tac-Toe

- Suppose that current configuration is stored in database:

x o
x
o

x(5). o(2).
x(1). o(8).

?- move(A).
A = 9 % move to win

- Another configuration:

x
o
x o

x(1). o(5).
x(8). o(9).

?- move(A).
A = 3 % move to block a split from the o player

Announcements

- Readings
  - Prolog resources