CS 403: Introduction to logic programming

Gregory Mierzwinski

Fall 2022

- A proposition is a logical statement that can be either false or true
- To work with propositions one needs a formal system i.e., a symbolic logic
- Predicate calculus or first-order logic is one such a logic
  - A term is a constant, structure, or variable
  - An atomic proposition (or predicate) denotes a relation. It is composed of a functor that names the relation, and an ordered list of terms (parameters):
    $\text{secure}(room)$, $\text{likes}(bob, steak)$, $\text{black}(crow)$, $\text{capital}(ontario, toronto)$
  - Variables can appear only as arguments. They are free:

    $$\text{capital}(ontario, X)$$

    unless bounded by one of the quantifiers $\forall$ and $\exists$:

    $$\exists X: \ \text{capital}(ontario, X) \qquad \forall Y: \ \text{capital}(Y, toronto)$$

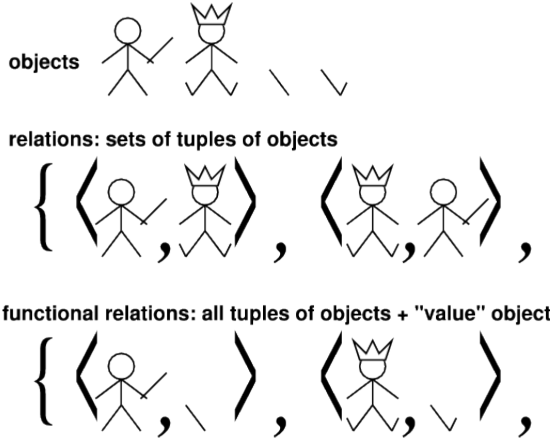
  - A compound proposition (formula) is composed of atomic propositions, connected by logical operators: $\neg$, $\wedge$, $\vee$, $\rightarrow$ ($\Rightarrow$). Variables are bound using quantifires

    $$\forall X.(\text{crow}(X) \rightarrow \text{black}(X))$$
    $$\exists X.(\text{crow}(X) \wedge \text{white}(X))$$
    $$\forall X.(\text{dog}(fido) \wedge (\text{dog}(X) \rightarrow \text{smelly}(X)) \rightarrow \text{smelly}(fido))$$

- The meaning is in the eye of the beholder
- Sentences are true with respect to a model and an interpretation
  - The model contains objects and relations among them (your view of the world)
  - An interpretation is a triple $I = (D, \phi, \pi)$, where
    - D (the domain) is a nonempty set; elements of D are individuals
    - ϕ is a mapping that assigns to each constant an element of D
    - π is a mapping that assigns to each predicate with n arguments a function $p : D^n \rightarrow \{True, False\}$ and to each function of k arguments a function $f : D^k \rightarrow D$
  - The interpretation specifies the following correspondences:

constant symbols	$\rightarrow$	objects (individuals)
predicate symbols	$\rightarrow$	relations
function symbols	$\rightarrow$	functional relations

    - An atomic sentence $predicate(term_1, \ldots, term_n)$ is true iff the objects referred to by $term_1, \ldots, term_n$ are in the relation referred to by $predicate$



- Objects (richard, kingJohn, leg1, leg2), predicates or relations (brother), functions (leftLegOf)

- Prolog is a logic/descriptive language
- Allows the specification of the problem to be solved using
  - Known facts about the objects in the universe of the problem (unit clauses):

```
locked(window).
dark(window).
capital(ontario,toronto).
```

  - Rules for inferring new facts from the old ones
  - Queries or goals about objects and their properties
    - The system answers such queries, based on the existing facts and rules

```
?- locked(window).
No
?- ['test.pl'].
Yes
?- locked(window).
Yes
?- locked(door).
No
```

- A variable in Prolog is anything that starts with a capital letter or an underscore ("_")
- A constant is a number or atom. An atom is:
  - Anything that starts with a lower case letter followed by letters, digits, and underscores
  - Any number of symbols `+,-,*,/,\,~,<,>,=,',^,:,.,?,@,#,$,$,&`
  - Any of the special atoms `[],{},!,;,%`
  - Anything surrounded by single quotes: `'atom surrounded by quotes!.'`
    - Escape sequence: just double the escaped character: `'insert '' in an atom'`

NB: The predicate calculus is called first-order logic because no predicate can take as argument another predicate, and no predicate can be a variable

Prolog rules

- A Horn clause is a conjunction in which exactly one atomic proposition is not negated

$$A \vee \neg B \vee \neg C \vee \neg D$$
$$B \wedge C \wedge D \rightarrow A$$

  - A sentence that contain exactly one atomic proposition is also a (degenerate form of a) Horn clause
  - Note in passing that not all the FOL formulae can be converted into a set of Horn clauses
- A Prolog program is a set of Horn clauses

