Computational Logic

Pure (Declarative) Logic Programs

Pure Logic Programs (Overview)

- Programs that only make use of unification.
- They are completely logical: the set of computed answers is exactly the set of logical consequences.
  - *Computed answers*: all calls that compute successfully
- Allow to program declaratively: declare the problem (specifications as programs)
- They have full computational power.

1. Database programming.
2. Arithmetics.
3. Data structure manipulation.
4. Recursive programming.

Database Programming

- A Logic Database is a set of facts and rules (i.e., a logic program):

```
father_of(john,peter) <- .
father_of(john,mary) <- .
father_of(peter,michael) <- 

mother_of(mary, david) <- .
```

```
grandfather_of(L,M) <- father_of(L,N),
                       father_of(N,M).
grandfather_of(X,Y) <- father_of(X,Z),
                       mother_of(Z,Y).
```

- Given such database, a logic programming system can answer questions (queries) such as:

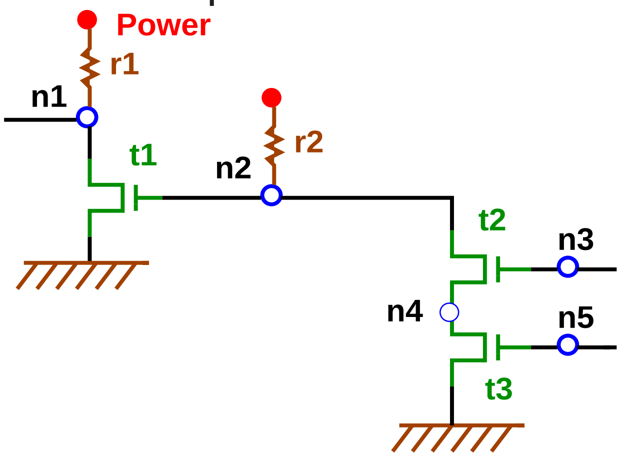
`<- father_of(john, peter).`
Answer: *Yes*
`<- father_of(john, david).`
Answer: *No*
`<- father_of(john, X).`
Answer: $\{X = peter\}$
Answer: $\{X = mary\}$

`<- grandfather_of(X, michael).`
Answer: $\{X = john\}$
`<- grandfather_of(X, Y).`
Answer: $\{X = john, Y = michael\}$
Answer: $\{X = john, Y = david\}$
`<- grandfather_of(X, X).`
Answer: *No*

- Rules for `grandmother_of(X, Y)`?

Database Programming (Contd.)

- Another example:




```
resistor(power,n1) <- .
resistor(power,n2) <- .

transistor(n2,ground,n1) <- .
transistor(n3,n4,n2) <- .
transistor(n5,ground,n4) <- .
```

```
inverter(Input,Output) <-
   transistor(Input,ground,Output), resistor(power,Output).
nand_gate(Input1,Input2,Output) <-
   transistor(Input1,X,Output), transistor(Input2,ground,X),
   resistor(power,Output).
and_gate(Input1,Input2,Output) <-
   nand_gate(Input1,Input2,X), inverter(X, Output).
```

- Query `and_gate(In1,In2,Out)` has solution: {`In1=n3, In2=n5, Out=n1`}

Structured Data and Data Abstraction

- The circuit example revisited:

```
resistor(r1,power,n1) <- .      transistor(t1,n2,ground,n1) <- .
resistor(r2,power,n2) <- .      transistor(t2,n3,n4,n2) <- .
                               transistor(t3,n5,ground,n4) <- .

inverter(inv(T,R),Input,Output) <-
   transistor(T,Input,ground,Output), resistor(R,power,Output).

nand_gate(nand(T1,T2,R),Input1,Input2,Output) <-
   transistor(T1,Input1,X,Output), transistor(T2,Input2,ground,X),
   resistor(R,power,Output).

and_gate(and(N,I),Input1,Input2,Output) <-
   nand_gate(N,Input1,Input2,X), inverter(I,X,Output).
```

- The query `<- and_gate(G,In1,In2,Out).`
  has solution: `{G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1}`

Logic Programs and the Relational DB Model

Traditional →	*Codd's Relational Model*	
File	Relation	Table
Record	Tuple	Row
Field	Attribute	Column

- *Example*:

Name	Age	Sex
Brown	20	M
Jones	21	F
Smith	36	M

Person

Name	Town	Years
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

Lived–in

- The order of the rows is immaterial.
- (Duplicate rows are not allowed)

Logic Programs and the Relational DB Model (Contd.)

Relational Database		*Logic Programming*
Relation Name	$\rightarrow$	Predicate symbol
Relation	$\rightarrow$	Procedure consisting of ground facts (facts without variables)
Tuple	$\rightarrow$	Ground fact
Attribute	$\rightarrow$	Argument of predicate

- *Example*:

```
person(brown,20,male) <- .
person(jones,21,female) <- .
person(smith,36,male) <- .
```

Name	Age	Sex
Brown	20	M
Jones	21	F
Smith	36	M

- *Example*:

```
lived_in(brown,london,15) <- .
lived_in(brown,york,5) <- .
lived_in(jones,paris,21) <- .
lived_in(smith,brussels,15) <- .
lived_in(smith,santander,5) <- .
```

Name	Town	Years
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

Logic Programs and the Relational DB Model (Contd.)

- The operations of the relational model are easily implemented as rules.
  - *Union*:

    $\texttt{r\_union\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n).$

    $\texttt{r\_union\_s}(X_1,\ldots,X_n) \leftarrow \texttt{s}(X_1,\ldots,X_n).$
  - *Set Difference*:

    $\texttt{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n),\ \texttt{not}\ \texttt{s}(X_1,\ldots,X_n).$

    $\texttt{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow \texttt{s}(X_1,\ldots,X_n),\ \texttt{not}\ \texttt{r}(X_1,\ldots,X_n).$

    (we postpone the discussion on *negation* until later.)
  - *Cartesian Product*:

    $\texttt{r\_X\_s}(X_1,\ldots,X_m,X_{m+1},\ldots,X_{m+n}) \leftarrow \texttt{r}(X_1,\ldots,X_m),\texttt{s}(X_{m+1},\ldots,X_{m+n}).$
  - *Projection*:

    $\texttt{r13}(X_1,X_3) \leftarrow \texttt{r}(X_1,X_2,X_3).$
  - *Selection*:

    $\texttt{r\_selected}(X_1,X_2,X_3) \leftarrow \texttt{r}(X_1,X_2,X_3),\leq(X_2,X_3).$

    (see later for definition of $\leq/2$)

Logic Programs and the Relational DB Model (Contd.)

- Derived operations – some can be expressed more directly in LP:
  - ◇ Intersection:

    $\texttt{r\_meet\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n),\ \texttt{s}(X_1,\ldots,X_n).$
  - ◇ Join:

    $\texttt{r\_joinX2\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,X_2,X_3,\ldots,X_n),\ \texttt{s}(X_1',X_2,X_3',\ldots,X_n').$
- Duplicates an issue: see “setof” later in Prolog.

Deductive Databases

- The subject of “deductive databases” uses these ideas to develop *logic-based databases*.
  - ◇ Often syntactic restrictions (a subset of definite programs) used (e.g. “Datalog” – no functors, no existential variables).
  - ◇ Variations of a “bottom-up” execution strategy used: Use the T_p operator (explained in the theory part) to compute the model, restrict to the query.

Recursive Programming

- Example: ancestors.

```
parent(X,Y) <- father(X,Y).
parent(X,Y) <- mother(X,Y).

ancestor(X,Y) <- parent(X,Y).
ancestor(X,Y) <- parent(X,Z), parent(Z,Y).
ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,Y).
ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,K), parent(K,Y).
...
```

- Defining ancestor recursively:

```
parent(X,Y) <- father(X,Y).
parent(X,Y) <- mother(X,Y).

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

- Exercise: define “related”, “cousin”, “same generation”, etc.

Types

- *Type:* a (possibly infinite) set of terms.
- *Type definition:* A program defining a type.
- *Example:* Weekday:
  - Set of terms to represent: `Monday`, `Tuesday`, `Wednesday`, . . .
  - Type definition:

```
is_weekday('Monday') <- .
is_weekday('Tuesday') <- .  ...
```

- *Example:* Date (weekday * day in the month):
  - Set of terms to represent: `date('Monday',23)`, `date(Tuesday,24)`, . . .
  - Type definition:

```
is_date(date(W,D)) <- is_weekday(W), is_day_of_month(D).
is_day_of_month(1) <- .
is_day_of_month(2) <- .
...
is_day_of_month(31) <- .
```

Recursive Programming: Recursive Types

- *Recursive types*: defined by recursive logic programs.
- *Example:* natural numbers (simplest recursive data type):
  - Set of terms to represent: `0`, `s(0)`, `s(s(0))`, ...
  - Type definition:

```
nat(0) <- .
nat(s(X)) <- nat(X).
```

  A *minimal recursive predicate*:
  one unit clause and one recursive clause (with a single body literal).

- We can reason about *complexity*, for a given *class of queries* ("*mode*").
  E.g., for mode `nat(`*ground*`)` complexity is *linear* in size of number.
- *Example:* integers:
  - Set of terms to represent: `0`, `s(0)`, `-s(0)`,...
  - Type definition:

```
integer( X) <- nat(X).
integer(-X) <- nat(X).
```

Recursive Programming: Arithmetic

- Defining the natural order ($\leq$) of natural numbers:

```
less_or_equal(0,X) <- nat(X).
less_or_equal(s(X),s(Y)) <- less_or_equal(X,Y).
```

- Multiple uses: `less_or_equal(s(0),s(s(0)))`, `less_or_equal(X,0)`,...
- Multiple solutions: `less_or_equal(X,s(0))`, `less_or_equal(s(s(0)),Y)`, etc.
- Addition:

```
plus(0,X,X) <- nat(X).
plus(s(X),Y,s(Z)) <- plus(X,Y,Z).
```

- Multiple uses: `plus(s(s(0)),s(0),Z)`, `plus(s(s(0)),Y,s(0))`
- Multiple solutions: `plus(X,Y,s(s(s(0))))`, etc.

Recursive Programming: Arithmetic (Contd.)

- Another possible definition of addition:

```
plus(X,0,X) <- nat(X).
plus(X,s(Y),s(Z)) <- plus(X,Y,Z).
```

- The meaning of `plus` is the same if both definitions are combined.
- Not recommended: several proof trees for the same query $\rightarrow$ not efficient, not concise. We look for minimal axiomatizations.
- The art of logic programming: finding compact and computationally efficient formulations!

- Try to define: `times(X,Y,Z)` (Z = X*Y), `exp(N,X,Y)` (Y = X^N), `factorial(N,F)` (F = N!), `minimum(N1,N2,Min)`, ...

Recursive Programming: Arithmetic (Contd.)

- Definition of `mod(X,Y,Z)`
  "`Z` is the remainder from dividing `X` by `Y`"
  ($\exists$ Q s.t. X = Y*Q + Z and Z $<$ Y):

```
mod(X,Y,Z) <- less(Z, Y), times(Y,Q,W), plus(W,Z,X).

less(0,s(X)) <- nat(X).
less(s(X),s(Y)) <- less(X,Y).
```

- Another possible definition:

```
mod(X,Y,X) <- less(X, Y).
mod(X,Y,Z) <- plus(X1,Y,X), mod(X1,Y,Z).
```

- The second is much more efficient than the first one
  (compare the size of the proof trees).

Recursive Programming: Arithmetic/Functions

- The Ackermann function:

```
ackermann(0,N) = N+1
ackermann(M,0) = ackermann(M-1,1)
ackermann(M,N) = ackermann(M-1,ackermann(M,N-1))
```

- In Peano arithmetic:

```
ackermann(0,N)         = s(N)
ackermann(s(M),0)      = ackermann(M,s(0))
ackermann(s(M),s(N)) = ackermann(M,ackermann(s(M),N))
```

- Can be defined as:

```
ackermann(0,N,s(N))        <- .
ackermann(s(M),0,Val)      <-  ackermann(M,s(0),Val).
ackermann(s(M),s(N),Val) <-  ackermann(s(M),N,Val1),
                               ackermann(M,Val1,Val).
```

- In general, *functions* can be coded as a predicate with one more argument, which represents the output (and additional syntactic sugar often available).
- Syntactic support available (see, e.g., the Ciao *functions* package).

Recursive Programming: Lists

- Type definition (no syntactic sugar):

```
list([]) <- .
list(.(X,Y)) <- list(Y).
```

- Type definition (with syntactic sugar):

```
list([]) <- .
list([X|Y]) <- list(Y).
```

- List concatenation (e.g., a list traversal):

```
append([],Ys,Ys) <- .
append([X|Y],Ys,[X|Zs]) <- append(Xs,Ys,Zs).
```

Recursive Programming: Binary Trees

- Represented by a ternary functor `tree(Element,Left,Right)`.
- Empty tree represented by `void`.
- Definition:

```
binary_tree(void) <- .
binary_tree(tree(Element,Left,Right)) <-
    binary_tree(Left),
    binary_tree(Right).
```

- Defining `tree_member(Element,Tree)`:

```
tree_member(X,tree(X,Left,Right)) <- .
tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Right).
```

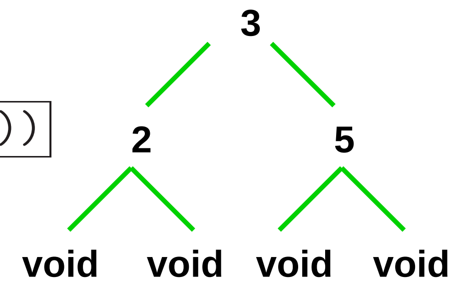
Recursive Programming: Binary Trees (Contd.)

- Defining `pre_order(Tree,Order)`:

```
pre_order(void,[]) <- .
pre_order(tree(X,Left,Right),Order) <-
     pre_order(Left,OrderLeft),
     pre_order(Right,OrderRight),
     append([X|OrderLeft],OrderRight,Order).
```

- Define `in_order(Tree,Order)`, `post_order(Tree,Order)`.

Creating a Binary Tree in Pascal and LP



- In Prolog:

```
T = tree(3, tree(2,void,void), tree(5,void,void))
```

- In Pascal:

```
type tree = ^treerec;
        treerec = record
                data : integer;
                left : tree;
                right: tree;
        end;

var t : tree;
```

```
...
new(t);
new(t^left);
new(t^right);
t^left^left := nil;
t^left^right := nil;
t^right^left := nil;
t^right^right := nil;
t^data := 3;
t^left^data := 2;
t^right^data := 5;
...
```

Polymorphism

- Note that the two definitions of `member/2` can be used *simultaneously*:

```
lt_member(X,[X|Y]) <- list(Y).
lt_member(X,[_|T]) <- lt_member(X,T).

lt_member(X,tree(X,L,R)) <- binary_tree(L), binary_tree(R).
lt_member(X,tree(Y,L,R)) <- binary_tree(R), lt_member(X,L).
lt_member(X,tree(Y,L,R)) <- binary_tree(L), lt_member(X,R).
```

  Lists only unify with the first two clauses, trees with clauses 3–5!

- `<- lt_member(X,[b,a,c]).`
  `X = b ; X = a ; X = c`

- `<- lt_member(X,tree(b,tree(a,void,void),tree(c,void,void))).`
  `X = b ; X = a ; X = c`

- Also, try (somewat surprising): `<- lt_member(M,T).`

Recursive Programming: Manipulating Symbolic Expressions

- Recognizing polynomials in some term X:
  - ◇ X is a polynomial in X
  - ◇ a constant is a polynomial in X
  - ◇ sums, differences and products of polynomials in X are polynomials
  - ◇ also polynomials raised to the power of a natural number and the quotient of a polynomial by a constant

```
polynomial(X,X) <- .
polynomial(Term,X)         <- pconstant(Term).
polynomial(Term1+Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1-Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1*Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1/Term2,X) <- polynomial(Term1,X), pconstant(Term2).
polynomial(Term1^N,X)     <- polynomial(Term1,X), nat(N).
```

Recursive Programming: Manipulating Symb. Expressions (Contd.)

- Symbolic differentiation: `deriv(Expression, X, DifferentiatedExpression)`

```
deriv(X,X,s(0)) <- .
deriv(C,X,0)                          <- pconstant(C).
deriv(U+V,X,DU+DV)                    <- deriv(U,X,DU), deriv(V,X,DV).
deriv(U-V,X,DU-DV)                    <- deriv(U,X,DU), deriv(V,X,DV).
deriv(U*V,X,DU*V+U*DV)                <- deriv(U,X,DU), deriv(V,X,DV).
deriv(U/V,X,(DU*V-U*DV)/V^s(s(0)))    <- deriv(U,X,DU), deriv(V,X,DV).
deriv(U^s(N),X,s(N)*U^N*DU)           <- deriv(U,X,DU), nat(N).
deriv(log(U),X,DU/U)                  <- deriv(U,X,DU).
...
```

- `<- deriv(s(s(s(0)))*x+s(s(0)),x,Y).`
- A simplification step can be added.

Recursive Programming: Graphs

- Usual: make use of another data structure, e.g., lists
  - ◇ Graphs as lists of edges.
- Alternative: make use of Prolog's program database
  - ◇ Declare the graph using facts in the program.

```
edge(a,b) <- .
edge(b,c) <- .
edge(c,a) <- .          edge(d,a) <- .
```

- Paths in a graph: `path(X,Y)` iff there is a path in the graph from node `X` to node `Y`.

```
path(A,B) <- edge(A,B).
path(A,B) <- edge(A,X), path(X,B).
```

- Circuit: a closed path. `circuit` iff there is a path in the graph from a node to itself.

```
circuit <- path(A,A).
```

Recursive Programming: Graphs (Exercises)

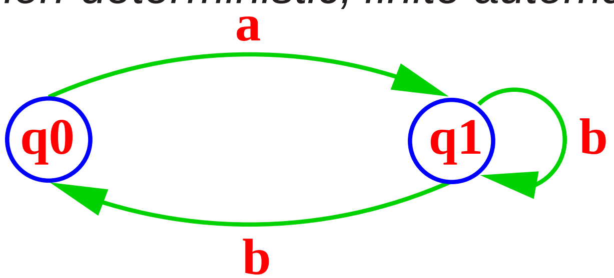
- Modify `circuit`/0 so that it gives the circuit.
  (You have to modify also `path`/2)
- Propose a solution for handling several graphs in our representation.
- Propose a suitable representation of graphs as data structures.
- Define the previous predicates for your representation.

- Consider unconnected graphs (there is a subset of nodes not connected in any way to the rest) versus connected graphs.
- Consider directed versus undirected graphs.

- Try `path(a,d)`. Solve the problem.

Recursive Programming: Automata (Graphs)

- Recognizing the sequence of characters accepted by the following *non-deterministic, finite automaton* (NDFA):



where **q0** is both the *initial* and the *final* state.

- Strings are represented as lists of constants (e.g., `[a,b,b]`).
- Program:

```
initial(q0) <- .       delta(q0,a,q1) <- .
                       delta(q1,b,q0) <- .
final(q0) <- .         delta(q1,b,q1) <- .

accept(S) <- initial(Q), accept_from(S,Q).

accept_from([],Q)     <- final(Q).
accept_from([X|Xs],Q) <- delta(Q,X,NewQ), accept_from(Xs,NewQ).
```

Recursive Programming: Automata (Graphs) (Contd.)

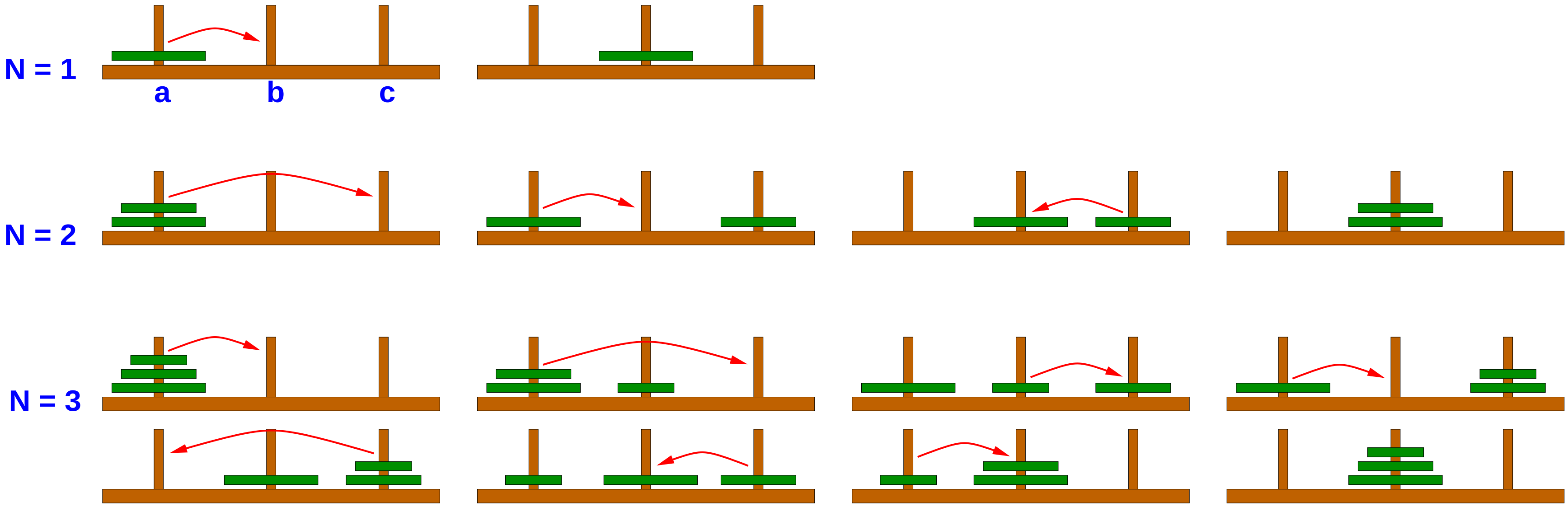
- A *nondeterministic, stack, finite automaton* (NDSFA):

```
accept(S) <- initial(Q), accept_from(S,Q,[]).

accept_from([],Q,[])     <- final(Q).
accept_from([X|Xs],Q,S) <- delta(Q,X,S,NewQ,NewS),
                           accept_from(Xs,NewQ,NewS).

initial(q0) <- .
final(q1) <- .

delta(q0,X,Xs,q0,[X|Xs]) <- .
delta(q0,X,Xs,q1,[X|Xs]) <- .
delta(q0,X,Xs,q1,Xs) <- .
delta(q1,X,[X|Xs],q1,Xs) <- .
```

- What sequence does it recognize?

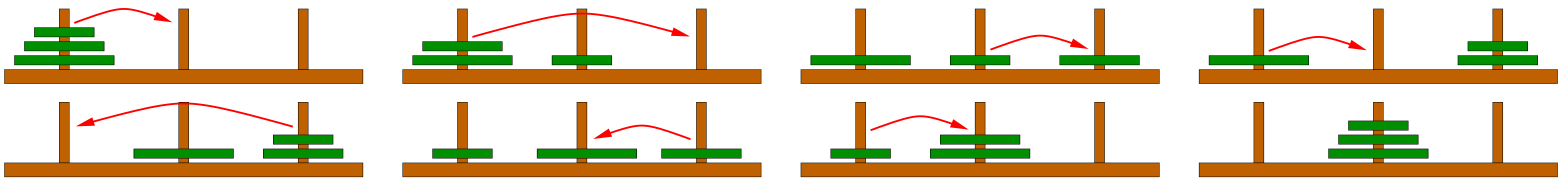
Recursive Programming: Towers of Hanoi

- *Objective:*
  - ◇ Move tower of N disks from peg `a` to peg `b`, with the help of peg `c`.
- *Rules:*
  - ◇ Only one disk can be moved at a time.
  - ◇ A larger disk can never be placed on top of a smaller disk.



Recursive Programming: Towers of Hanoi (Contd.)

- We will call the main predicate `hanoi_moves(N,Moves)`
- `N` is the number of disks and `Moves` the corresponding list of "moves".
- Each move `move(A, B)` represents that the top disk in A should be moved to B.
- *Example:*

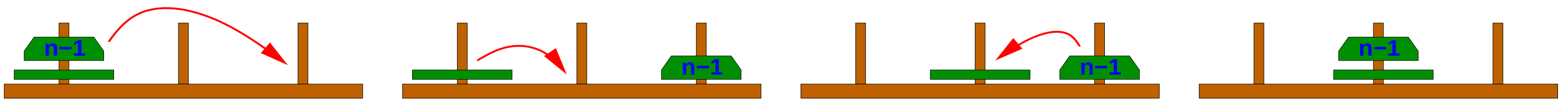


is represented by:

```
hanoi_moves( s(s(s(0))),
             [ move(a,b), move(a,c), move(b,c), move(a,b),
               move(c,a), move(c,b), move(a,b) ])
```

Recursive Programming: Towers of Hanoi (Contd.)

- A general rule:



- We capture this in a predicate $\boxed{\texttt{hanoi(N,Orig,Dest,Help,Moves)}}$ where "`Moves` contains the moves needed to move a tower of `N` disks from peg `Orig` to peg `Dest`, with the help of peg `Help`."

```
hanoi(s(0),Orig,Dest,_Help,[move(Orig, Dest)]) <- .
hanoi(s(N),Orig,Dest,Help,Moves) <-
        hanoi(N,Orig,Help,Dest,Moves1),
        hanoi(N,Help,Dest,Orig,Moves2),
        append(Moves1,[move(Orig, Dest)|Moves2],Moves).
```

- And we simply call this predicate:

```
hanoi_moves(N,Moves) <-
        hanoi(N,a,b,c,Moves).
```

Summary

- Pure logic programs allow purely declarative programming.
- Still, pure logic programming has full computational power.