CSE 307 Constraint Logic Programming

Lecture 2. From Datalog to Prolog: First-Order Logic Terms as Data Structure

François Fages

Inria Saclay Ile de France

Bachelor of Science de l'Ecole polytechnique

ECOLE POLYTECHNIQUE

First-Order Logic Terms as Data Structure

1. First-Order Term Structure $T(V, S_F)$
2. Equality in $T(V, S_F)$
3. Herbrand's Unification Algorithm
4. Unification Alg. for (finite and infinite) Rational Terms
5. List Processing in Prolog
6. Proving Recursive Programs Correct

1. Alphabet with Function Symbols

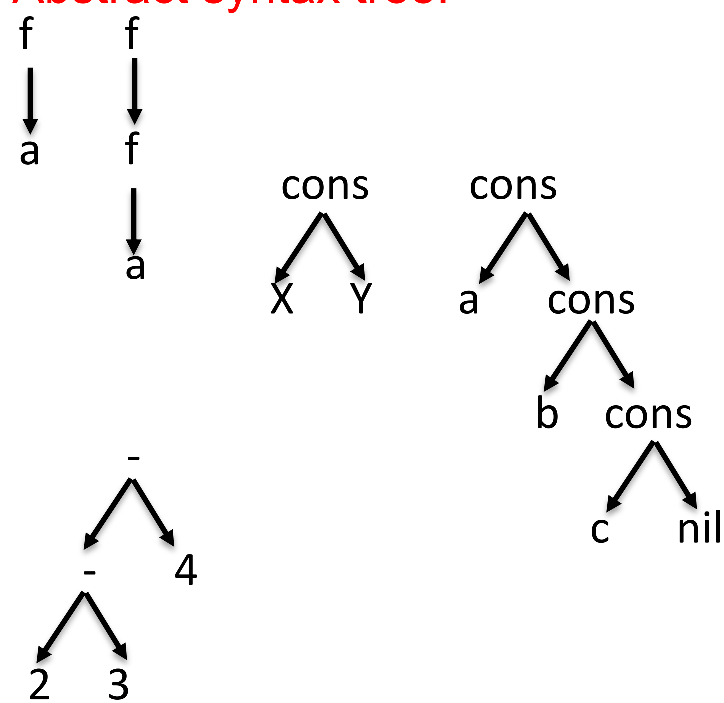
- Variables
  - Words starting with a upper case letter
  - `X, V, Start, End`, …
- Constants
  - Numbers and words starting with a lower case letter
  - `0, 1, -20, 3.14, 1e5, x, v, start, end`, …
- Function symbols
  - Words starting with lower case and **given an arity** (number of arguments)
  - Special characters `+/2, -/1, -/2, cons/2`, …
  - constants are function symbols of arity 0: `nil/0`,

First-Order Terms as Data Structure

A term is obtained by applying
- one function symbol of arity *n* to *n* arguments:
- variables, constants, and terms of smaller size !

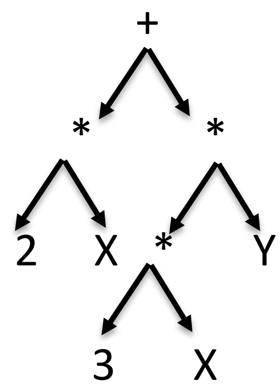
Examples:
- f(a)
- f(f(a))
- cons(X,Y)
  list constructor noted in Prolog
  in infix notation `[X | Y]`
- cons(a,cons(b,cons(c,nil)))
  `[a | [b | [c | [] ]]]`
  `[a,b,c]`
- 2-3-4

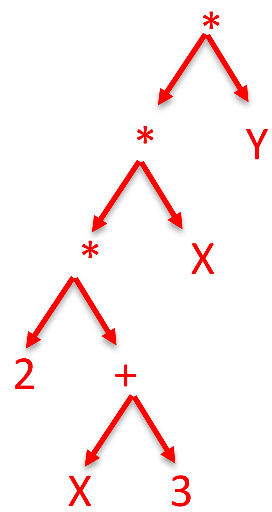
Abstract syntax tree:



Infix Operators: Precedence and Parentheses

2*X+3*X*Y





infix operators are declared in Prolog by `op(+precedence,+type,:name)`

For some reason in Prolog, the higher the “precedence” the lower the priority:

`op(500,yfx,['+','-']).` High “precedence” left parenthesized infix operator

`op(400,yfx,['*','/']).`

`op(200,fy,['+','-']).` Low “precedence” parenthesized prefix operator

Syntax tree for 2*X+3*X*Y with redefinition `op(300,yfx,['+','-']).`

Inductive Definition of First-Order Terms $T(V, S_F)$

Let V be a set of variables
Let S_F be a set of constant and function symbols given with their arity α

Inductive definition. The set of terms $T(V, S_F)$ noted T is the *least set*
1. containing variables, $V \subseteq T$
2. containing constants, $\{c\} \in T$ for all $c \in S_F$ with $\alpha(c)=0$
3. and closed by application of function symbols:
   $f(M_1, \dots, M_n) \in T$ for all $f \in S_F$ with $\alpha(f)=n$, $n > 0$ and all $M_1 \in T, \dots, M_n \in T$

Now, any function on terms can be defined recursively

Example. The size $s(t)$ of a term t (i.e. number of symbols) is defined by:
1. $s(v) = 1$ for $v \in V$
2. $s(c) = 1$ for $c \in S_F$ $\alpha(c) = 0$
3. $s\big(f(M_1, \dots, M_n)\big) = 1 + s(M_1) + \cdots + s(M_n)$ for any $f \in S_F$ $\alpha(f) = n \geq 1$

Exercises

Define by recursion the set $subterms(t)$ of all terms contained in a term t

$subterms(v) = \{v\}$ for any $v \in V$

$subterms(c) = \{c\}$ for any $c \in S_F\ \alpha(c) = 0$

$subterms\big(f(M_1, \dots, M_n)\big) = \{f(M_1, \dots, M_n)\} \cup subterms(M_1) \cup \cdots \cup subterms(M_n)$

Define by recursion the set $leaves(t)$ of all leaves subterms of t

$leaves(v) = \{v\}$ for any $v \in V$

$leaves(c) = \{c\}$ for any $c \in S_F\ \alpha(c) = 0$

$leaves\big(f(M_1, \dots, M_n)\big) = leaves(M_1) \cup \cdots \cup leaves(M_n)$ for any $f \in S_F\ \alpha(f) = n$

Define by recursion the set of variables $Var(t)$ in a term t

$var(v) = \{v\}$ for any $v \in V$

$var(c) = \emptyset$ for any $c \in S_F\ \alpha(c) = 0$

$var\big(f(M_1, \dots, M_n)\big) = var(M_1) \cup \cdots \cup var(M_n)$ for any $f \in S_F\ \alpha(f) = n$

Proofs by Structural Induction

Principle of induction on the integers :
A property is true for all integers if
1. it is true for 0
2. it is true for $n + 1$ supposing it true for n (or equivalently for all $0 \leq i \leq n$)

Axiom schema in Peano arithmetic FOL theory:
for any predicate p, $(p(0) \wedge \forall n \geq 0 \;\; p(n) \Rightarrow p(n+1)\,) \Rightarrow \forall n \geq 0 \; p(n)$

Principle of structural induction on terms $T(V, S_F)$:

A property is true for all terms if and only if
1. it is true for all variables
2. and true for all constants
3. and true for any complex term $f(M_1, \ldots, M_n)$ supposing it true for its subterms $M_1, \ldots, M_n$

Exercise

Prove that the set of leaves of a term is included in the set of its subterms.

We have

$leaves(v) = \{v\} = subterms(v)$ for any $v \in V$

$leaves(c) = \{c\} = subterms(v)$ for any $c \in S_F$ $\alpha(c) = 0$

$leaves\big(f(M_1, \dots, M_n)\big) = leaves(M_1) \cup \cdots \cup leaves(M_n)$ for any $f \in S_F$ $\alpha(c) \geq 1$

$\subseteq subterms(M_1) \cup \cdots \cup subterms(M_n)$ by recursion hypothesis

$\subseteq \{f(M_1, \dots, M_n\} \cup subterms(M_1) \cup \cdots \cup subterms(M_n)$

$\subseteq subterms(f(M_1, \dots, M_n))$ by definition of subterms

2. Equality on Terms

Syntactic equality
no interpretation of
function symbols

Inductive definition of equality on terms `==/2`
The equality relation between terms $s == t$
is the least relation such that

- $t == t$ for a variable (or a constant) t
- $f(s_1, \dots, s_n) == f(t_1, \dots, t_n)$ whenever $s_1 == t_1, \dots, s_n == t_n$

In Prolog, `=../2` decomposes a term in the list of head symbol and arguments
The equality predicate between terms `==/2` can defined in Prolog by `equal/2`

```
equal(X,Y) :- var(X), var(Y), X==Y.
equal(S,T) :- S=..[F|LS], T=..[F|LT], equal_list(LS,LT).

equal_list([],[]).
equal_list([S|LS],[T|LT]):- equal(S,T), equal_list(LS,LT).

? equal(1+1,2).
fail
```

Equation Solving on Terms

```
? 2*x+3*x*y=A*B.
false.

? 2*x+3*x*y=A+B.
A = 2*x,
B = 3*x*y.
```

```
?- [a,b,c]=[X|Y].
X = a,
Y = [b, c].

?- [a,b,c]=[X,Y|Z].
X = a,
Y = b,
Z = [c].

?- [a,b,c]=[X,Y,Z].
X = a,
Y = b,
Z = c.
```

```
?- [a,b,c]=[X,Y,Z|L].
X = a,
Y = b,
Z = c,
L = [].

?- [a]=[X|Y].
X = a,
Y = [].
```

3. Unification Algorithm [Herbrand 1928] as a Rewriting System of Equations until Solved Form

Let Γ be a system of term equations, initially the two terms to unifiy $s = t$

1. $\mathrm{x} = \mathrm{x} \wedge \Gamma \longrightarrow \Gamma$ if x is a variable

2. $\mathrm{x} = \mathrm{t} \wedge \Gamma \longrightarrow \Gamma[\mathrm{x} \leftarrow \mathrm{t}]$ if x is a variable and $x \notin Var(t)$ (same rule for t = x)

3. $\mathrm{x} = \mathrm{t} \wedge \Gamma \longrightarrow$ false if x is a variable, $x \neq t$ and $x \in Var(t)$ (same rule for t = x)

4. $\mathrm{s} = \mathrm{t} \wedge \Gamma \longrightarrow \mathrm{s}_1 = \mathrm{t}_1 \wedge \cdots \wedge s_n = t_n \wedge \Gamma$ if $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$

5. $\mathrm{s} = \mathrm{t} \wedge \Gamma \longrightarrow$ false if $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \neq g$

Example:

$\boldsymbol{f}(\boldsymbol{g}(\boldsymbol{X}), \boldsymbol{X}) = \boldsymbol{f}(\boldsymbol{Y}, \boldsymbol{a}) \rightarrow_4 g(X) = Y, \boldsymbol{X} = \boldsymbol{a} \rightarrow_2 g(a) = Y \rightarrow_2 \emptyset$

$\boldsymbol{f}(\boldsymbol{g}(\boldsymbol{X}), \boldsymbol{Y}) = \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{a}) \rightarrow_4 \boldsymbol{g}(\boldsymbol{X}) = \boldsymbol{X}, Y = A \rightarrow_3 false$

Solved Form and Soundness

Lemma (solved form). If Γ is irreducible then either $\Gamma = \emptyset$ or Γ=false

Proof. The system rewrites equations.

If Γ contains an equation with a variable, it is eliminated by rules 1, 2, or 3

If Γ contains an equation between non-variable terms, it is eliminated by 4, 5

Therefore, if Γ is irreducible it is either empty or false.

Proposition (soundness). Let $\Gamma \longrightarrow \Gamma$'.

Γ is satisfiable if and only Γ' is satisfiable.

Proof.

We check that this is indeed the case for each of the 5 rewriting rules.

In rule 3, if $x = t$ with $x \in Var(t)$ then the system is false

because the equation $x = f(x)$ has no solution in finite terms $T(V, S_F)$.

The other cases are trivial by the inductive definition of equality.

Termination

Proposition (termination). On any finite system Γ of term equations, the rewriting rules terminate.

Proof. Find a complexity measure of Γ that strictly decreases at each rewriting.

Size s(Γ) ?

no the size of the subterms may grow after substitution (rule 1)

Number of variables $\mathrm{card}(\mathrm{var}(\Gamma))$?

no the subterms have the same number of variables (rule 3)

Take as complexity measure the couple ($\mathrm{card}(\mathrm{var}(\Gamma))$, s(Γ))

ordered by lexicographic ordering $(v', s') < (v, s)$ iff $v' < v$ or ($v' = v$ and $s' < s$)

- After rule 1 (relexivity) we have $v' \leq v$ and $s' < s$
- Rule 3 and 5 (false) are trivial $v' = 0$ and $s' = 0$
- After rule 2 (substitution of a variable) we have $v' < v$
- After rule 4 (decomposition) we have $v' = v$ and $s' < s$

Completeness

Proposition (completeness). If Γ is unsatisfiable then $\Gamma \longrightarrow^* \text{false}$.

Proof.

By termination, let us consider the last rewriting step of $\Gamma \rightarrow^* \Gamma' \rightarrow \Gamma''$

where Γ'' is irreducible.

By the solved form lemma, Γ'' is either false or empty.

By the soundness lemma, Γ' and Γ'' are unsatisfiable,

hence Γ''=false not empty (i.e. true).

Therefore that rewriting system allows us to decide the satisfiability of a set of equality constraints over FOL terms,

but in its current form, it looses the substitutions of the variables.

Modification of the rewriting rules to keep track of the substitutions ?

Rewriting System Preserving the Substitutions

1. $x = x \wedge \Gamma \longrightarrow \Gamma$ if x is a variable
2. $x = t \wedge \Gamma \longrightarrow \Gamma[x \leftarrow t]$ if x is a variable and $x \notin Var(t)$ (same rule for $t = x$)
3. $x = t \wedge \Gamma \longrightarrow$ false if x is a variable, $x \neq t$ and $x \in Var(t)$ (same rule for $t = x$)
4. $s = t \wedge \Gamma \longrightarrow s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge \Gamma$ if $s = f(s_1, \dots, s_n)$, $t = f(t_1, \dots, t_n)$
5. $s = t \wedge \Gamma \longrightarrow$ false if $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$, $f \neq g$

Which rewriting rules must be modified ?

2. $x = t \wedge \Gamma \longrightarrow x = t \wedge \Gamma[x \leftarrow t]$ if x is a variable, $x \notin Var(t)$, $x \in Var(\Gamma)$ (idem $t = x$)

Complexity measure for termination ?

Couple (number of variables with at least 2 occurrences, size of the system)

Solved form lemma ?

$\Gamma = \emptyset$ or Γ=false or Γ is composed of equations of the form variable=term only where the variable has that single occurrence in the system

All solutions are expressed by a single conjunction of var=term equalities

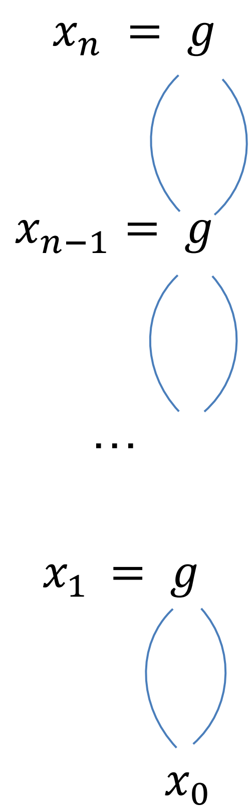
Herbrand's Unification Algorithm in Prolog

In Prolog using conditional goals `(condition -> thengoal ; elsegoal)`

```
unif(X,Y) :-
    var(X) -> (var(Y) -> X = Y; not_occurs(X,Y),X=Y);
    var(Y) -> not_occurs(Y,X),Y=X;
    X=..[F|LX],
    Y=..[F|LY],
    unify_args(LX,LY).

unify_args([],[]).
unify_args([X|LX],[Y|LY]):- unif(X,Y), unify_args(LX,LY).

not_occurs(Var,Term) :-
    Var==Term -> fail;
    var(Term) -> true;
    Term=..[_|L],
    not_occurs_rec(L,Var).

not_occurs_rec([],_).
not_occurs_rec([T|L],Var):-
    not_occurs(Var,T),
    not_occurs_rec(L,Var).
```

Time Complexity of Herbrand's Unification Alg.

The terms $f(x_1, \dots, x_n, x_n) = (f(g(x_0, x_0), \dots, g(x_{n-1}, x_{n-1}), x_n)$ are unifiable with a substitution of exponential size in n creating $2^{n+1} - 1$ subterms for x_n and taking exponential time for checking the last argument by decomposition



Disequality constraint `dif/2`
(first constraint introduced in Prolog II by A. Colmerauer in 1982)

The dif/2 predicate is a *constraint* true if and only if *A* and *B* are different terms.

- If *A* and *B* can never unify, dif/2 succeeds deterministically.
- If *A* and *B* are identical, it fails immediately.
- If *A* and *B* can unify, the goal is delayed to prevent *A*=*B* by unification

More declarative than built-in non-logical (non-monotonic) predicates

  - \== /2 negation by failure on equality (terms are not equal)
  - \= /2 negation by failure on unification (terms do not unify)
  - Non-monotonic predicates: may change from false to true when adding constraints

?= /2 true if the equality of the terms is independent from variables' instantiation

Defined in SWI Prolog autoloaded library(dif) using **goal section predicate** `when/2`

```
dif(X, Y) :- when(?=(X,Y), X \== Y).
```

4. By default SWI-Prolog accepts Infinite Terms

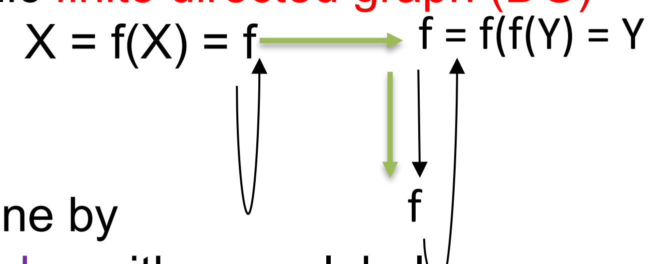
```
?- f(g(X), X)=f(Y,a).
X = a,
Y = g(a).

?- f(g(X), Y)=f(X, a).
X = g(X),
Y = a.

?- X=f(X).
X = f(X).

?- f(X)=f(f(X)).
X = f(X).
```

Solving Equations in Rational Infinite Terms

A finite term can be represented by a Directed Acyclic Graph (DAG) with nodes

- labelled by function symbols
- with as many successors as the arity of their label

Def. A rational term is an infinite term with a finite set of (rational) subterms
Can be represented by a possibly cyclic finite directed graph (DG)



Unification of rational terms can be done by

- merging equivalence classes of nodes with same label or one variable
- using Tarjan's set-union-find algorithm
  - in $O(n^2)$ with simple pointer path following
  - $O(n.\log n)$ if path compression or pointing from the smaller to larger class
  - $O(n.\alpha(n))$ if both: quasi-linear (α is the inverse of Ackermann function)

Unify f(X,g(X)) = f(g(Y),Y) ?

→ Checking acyclicity gives a quasi linear unification algorithm for finite terms

5. Lists in Prolog: `member/2`

List constructor `[|]/2` with list constant `[]/0`

Definition of `member/2` by recursion on the second argument (the list)

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

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.
```

```
?- member(a,L).
L = [a|_2814] ;
L = [_2812, a|_2820] ;
L = [_2812, _2818, a|_2826] ;
L = [_2812, _2818, _2824, a|_2832]
```

Lists in Prolog: append/2

```
append([],L,L).
append([X|L],L2,[X|L3]):-append(L,L2,L3).

?- append([a,b],[c,d],L).
L = [a, b, c, d].

?- append(X,Y,L).
X = [], Y = L ;
X = [_2888], L = [_2888|Y] ;
X = [_2888, _2900], L = [_2888, _2900|Y] ;
X = [_2888, _2900, _2912], L = [_2888, _2900, _2912|Y]

?- append(X,X,X).
X = [] ;
ERROR: Out of global stack
```

append([a1,…,an],L,R)
time complexity?
O(n)

Lists in Prolog: `reverse/2`

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

?- reverse([a,b,c,d],L).
L = [d, c, b, a].

?- reverse(L,[a,b,c,d]).
L = [d, c, b, a] .

?- reverse(X,Y).
X = Y, Y = [] ;
X = Y, Y = [_2848] ;
X = [_2848, _2854], Y = [_2854, _2848] ;
X = [_2848, _2866, _2854], Y = [_2854, _2866, _2848] ;

?- append(X,Y,L),reverse(L,L).
X = Y, Y = L, L = [] ;
X = [], Y = L, L = [_3290] ;
X = [], Y = L, L = [_3290, _3290] ;
X = [], Y = L, L = [_3290, _3302, _3290]
```

reverse([a1,…,an],R)
time complexity?
$O(n^2)$

Linear `reverse/3` using an Accumulator

```
reverse_linear(L,R):-reverse(L,[],R).

reverse([],R,R).
reverse([X|L],K,R):-reverse(L,[X|K],R).


? reverse_linear([a,b,c], L).
L=[c,b,a]
```

rev([a1,…,an],R)
time complexity?
O(n)

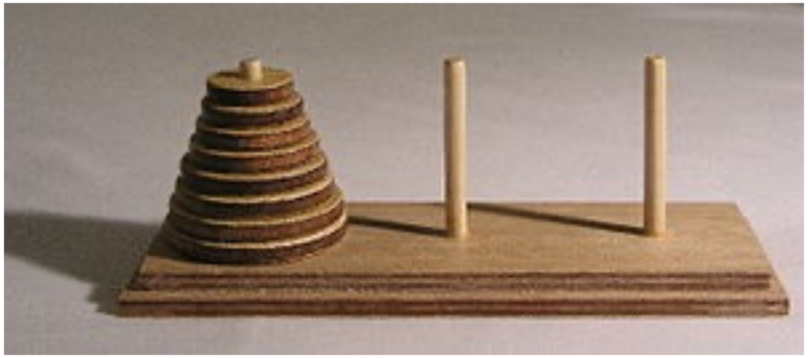
Grammars in Prolog
(formal grammars A. Colmerauer 1969 at the origin of Prolog)

```
sentence(L):-nounphrase(L1), verbphrase(L2), append(L1,L2,L).
nounphrase(L):- determiner(L1), noun(L2), append(L1,L2,L).
nounphrase(L):- noun(L).
verbphrase(L):- verb(L).
verbphrase(L):- verb(L1), nounphrase(L2), append(L1,L2,L).
verb([eats]).
determiner([the]).
noun([monkey]).
noun([banana]).

| ?- sentence([the,monkey,eats]).
yes
| ?- sentence([the,eats]).
fail
| ?- sentence(L).
L = [the,monkey,eats] ? ;
L = [the,monkey,eats,the,monkey] ? ;
L = [the,monkey,eats,the,banana] ? ;
L = [the,monkey,eats,monkey] ?
```

6. Proving Recursive Programs Correct

The Hanoi Towers puzzle:

- how to transfer N disks from peg 1 to peg 3 using peg 2
- one disk at a time
- without never putting a disk of larger size on top of a smaller disk



Recursive Programming is an Act of Belief !

Believe in your recursive hypotheses! (it's necessary to reuse them…)

Try a **recursion hypothesis** on `hanoi(N,X,Y,Z)`
Precondition: the N top disks of X are smaller than the disks on Y and Z
Postcondition: the N top disks of X have been correctly moved on top of peg Z

So, let us try to recurse on N, assuming that the precondition is satisfied

```
hanoi(N,X,_,Z) :-
  N #= 1,
  format('Move top disk from ~w to ~w~n', [X,Z]).
```

postcondition is satisfied since the top disk of X was smaller than Z top disk

```
hanoi(N,X,Y,Z) :-
  N #> 1,
  N1 #= N-1,
  …
```

Recursive Programming is an Act of Belief !

precondition 1: the N top disks of X are smaller than the disks on Y and Z

```
hanoi(N,X,Y,Z) :-
N #> 1,
N1 #= N-1,
```

precondition 2 is satisfied by precondition 1 since N1<N

```
hanoi(N1,X,Z,Y),
```

postcondition 2 says N1 top X disks correctly moved on top of Y

```
format('Move top disk from ~w to ~w~n', [X,Z]),
```

plus now the Nth former top X disk correctly moved on top of Z

precondition 3 is satisfied: the N1 disks on top of Y are smaller than X and Z

```
hanoi(N1,Y,X,Z).
```

postcondition 3 says the N1 former X disks have been correctly moved on Z

postcondition 1: the N top disks of X have been correctly moved on top of Z

Hanoi Towers Program Execution

```
hanoi(N,X,_,Z) :-
  N #= 1,
  format('Move top disk from ~w to ~w~n', [X,Z]).

hanoi(N,X,Y,Z) :-
  N #> 1,
  N1 #= N-1,
  hanoi(N1,X,Z,Y),
  format('Move top disk from ~w to ~w~n', [X,Z]),
  hanoi(N1,Y,X,Z).

?- hanoi(3,left,middle,right).
Move top disk from left to right
Move top disk from left to middle
Move top disk from right to middle
Move top disk from left to right
Move top disk from middle to left
Move top disk from middle to right
Move top disk from left to right
```

Next TP: Symbolic Computation in Prolog

- Symbolic differentiation in Prolog
  - Arithmetic expressions as Prolog terms
  - Predicate for automatic differentiation

- List processing with arithmetic constraints

- Pathways in cyclic graphs
  - Non-looping transitive closure program
  - GPS navigation with travel time

- Hanoi tower puzzle visualization using lists

Next Lecture: Constraint Logic Programming

How to generalize Prolog over first-order terms (the "Herbrand domain”)
to any “domain of discourse” X ?

keeping the same resolution principle with Horn clauses and constraints =
Constraint Logic Programming CLP(X)

From equality constraints over First-order Terms
Prolog = CLP(($T(V, S_F)$, $=/2$))
Prolog II = CLP($T(V, S_F)$, $=/2$, `dif/2`);)

to any constraints over an arbitrary algebraic structure X

- Real numbers with linear equations and inequations
  Prolog III = CLP($\mathcal{R}, \{0,1,+,*\}, \{=,\leq\}$) in Lecture 3
- Integer arithmetic and finite domain constraints CLP(FD) in Lecture 4
- Graph constraints
- Set constraints ...