Rules

- Natural Language:
  The window is locked. If the light is off and the door is locked, the room is secure. The light is off if the window is dark. The window is dark.
- Horn clauses:

$$\begin{array}{l} \text{locked}(\textit{window}) \\ \text{dark}(\textit{window}) \\ \text{off}(\textit{light}) \wedge \text{locked}(\textit{door}) \rightarrow \text{secure}(\textit{room}) \\ \text{dark}(\textit{window}) \rightarrow \text{off}(\textit{light}) \end{array}$$

- The Prolog program:

```
dark(window).
locked(window).
secure(room) :- off(light), locked(door).
off(light) :- dark(window).
```

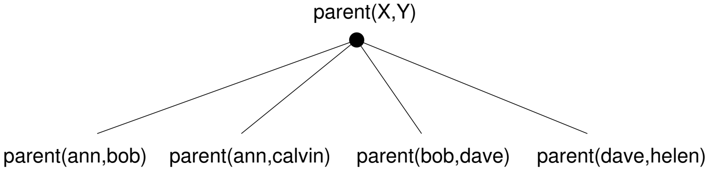
- Now, one can ask something:

```
?-  off(light).
Yes

?-  secure(room).
No

?-  locked(door).
No

?-  locked(Something).
Something = window
Yes

?-  locked(Something).
Something = window ;
No
```

  - Query variables are all existentially quantified

- A family tree:

```
parent(ann,bob).  parent(ann,calvin).
parent(bob,dave).  parent(dave,helen).
```



- Other family relations:

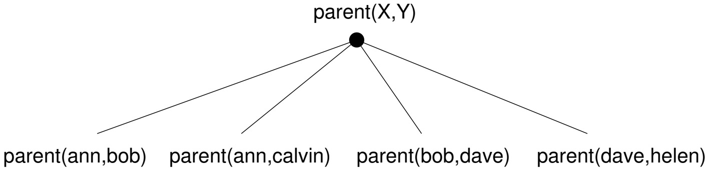
```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
siblings(X,Y) :- parent(Z,X), parent(Z,Y) .
```

- All the rule variables are universally quantified

CONJUNCTIVE RULES

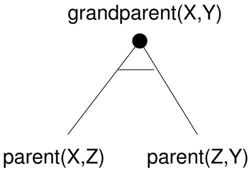
- A family tree:

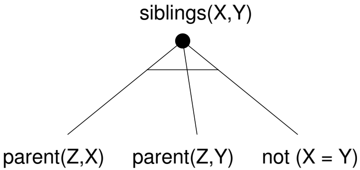
```
parent(ann,bob).   parent(ann,calvin).
parent(bob,dave).   parent(dave,helen).
```



- Other family relations:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
siblings(X,Y) :- parent(Z,X), parent(Z,Y) , not(X = Y).
```



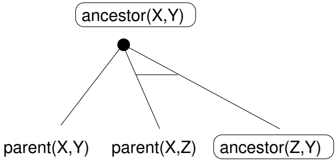


- All the rule variables are universally quantified

Disjunctive rules

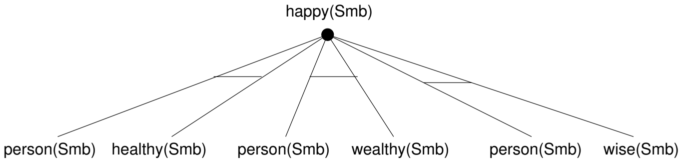
- Yet another family relation:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```



- A person is happy if she is healthy, wealthy, or wise:

```
happy(Smb) :- person(Smb), healthy(Smb).
happy(Smb) :- person(Smb), wealthy(Smb).
happy(Smb) :- person(Smb), wise(Smb).
```



Predicate Calculus proofs

- Inference rules $\rightarrow$ sound generation of new sentences from old
  - Prolog uses generalized modus ponens as inference rule

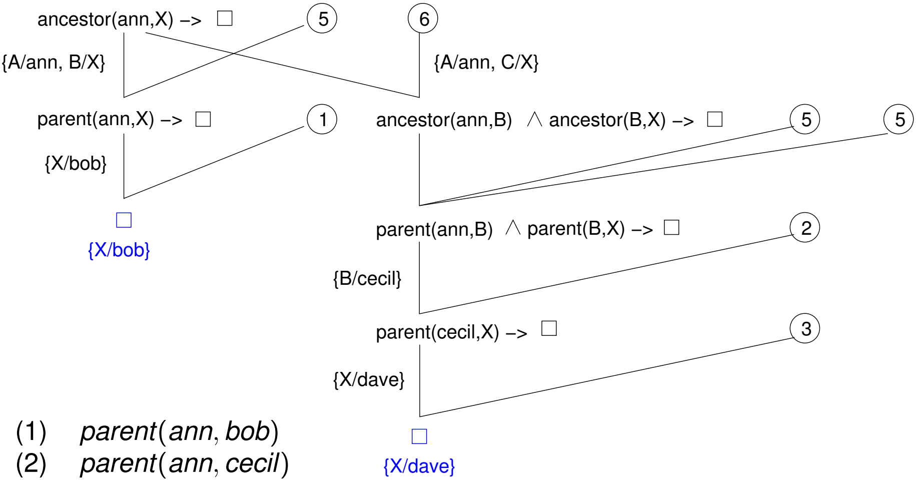
$$\frac{\alpha_1, \ldots, \alpha_n \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta}{\beta} \text{ (modus ponens)}$$

$$\frac{\begin{array}{l} \alpha_1, \ldots, \alpha_n \\ \alpha'_1 \wedge \cdots \wedge \alpha'_n \rightarrow \beta \\ \exists \sigma : (\alpha_1)_\sigma = (\alpha'_1)_\sigma \wedge \cdots \wedge (\alpha_n)_\sigma = (\alpha'_n)_\sigma \end{array}}{\beta_\sigma} \text{ (generalized modus ponens)}$$

- Proof $\rightarrow$ a sequence of applications of inference rules
  - Generates in effect a logically sound traversal of the proof tree

KB		
Bob is a buffalo	1.	$\mathit{buffalo}(\mathit{bob})$
Pat is a pig	2.	$\mathit{pig}(\mathit{pat})$
Buffaloes outrun pigs	3.	$\mathit{buffalo}(X) \wedge \mathit{pig}(Y) \rightarrow \mathit{faster}(X, Y)$
Query		
Is something outran		
by something else?		$\mathit{faster}(U, V)$
Negated query:	4.	$\mathit{faster}(U, V) \rightarrow \square$
(1), (2), and (3) with		
$\sigma = \{X/\mathit{bob}, Y/\mathit{pat}\}$	5.	$\mathit{faster}(\mathit{bob}, \mathit{pat})$
(4) and (5) with		
$\sigma = \{U/\mathit{bob}, V/\mathit{pat}\}$		$\square$

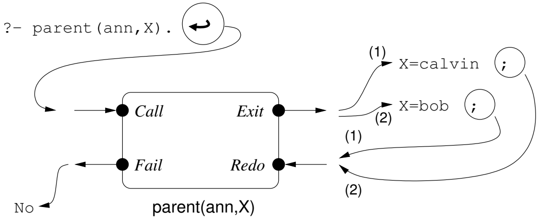
- All the substitutions regarding variables appearing in the query are typically reported (why?)



(1) $parent(ann, bob)$
(2) $parent(ann, cecil)$
(3) $parent(cecil, dave)$
(4) $parent(cecil, eric)$
(5) $parent(A, B) \rightarrow ancestor(A, B)$
(6) $ancestor(A, B) \wedge ancestor(B, C) \rightarrow ancestor(A, C)$

```
parent(ann,calvin).
parent(ann,bob).
parent(bob,dave).
parent(dave,helen).
```

```
2 ?- trace(parent).
        parent/2: call redo exit fail
Yes
[debug] 3 ?- parent(ann,X).
T Call:  (  7) parent(ann, _G365)
T Exit:  (  7) parent(ann, calvin)

X = calvin ;
T Redo:  (  7) parent(ann, _G365)
T Exit:  (  7) parent(ann, bob)

X = bob ;
No
```



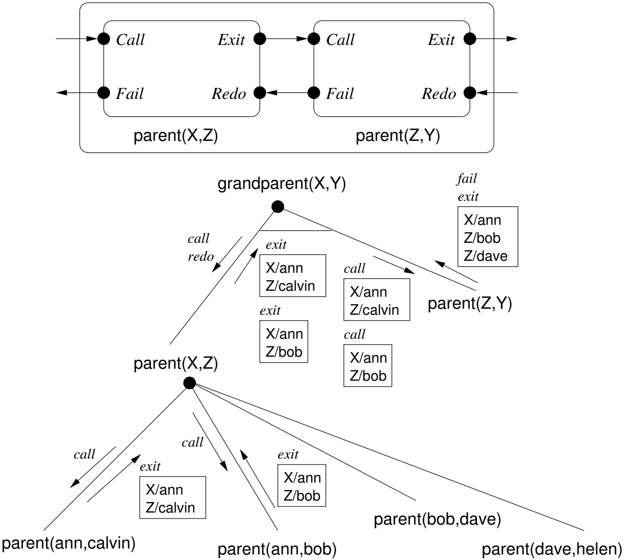
Searching the knowledge base (cont'd)

```
parent(ann,calvin).       parent(ann,bob).
parent(bob,dave).         parent(dave,helen).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
```

```
[debug] 8 ?- grandparent(X,Y).
T Call: (7) grandparent(_G382, _G383)
T Call: (8) parent(_G382, _L224)
T Exit: (8) parent(ann, calvin)
T Call: (8) parent(calvin, _G383)
T Fail: (8) parent(calvin, _G383)
T Redo: (8) parent(_G382, _L224)
T Exit: (8) parent(ann, bob)
T Call: (8) parent(bob, _G383)
T Exit: (8) parent(bob, dave)
T Exit: (7) grandparent(ann, dave)

X = ann
Y = dave ;
```

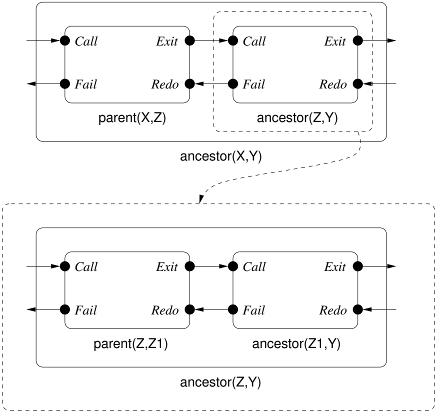
```
T Redo: (8) parent(_G382, _L224)
T Exit: (8) parent(bob, dave)
T Call: (8) parent(dave, _G383)
T Exit: (8) parent(dave, helen)
T Exit: (7) grandparent(bob, helen)

X = bob
Y = helen ;
T Redo: (8) parent(_G382, _L224)
T Exit: (8) parent(dave, helen)
T Call: (8) parent(helen, _G383)
T Fail: (8) parent(helen, _G383)
T Fail: (7) grandparent(_G382, _G383)

No
```

Searching the Knowledge Base (cont'd)



```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

- A recursive call is treated as a brand new call, with all the variables renamed



Unification

- There is no explicit assignment in Prolog
- Bindings to variables are made through the process of unification, which is done automatically most of the time
  - The predicate **=/2** is used to request an explicit unification of its two arguments

    ```
    ?- book(prolog,X) = book(Y,brna).
    X = brna
    Y = prolog
    ```

  - The binding {X/brna,Y/prolog} is the most general unifier
  - The most general unifier can contain free variables: the general unifier of `book(prolog,X) = book(Y,Z)` is {Y/brna,X/Z}
    - even if {Y/prolog,X/brna,Z/brna} is also a unifier, it is not the most general
- In passing, note that the following predicates are different, even if they have the same name

```
tuple(1,2).       % tuple/2
tuple(1,2,3).     % tuple/3
tuple(a,b,c).     % tuple/3
tuple(a,b,c,d).   % tuple/4
```

```
?- tuple(X,Y).
X = 1
Y = 2 ;
No
```

algorithm UNIFY(T_1, T_2, S) **returns** substitution or FAILURE:

- Input: T_1, T_2: the structures to unify; S: the substitution representing the variable bindings that are already in place
  - Initial call is typically made with an empty substitution: UNIFY(T_1, T_2, $\emptyset$)
- Output: A new substitution (including S) or the special value FAILURE specifying that the unification has failed

1. **if** T_1 and T_2 are both atoms, or bound to atoms in S **and** $T_1 == T_2$ **then return** S
2. **if** T_1 is a free variable **then return** $S \cup \{T_1/T_2\}$
3. **if** T_2 is a free variable **then return** $S \cup \{T_2/T_1\}$
4. **if** $T_1 == p(a_1, a_2, \ldots, a_n)$ and $T_2 == p(b_1, b_2, \ldots, b_n)$ (by themselves or because they are bound in S to such values) **then**
   1. **for** $i = 1$ **to** n **do**
      1. **let** $A =$ UNIFY(a_i, b_i, S), $S = S \cup A$
      2. **if** $A ==$ FAILURE **then return** FALURE
   2. **return** S
5. **return** FAILURE

Unification (cont'd)

- Unification can be attempted between any two Prolog entities. Unification succeeds of fails. As a side effect, free variables may become bound

```
[debug] 10 ?- parent(ann,Y).
T Call:  (  7) parent(ann, _G371)
T Exit:  (  7) parent(ann, calvin)

Y = calvin
Yes
```

```
[debug] 11 ?- parent(X,ann).
T Call:  (  7) parent(_G370, ann)
T Fail:  (  7) parent(_G370, ann)

No
```

- Once a variable is bound through some unification process, it cannot become free again

```
[debug] 15 ?- X=1, X=2.
T Call:  (  7) _G340=1
T Exit:  (  7) 1=1
T Call:  (  7) 1=2
T Fail:  (  7) 1=2

No
```

- Do not confuse **=/2** with assignment!

- What is the result of `X = pair(1,2)`?

- A structure has the same syntax as a predicate. The difference is that a structure appears as a parameter
- You do not have to define a structure, you just use it.
  - This is possible because of the unification process

- What is the result of `X = pair(1,2)`?

```
?- X = pair(1,2).
X = pair(1, 2)
```

- A structure has the same syntax as a predicate. The difference is that a structure appears as a parameter
- You do not have to define a structure, you just use it.
  - This is possible because of the unification process
- Example: binary search trees

```
% if I found the element, then succeed.
member_tree(X,tree(X,L,R)).

% Otherwise, if my element is larger than the current key, then I
% search in the right child.
member_tree(X,tree(Y,L,R)) :- X > Y, member_tree(X,R).

% Eventually (otherwise) search in the left child.
member_tree(X,tree(Y,L,R)) :- X < Y, member_tree(X,L).

% An empty tree cannot contain any element, so anything else fails.
```

SEARCH TREES (CONT'D)

```
?- member_tree(3,nil).
No

[debug] ?-  member_tree(3,tree(2,tree(1,nil,nil),tree(3,nil,nil))).
T Call: (  7) member_tree(3, tree(2, tree(1, nil, nil), tree(3, nil, nil)))
T Call: (  8) member_tree(3, tree(3, nil, nil))
T Exit: (  8) member_tree(3, tree(3, nil, nil))
T Exit: (  7) member_tree(3, tree(2, tree(1, nil, nil),tree(3, nil, nil)))
Yes

[debug] ?- member_tree(5,tree(2,tree(1,nil,nil),tree(3,nil,nil))).
T Call: (  7) member_tree(5, tree(2, tree(1, nil, nil),tree(3, nil, nil)))
T Call: (  8) member_tree(5, tree(3, nil, nil))
T Call: (  9) member_tree(5, nil)
T Fail: (  9) member_tree(5, nil)
T Redo: (  8) member_tree(5, tree(3, nil, nil))
T Fail: (  8) member_tree(5, tree(3, nil, nil))
T Redo: (  7) member_tree(5, tree(2, tree(1, nil, nil),tree(3, nil, nil)))
T Fail: (  7) member_tree(5, tree(2, tree(1, nil, nil),tree(3, nil, nil)))
No
```

LISTS

- Lists are nothing special, just a structure named ".", and containing two parameters
  - The first one is the elements at the head of the list,
  - The second is a structure ".", or the empty list "[]"
- That is, `.(X,XS)` is equivalent to Haskell's `(x::xs)`
- The difference from Haskell is given by the absence of types in Prolog: A list can contain any kind of elements
- As in Haskell, there is some syntactic sugar:
  - One can enumerate the elements: `[1,[a,4,10],3]`
  - The expression `[X|Y]` is equivalent to `.(X,Y)`
  - We also have the equivalence between `[X,Y,Z|R]` and `.(X,.(Y,.(Z,R)))`, and so on

```
?- [b,a,d] = [d,a,b].
?- [X|Y] = [a,b,c].
?- [X|Y] = [].
?- [[X1|X2]|X3] = [[1,2,3],4,5].
```

- The absence of types in Prolog is brought to extremes: the list `[1]` is the structure `.(1,[])`. However, the empty list `[]` is an atom!

- Lists are nothing special, just a structure named ".", and containing two parameters
  - The first one is the elements at the head of the list,
  - The second is a structure ".", or the empty list "[]"
- That is, `.(X,XS)` is equivalent to Haskell's `(x::xs)`
- The difference from Haskell is given by the absence of types in Prolog: A list can contain any kind of elements
- As in Haskell, there is some syntactic sugar:
  - One can enumerate the elements: `[1,[a,4,10],3]`
  - The expression `[X|Y]` is equivalent to `.(X,Y)`
  - We also have the equivalence between `[X,Y,Z|R]` and `.(X,.(Y,.(Z,R)))`, and so on

`?- [b,a,d] = [d,a,b].`	$\rightarrow$	unification failure
`?- [X|Y] = [a,b,c].`	$\rightarrow$	`X=a,Y=[b,c]`
`?- [X|Y] = [].`	$\rightarrow$	unification failure
`?- [[X1|X2]|X3] = [[1,2,3],4,5].`	$\rightarrow$	`X1=1,X2=[2,3],X3=[4,5]`

- The absence of types in Prolog is brought to extremes: the list `[1]` is the structure `.(1,[])`. However, the empty list `[]` is an atom!

- Membership: `member/2`

- Membership: `member/2`
  ```
  member(X,[X|_]).
  member(X,[_|Y]) :- member(X,Y).
  ```
- What is the answer to the query `?- member(X,[1,2,3,4]).`

- Membership: `member/2`

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

- What is the answer to the query `?- member(X,[1,2,3,4]).`
  - Both arguments of `member/2` may be bound – if so we write `member(+E,+L)`
  - In fact, `member/2` also works as `member(-E,+L)` (first argument free)
  - The general specification is `member(?E,+L)` (last argument is bound, but the first is not necessarily bound)

LIST PROCESSING

- Membership: `member/2`
  ```
  member(X,[X|_]).
  member(X,[_|Y]) :- member(X,Y).
  ```
- What is the answer to the query `?- member(X,[1,2,3,4]).`
  - Both arguments of `member/2` may be bound – if so we write `member(+E,+L)`
  - In fact, `member/2` also works as `member(-E,+L)` (first argument free)
  - The general specification is `member(?E,+L)` (last argument is bound, but the first is not necessarily bound)
- There are no functions in Prolog. What if we want that our program to compute a value?

LIST PROCESSING

- Membership: `member/2`
```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```
- What is the answer to the query `?- member(X,[1,2,3,4]).`
  - Both arguments of `member/2` may be bound – if so we write `member(+E,+L)`
  - In fact, `member/2` also works as `member(-E,+L)` (first argument free)
  - The general specification is `member(?E,+L)` (last argument is bound, but the first is not necessarily bound)
- There are no functions in Prolog. What if we want that our program to compute a value?
  - We invent a new variable that will be bound to the result by various unification processes

- Membership: `member/2`
```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```
- What is the answer to the query `?- member(X,[1,2,3,4]).`
  - Both arguments of `member/2` may be bound – if so we write `member(+E,+L)`
  - In fact, `member/2` also works as `member(-E,+L)` (first argument free)
  - The general specification is `member(?E,+L)` (last argument is bound, but the first is not necessarily bound)
- There are no functions in Prolog. What if we want that our program to compute a value?
  - We invent a new variable that will be bound to the result by various unification processes
- A predicate for concatenating (”appending”) two lists: `append/3`
```
append([],L,L).
append([X|R],L,[X|R1]) :- append(R,L,R1).
```

- Membership: `member/2`

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

- What is the answer to the query `?- member(X,[1,2,3,4]).`
  - Both arguments of `member/2` may be bound – if so we write `member(+E,+L)`
  - In fact, `member/2` also works as `member(-E,+L)` (first argument free)
  - The general specification is `member(?E,+L)` (last argument is bound, but the first is not necessarily bound)
- There are no functions in Prolog. What if we want that our program to compute a value?
  - We invent a new variable that will be bound to the result by various unification processes
- A predicate for concatenating (”appending”) two lists: `append/3`

```
append([],L,L).
append([X|R],L,[X|R1]) :- append(R,L,R1).
```

- What is the result of the query `?- append(X,Y,[1,2,3,4]).`

- What means “3+4” to Prolog? (as in `?- X = 3 + 4.`)

- What means “3+4” to Prolog? (as in `?- X = 3 + 4.`)
- In order to actually evaluate an arithmetic expression, one must use the operator `is(?Var,+Expr)`:

```
?- X is 3+4
X = 7
Yes
```

- Example: A Prolog program that receives one number *n* and computes *n*!

```
fact(1,1).
fact(N,R) :- R is N*fact(N-1,R1).


13 ?- fact(1,X).
X = 1
Yes
```

- What means "3+4" to Prolog? (as in `?- X = 3 + 4.`)
- In order to actually evaluate an arithmetic expression, one must use the operator `is(?Var,+Expr)`:

```
?- X is 3+4
X = 7
Yes
```

- Example: A Prolog program that receives one number *n* and computes *n*!

```
fact(1,1).
fact(N,R) :- R is N*fact(N-1,R1).
```

```
13 ?- fact(1,X).
X = 1
Yes
14 ?- fact(2,X).
[WARNING: Arithmetic:  'fact/2' is not a function]
 Exception:  ( 8) _G185 is 2*fact(2-1, _G274) ?
[WARNING: Unhandled exception]
```

Numbers and Operations on Numbers

- What means "3+4" to Prolog? (as in `?- X = 3 + 4.`)
- In order to actually evaluate an arithmetic expression, one must use the operator `is(?Var,+Expr)`:

```
?- X is 3+4
X = 7
Yes
```

- Example: A Prolog program that receives one number *n* and computes *n*!

```
fact(1,1).

fact(N,R) :- N1 is N-1, fact(N1,R1), R is N*R1.
```

- All the expected operators on numbers work as expected
  - One (annoying) difference: the operator for $\leq$ is not <=, but =< instead

- Given the call `fact(5,X)`, what happens if one requests a new solution after Prolog answers `X=120`? Why? How to fix?

```
fact(1,1).
fact(N,R) :- N1 is N-1, fact(N1,R1), R is N*R1.

?- fact(5,X).

X = 120 ;
???
```

NEGATION AS FAILURE

- Negation in Prolog: `not/1` or `\+/1`
- Prolog assumes the closed world paradigm. The negation is therefore different from logical negation:

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

?- not(member(X,[1,2,3])).
No

?- not(not(member(X,[1,2,3]))).
X = _G332 ;
No
```

  - `not/1` fails upon resatisfaction (a goal can fail in only one way)
  - `not/1` does not bind variables

NEGATION IN CASE SELECTIONS

```
positive(X) :- X > 0.
negative(X) :- X < 0.

sign(X,+) :- positive(X).
sign(X,-) :- negative(X).
sign(X,0).

sign1(X,+) :- positive(X).
sign1(X,-) :- negative(X).
sign1(X,0) :- not(positive(X)), not(negative(X)).


?- sign(1,X).
X = + ;
X = 0 ;
No

?- sign1(1,X).
X = + ;
No
```

State space search

- The concept of state space search is widely used in AI
  - Idea: a problem can be solved by examining the steps which might be taken towards its solution
  - Each action takes the solver to a new state
  - The solution to such a problem is a list of steps leading from the initial state to a goal state
- Classical example: A Farmer who needs to transport a Goat, a Wolf and some Cabbage across a river one at a time. The Wolf will eat the Goat if left unsupervised. Likewise the Goat will eat the Cabbage. How will they all cross the river?
  - A state is described by the positions of the Farmer, Goat, Wolf, and Cabbage
  - The solver can move between states by making a “legal” move (which does not result in something being eaten)
- General form for a state space search problem:
  - Input:
    1. The start state
    2. One (or more) goal states or final states
    3. The state transition function, or how to get from one state to another
  - Output: a list of moves or state transitions that lead from the initial state to one of the final states

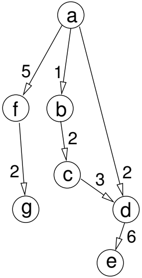
- Prolog already does it:

```
search(Final,Final,[]).
search(Current,Final,[M|Result]) :-
    move(Current,SomeState,M),
    search(SomeState,Final,Result).
```

- The only trick is that Prolog does not explain how it reached the goal state; it just states whether a goal state is reachable or not
- So we also need to provide a way to report the list of moves (hence the third parameter)

- Finding a path in a directed, acyclic graph:
  - A state is a vertex of the graph

```
distance(a,f,5).
distance(f,g,2).
distance(a,b,1).
distance(a,d,2).
distance(b,c,2).
distance(c,d,3).
distance(d,e,6).
move(A,B,to(A,B)) :- distance(A,B,_).
```




```
?- search(a,e,R).
R = [to(a,b),to(b,c),to(c,d),to(d,e)] ;
R = [to(a,d),to(d,e)] ;
No
?- search(e,a,R).
No
```

Searching a state space, revised

- Often, the search space contains cycles. Then, Prolog search strategy may fail to produce a solution.

```
move(A,B,to(A,B)) :- distance(A,B,_).
move(A,B,to(A,B)) :- distance(B,A,_).

?- search(a,e,R).
ERROR: Out of local stack
```

- We can use then a generate and test technique:
  - We keep track of the previously visited states
  - Then, we generate a new state (as before), but we also test that we haven't been in that state already; we proceed forward only if the test succeeds

```
search(Initial,Final,Result) :-
    search(Initial,Final,[Initial],Result).
search(Final,Final,_,[]).
search(Crt,Final,Visited,[M|Result]) :-
    move(Crt,AState,M),             % generate
    not(member(AState,Visited)),   % test
    search(AState,Final,[AState|Visited],
           Result).
```

```
?- search(a,e,R).
R = [to(a, b), to(b, c),
     to(c, d), to(d, e)] ;
R = [to(a, d), to(d, e)] ;
No
```

Things to do for solving a specific state space search problem:

- Establish what is a state for your problem and how will you represent it in Prolog
- Establish your state transition function; that is, define the `move/3` predicate
  - Such a predicate should receive a state, and return another state together with the move that generates it
  - Upon resatisfaction, a new state should be returned
  - If no new state is directly accessible from the current one, `move/3` should fail

LIMITATIONS

- The predicate `search/3` works on any finite search space
  - It exploits the fact that Prolog performs by itself a depth-first search.
    - Since the depth-first search is not guaranteed to terminate on an infinite search space, neither is `search/3`
  - It is possible to implement a breadth-first search in Prolog
    - However, this cannot take advantage of the search strategy which is built in the Prolog interpreter (in fact, it sidesteps it altogether)
    - Such an implementation is thus more complicated and exceeds the scope of this course (but if you are really curious, contact me)

```
% A state: [Boat,Cabbage,Goat,Wolf]
% Moving around.  We use the "generate and test" paradigm:
move(A,B,M) :- move_attempt(A,B,M), legal(B).

% first, attempt to move the Cabbage, then the Goat, then the Wolf:
move_attempt([B,B,G,W],[B1,B1,G,W], moved(cabbage,B,B1)) :- opposite(B,B1).
move_attempt([G,B,G,W],[G1,B,G1,W], moved(goat,G,G1)) :- opposite(G,G1).
move_attempt([W,B,G,W],[W1,B,G,W1], moved(wolf,W,W1)) :- opposite(W,W1).
%... eventually, move the empty boat:
move_attempt([X,C,G,W],[Y,C,G,W], moved(nothing,X,Y)) :- opposite(X,Y).

opposite(south,north).      opposite(north,south).

% Make sure that nothing gets eaten:
legal(State) :- not(conflict(State)).
% we cannot allow the Cabbage and the Goat on the same shore unsupervised
conflict([B,C,C,W]) :- opposite(C,B).
% ... nor the Goat and the Wolf...
conflict([B,C,W,W]) :- opposite(W,B).
% ... but anything else is fine.
```

On Goats, Wolves, and Cabbage (cont'd)

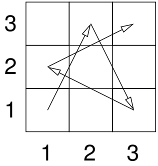
```
?- search([north,north,north,north],
          [south,south,south,south], R).

R = [moved(goat, north, south),
     moved(nothing, south, north),
     moved(cabbage, north, south),
     moved(goat, south, north),
     moved(wolf, north, south),
     moved(nothing, south, north),
     moved(goat, north, south)] ;

R = [moved(goat, north, south),
     moved(nothing, south, north),
     moved(wolf, north, south),
     moved(goat, south, north),
     moved(cabbage, north, south),
     moved(nothing, south, north),
     moved(goat, north, south)] ;
```

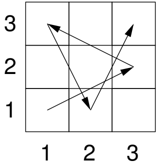
On Knights and their tours

```
% The board size is given by the predicate size/1
size(3).
% The position of the Knight is represented by the structure -(X,Y)
% (or X-Y), where X and Y are the coordinates of the square where the
% Knight is located. We represent a move by the position it generates.

% We use, again, the generate and test technique:
move(A,B,B) :- move_attempt(A,B), inside(B).
% There are 8 possible moves in the middle of the board:
move_attempt(I-J, K-L) :- K is I+1, L is J-2.
move_attempt(I-J, K-L) :- K is I+1, L is J+2.
move_attempt(I-J, K-L) :- K is I+2, L is J+1.
move_attempt(I-J, K-L) :- K is I+2, L is J-1.
move_attempt(I-J, K-L) :- K is I-1, L is J+2.
move_attempt(I-J, K-L) :- K is I-1, L is J-2.
move_attempt(I-J, K-L) :- K is I-2, L is J+1.
move_attempt(I-J, K-L) :- K is I-2, L is J-1.
% However, if the Knight is somwhere close to board's margins, then
% some moves might fall out of the board.
inside(A-B) :- size(Max), A > 0, A =< Max, B > 0, B =< Max.
```

On Knights and their tours (cont'd)

```
?- search(1-1,3-3,R).

R = [2-3, 3-1, 1-2, 3-3] ;

R = [3-2, 1-3, 2-1, 3-3] ;

No
```





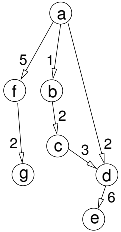
- Since our `search/3` predicate generates all the possible solutions, we can use it within another generate and test process!
- On a 4 × 4 board, a Knight moves from one square *S* to another square *D*. For a given *N*, find all the paths between *S* and *D* in which the Knight does not make more than *N* moves.

```
search_shorter(S,D,N,Result) :- search(S,D,Result),        % generate
                                length(Result,L), L =< N.  % test

% length([],0).
% length([_|T],L) :- length(T,L1), L is L1+1.

?- search_shorter(1-1,4-3,5,R).

R = [2-3, 3-1, 4-3] ;                   R = [3-2, 2-4, 4-3] ;
R = [2-3, 3-1, 1-2, 2-4, 4-3] ;         R = [3-2, 2-4, 1-2, 3-1, 4-3] ;
R = [2-3, 4-4, 3-2, 2-4, 4-3] ;         R = [3-2, 1-3, 3-4, 2-2, 4-3] ;
R = [2-3, 4-2, 3-4, 2-2, 4-3] ;         No
R = [3-2, 4-4, 2-3, 3-1, 4-3] ;

?- search_shorter(1-1,4-3,4,R).

R = [2-3, 3-1, 4-3] ;          R = [3-2, 2-4, 4-3] ;          No
```

VARIATIONS ON A SEARCH THEME (CONT'D)

- Given some integer *n* and two vertices *A* and *B*, is there a path from *A* to *B* of weight smaller than *n*?

```
distance(a,f,5).
distance(f,g,2).
distance(a,b,1).
distance(a,d,2).
distance(b,c,2).
distance(c,d,3).
distance(d,e,6).
move(A,B,to(A,B,C)) :- distance(A,B,C).
move(A,B,to(A,B,C)) :- distance(B,A,C).
weight([],0).
weight([to(_,_,C)|P],W) :- weight(P,W1), W is W1+C.

smaller(A,B,N,Result) :- search(A,B,Result),
                         weight(Result,W), W =< N.
```



Logic programming

	Logic programming	Ordinary programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Coffee break	Figure out solution
4.	Encode information in KB	Program solution
5.	Encode problem instance as facts	Encode problem instance as data
6.	Ask queries	Apply program to data
7.	Find false facts	Debug procedural errors