

Computational Logic

A “Hands-on” Introduction to Pure Logic Programming

1

Syntax: Terms (Variables, Constants, and Structures)

(using Prolog notation conventions)

- **Variables:** start with uppercase character (or “_”), may include “_” and digits:
Examples: X, Im4u, A_little_garden, _, _x, _22
- **Constants:** lowercase first character, may include “_” and digits. Also, numbers and some special characters. Quoted, any character:
Examples: a, dog, a_big_cat, 23, 'Hungry man', []
- **Structures:** a **functor** (the structure name, is like a constant name) followed by a fixed number of arguments between parentheses:
Example: date(monday, Month, 1994)
Arguments can in turn be variables, constants and structures.
 - ◇ **Arity:** is the number of arguments of a structure. Functors are represented as *name/arity*. A constant can be seen as a structure with arity zero.

Variables, constants, and structures as a whole are called **terms** (they are the terms of a “first-order language”): the *data structures* of a logic program.

2

Syntax: Terms

(using Prolog notation conventions)

- Examples of terms:

Term	Type	Main functor:
dad	constant	dad/0
time(min, sec)	structure	time/2
pair(Calvin, tiger(Hobbes))	structure	pair/2
Tee(Alf, rob)	illegal	—
A_good_time	variable	—

- *Functors* can be defined as *prefix*, *postfix*, or *infix* operators (just syntax!):

a + b	is the term	'+' (a, b)	if +/2 declared infix
- b	is the term	'-' (b)	if -/1 declared prefix
a < b	is the term	'<' (a, b)	if </2 declared infix
john father mary	is the term	father(john, mary)	if father/2 declared infix

We assume that some such operator definitions are always preloaded.

3

Syntax: Rules and Facts (Clauses)

- **Rule:** an expression of the form:

$$\begin{array}{l}
 p_0(t_1, t_2, \dots, t_{n_0}) \leftarrow \\
 \quad p_1(t_1^1, t_2^1, \dots, t_{n_1}^1), \\
 \quad \dots \\
 \quad p_m(t_1^m, t_2^m, \dots, t_{n_m}^m).
 \end{array}$$

- ◇ $p_0(\dots)$ to $p_m(\dots)$ are *syntactically* like *terms*.
- ◇ $p_0(\dots)$ is called the **head** of the rule.
- ◇ The p_i to the right of the arrow are called *literals* and form the **body** of the rule. They are also called **procedure calls**.

- **Fact:** an expression of the form $p(t_1, t_2, \dots, t_n) \leftarrow .$ (i.e., a rule with empty body).

Example: meal(soup, beef, coffee) <- .
 meal(First, Second, Third) <-
 appetizer(First),
 main_dish(Second),
 dessert(Third).

- Rules and facts are both called **clauses**.

4

Syntax: Predicates, Programs, and Queries

- **Predicate** (or *procedure definition*): a set of clauses whose heads have the same name and arity (called the **predicate name**).

Examples:

```
pet(spot) <- .                animal(spot) <- .
pet(X) <- animal(X), barks(X). animal(barry) <- .
pet(X) <- animal(X), meows(X). animal(hobbes) <- .
```

Predicate `pet/1` has three clauses. Of those, one is a fact and two are rules.
Predicate `animal/1` has three clauses, all facts.

- **Logic Program:** a set of predicates.

- **Query:** an expression of the form:
(i.e., a clause without a head).

$$\leftarrow p_1(t_1^1, \dots, t_{n_1}^1), \dots, p_n(t_1^n, \dots, t_{n_m}^n).$$

A query represents a *question to the program*.

Example: `<- pet(X)`.

5

“Declarative” Meaning of Facts and Rules

The declarative meaning is the corresponding one in first order logic, according to certain conventions:

- **Facts:** state things that are true.
(Note that a fact “`p <- .`” can be seen as the rule “`p <- true.`”)

Example: the fact `animal(spot) <-.`
can be read as “spot is an animal”.

- **Rules:**

◇ Commas in rule bodies represent conjunction, i.e.,

$p \leftarrow p_1, \dots, p_m$ represents $p \leftarrow p_1 \wedge \dots \wedge p_m$.

◇ “`<-`” represents as usual logical implication.

Thus, a rule $p \leftarrow p_1, \dots, p_m$ means “if p_1 and ... and p_m are true, then p is true”

Example: the rule `pet(X) <- animal(X), barks(X)`.
can be read as “X is a pet if it is an animal and it barks”.

6

“Declarative” Meaning of Predicates and Queries

- **Predicates:** clauses in the same predicate

$p \leftarrow p_1, \dots, p_n$

$p \leftarrow q_1, \dots, q_m$

...

provide different *alternatives* (for p).

Example: the rules

`pet(X) <- animal(X), barks(X).`

`pet(X) <- animal(X), meows(X).`

express two ways for X to be a pet.

- **Note** (variable *scope*): the X vars. in the two clauses above are different, despite the same name. Vars. are *local to clauses* (and are *renamed* any time a clause is used –as with vars. local to a procedure in conventional languages).
- A **query** represents a *question to the program*.

Examples:

`<- pet(spot).`

asks whether `spot` is a pet.

`<- pet(X).`

asks: “Is there an X which is a pet?”

7

“Execution” and Semantics

- Example of a **logic program:**

`pet(X) <- animal(X), barks(X).`

`pet(X) <- animal(X), meows(X).`

`animal(spot) <-.`

`barks(spot) <-.`

`animal(barry) <-.`

`meows(barry) <-.`

`animal(hobbes) <-.`

`roars(hobbes) <-.`

- **Execution:** given a program and a query, *executing* the logic program is *attempting to find an answer to the query*.

Example: given the program above and the query `<- pet(X).`

the system will try to find a “substitution” for X which makes `pet(X)` true.

- ◇ The **declarative semantics** specifies *what* should be computed (all possible answers).
⇒ Intuitively, we have two possible answers: $X = \text{spot}$ and $X = \text{barry}$.
- ◇ The **operational semantics** specifies *how* answers are computed (which allows us to determine *how many steps* it will take).

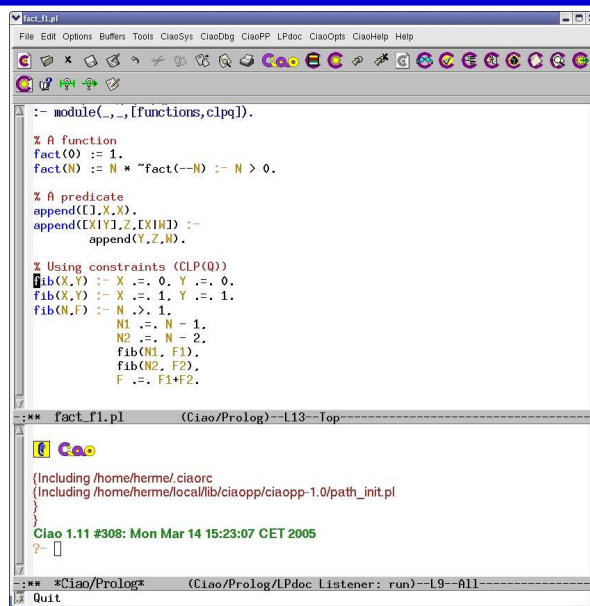
8

Running Pure Logic Programs: the Ciao System's bf/af Packages

- We will be using *Ciao*, a multiparadigm programming system which includes (as one of its “paradigms”) a *pure logic programming* subsystem:
 - ◇ A number of *fair* search rules are available (breadth-first, iterative deepening, ...): we will use “breadth-first” (bf or af).
 - ◇ Also, a module can be set to *pure* mode so that impure built-ins are not accessible to the code in that module.
 - ◇ This provides a reasonable first approximation of “Greene’s dream” (of course, at a cost in memory and execution time).
- Writing programs to execute in bf mode:
 - ◇ All files should start with the following line:
`:- module(_,_,[bf]).` (or `:- module(_,_,['bf/af']).`)
or, for “user” files, i.e., files that are not modules: `:- use_package(bf).`
 - ◇ The *neck* (arrow) of rules must be `<-`.
 - ◇ Facts must end with `<- .`.

9

Ciao Programming Environment: file being edited and top-level



```
fact.pl
File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help
:- module(_,_,[functions,clpq]).

% A function
fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.

% A predicate
append([_],_,_).
append([X|_],Z,[X|_]) :-
    append(_ ,Z,_).

% Using constraints (CLP(Q))
fib(X,Y) :- X .:= 0, