KNOWLEDGE REPRESENTATION

- A proposition is a logical statement that can be either false or true.
- In order to work with propositions, one needs a formal system, that is, a symbolic logic.
- A particular form used in symbolic logic is the predicate calculus, or the first-order logic.
  - Another Turing complete formalism.

THE PREDICATE CALCULUS

- 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): $\mathsf{secure}(room)$, $\mathsf{likes}(bob, steak)$, $\mathsf{black}(crow)$, $\mathsf{capital}(ontario, toronto)$.
- Variables can appear only as arguments. They are free:

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

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

  $$\exists X : (\mathsf{capital}(ontario, X))$$
  $$\forall Y : (\mathsf{capital}(Y, toronto))$$
- A compound proposition (formula) is composed of atomic propositions, connected by logical operators: $\neg$, $\wedge$, $\vee$, $\rightarrow$ ($\Rightarrow$). Variables may be bound using quantifires.

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

SEMANTICS OF THE PREDICATE CALCULUS

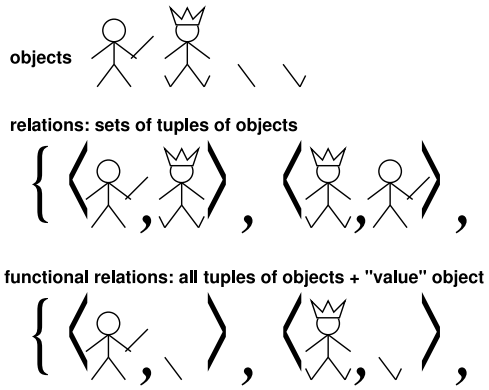
- Sentences are true with respect to a model and an interpretation.
  - The model contains objects and relations among them
  - An interpretation is a triple $I = (D, \phi, \pi)$, where
    - D (the domain) is a nonempty set; elements of D are individuals.
    - ϕ is a mapping that assigns to each constant 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 referents for

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 $predicate$.

SEMANTICS OF THE PREDICATE CALCULUS: EXAMPLE



KNOWLEDGE REPRESENTATION IN PROLOG

- 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 program answers such queries, based on the existing facts and rules.

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


CONSTANTS AND VARIABLES

- 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 the following symbols:
    `+,-,*,/,\,~,<,>,=,',^,:,.,?,@,#,$,$,&`.
  - Any of the special atoms `[],{},!,;,%`.
  - Anything surrounded by single quotes:
    `'this is an atom surrounded by quotes!.'`
    - Escape sequence: just double the escaped character:
      `'how to 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.

CLAUSAL FORM

- There are many ways of stating the same formula:

$$\forall X.\forall Y.(p(X) \rightarrow q(Y))$$
$$\forall X.\forall Y.(\neg p(X) \vee q(Y))$$

- But any formula can be expressed in the conjunctive normal form:

$$(A_{11} \vee A_{12} \vee \ldots A_{1n_1}) \wedge (A_{21} \vee A_{22} \vee \ldots A_{2n_2}) \wedge \ldots \wedge (A_{k1} \vee A_{k2} \vee \ldots A_{kn_k})$$

- Then, we can drop the $\wedge$ operators, we obtain in this way a set of clauses, and thus we get the clausal form:

$$\begin{array}{ll} \{ & (A_{11} \vee A_{12} \vee \ldots \vee A_{1n_1}), \\ & (A_{21} \vee A_{22} \vee \ldots \vee A_{2n_2}), \\ & \ldots \\ & (A_{k1} \vee A_{k2} \vee \ldots \vee A_{kn_k}) \quad \} \end{array}$$

INTERLUDE: CONVERSION TO CLAUSAL FORM

Any sentence (or KB) can be transformed into a clausal form.

$$\neg((a \leftrightarrow b) \vee (c \rightarrow \neg(d \wedge (f \rightarrow e))))$$

1. Eliminate $\leftrightarrow$ and $\rightarrow$: $\alpha \rightarrow \beta$ is changed to $\neg\alpha \vee \beta$, and $\alpha \leftrightarrow \beta$ is equivalent to $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.
$$\neg(((\neg a \vee b) \wedge (\neg b \vee a)) \vee (\neg c \vee (\neg(d \wedge (\neg f \vee e)))))$$
2. Apply De Morgan rules to move all the negations in, and remove double negations.
$$\neg((\neg a \vee b) \wedge (\neg b \vee a)) \wedge \neg(\neg c \vee (\neg(d \wedge (\neg f \vee e))))$$
$$(\neg(\neg a \vee b) \vee \neg(\neg b \vee a)) \wedge (\neg\neg c \wedge (\neg\neg(d \wedge (\neg f \vee e))))$$
$$((a \wedge \neg b) \vee (b \wedge \neg a)) \wedge (c \wedge (d \wedge (\neg f \vee e)))$$
3. Use the distributedness, associativity and commutativity to move the $\wedge$'s out: $\alpha \vee (\beta \wedge \gamma)$ becomes $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$.
$$((a \vee (b \wedge \neg a)) \wedge (\neg b \vee (b \wedge \neg a))) \wedge c \wedge d \wedge (\neg f \vee e)$$
$$(a \vee b) \wedge (a \vee \neg a) \wedge (\neg b \vee b) \wedge (\neg b \vee \neg a) \wedge c \wedge d \wedge (\neg f \vee e)$$
$$(a \vee b) \wedge (\neg b \vee \neg a) \wedge c \wedge d \wedge (\neg f \vee e)$$
4. Clausal form is therefore:
$$\{ (a \vee b), (\neg b \vee \neg a), c, d, (\neg f \vee e) \}$$

CLAUSAL FORM (CONT'D)

- A Horn clause is a clause 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 clause that contain exactly one atomic proposition is also a (degenerate form of) Horn clause.

- Any FOL formula can be converted in conjunctive normal form, but 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.

- Clausal form:

  $\mathsf{locked}(window)$
  $\mathsf{dark}(window)$
  $\neg\mathsf{off}(light) \vee \neg\mathsf{locked}(door) \vee \mathsf{secure}(room)$
  or: $\mathsf{off}(light) \wedge \mathsf{locked}(door) \rightarrow \mathsf{secure}(room)$
  $\neg\mathsf{dark}(window) \vee \mathsf{off}(light)$
  or: $\mathsf{dark}(window) \rightarrow \mathsf{off}(light)$

- The Prolog program:

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


QUERIES

- Now, one can ask something:

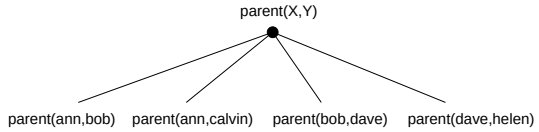
```
?- off(light).
Yes

?- secure(room).
No

?- locked(door).
No

?- locked(Something).
Something = window
Yes

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

  - Query variables are all existentially quantified

CONJUNCTIVE RULES

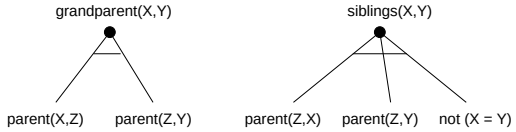
- A family tree:

```
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).
```

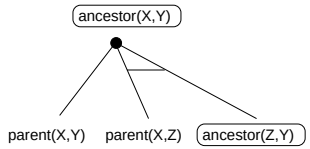


- All the rule variables are universally quantified

Disjunctive rules

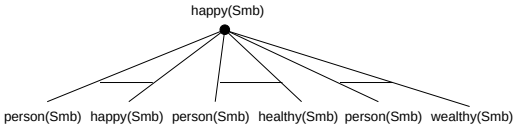
- Yet another family relation:

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



Predicate Calculus proofs

Application of inference rules: sound generation of new sentences from old

Proof = a sequence of inference rule applications
Can use inference rules as operators in a standard search algorithm.

Inference rules: Generalized resolution

$$\frac{\alpha \vee \beta', \qquad \neg\beta'' \vee \gamma, \qquad \exists\,\sigma:\ \beta = \beta'_\sigma \wedge \beta = \beta''_\sigma}{\alpha_\sigma \vee \gamma_\sigma}$$

and Generalized modus ponens

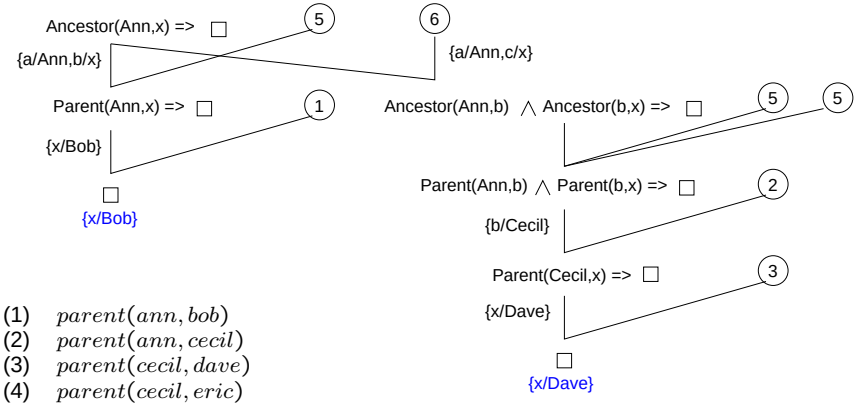
$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha'_1 \wedge \cdots \wedge \alpha'_n \rightarrow \beta, \qquad \exists\,\sigma:\ (\alpha_1)_\sigma = (\alpha'_1)_\sigma \wedge \cdots \wedge (\alpha_n)_\sigma = (\alpha'_n)_\sigma}{\beta_\sigma}$$

Proof by contradiction

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

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

Inference and Multiple Solutions

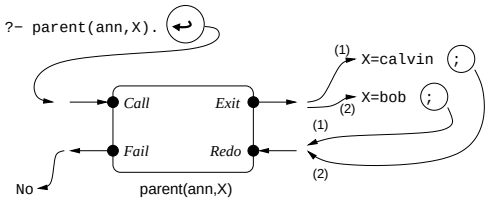


(1) $parent(ann, bob)$
(2) $parent(ann, cecil)$
(3) $parent(cecil, dave)$
(4) $parent(cecil, eric)$
(5) $parent(A, B) \rightarrow Ancestor(A, B)$
(6) $ancestor(A, B) \wedge ancestor(B, C) \rightarrow ancestor(A, C)$

SEARCHING THE KNOWLEDGE BASE

```
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
```



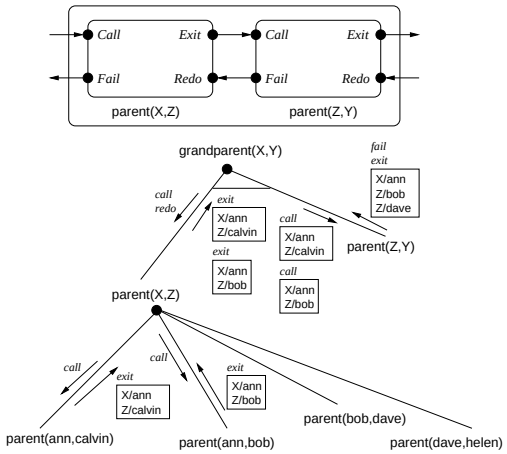
SEARCHING THE KNOWLEDGE BASE (CONT'D)

```
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
```

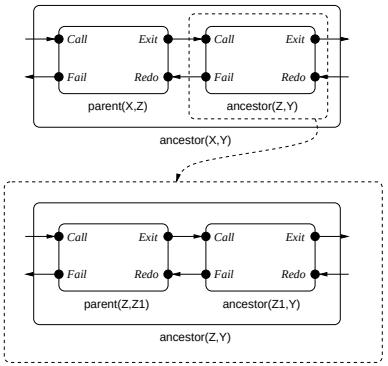

SEARCHING THE KNOWLEDGE BASE (CONT'D)



RECURSIVE PREDICATES

```
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 are no explicit assignments 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(X,Y).
tuple(1,2,3).   % tuple/3          X = 1
tuple(a,b,c).   % tuple/3          Y = 2 ;
tuple(a,b,c,d). % tuple/4          No
```


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 take **=/2** to mean assignment!

UNIFICATION AND STRUCTURES

- 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.

UNIFICATION AND STRUCTURES (EXAMPLE)

```
% 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).

% ...and eventually 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!

LIST PROCESSING

- Membership:

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

- What is the answer to the query `?- member(X,[1,2,3,4]).`?
  - Normally, both arguments of `member/2` are bound (we write henceforth `member(+E,+L)`). In fact, `member/2` also works as `member(-E,+L)`. The general specification is `member(?E,+L)`.
- 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 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]).`?

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

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_a(1,1).
fact_a(N,R) :- R is N*fact_a(N-1,R1).

13 ?- fact_a(1,X).
X = 1
Yes
14 ?- fact_a(2,X).
[WARNING: Arithmetic: 'fact_a/2' is
not a function]
 ^ Exception: (  8) _G185 is
2*fact_a(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_a(1,1).
fact_a(N,R) :- R is N*fact_a(N-1,R1).

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

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

?- fact(5,X).
X = 120
Yes
```


NUMBERS (CONT'D)

- 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?

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

?- fact(5,X).

X = 120 ;
```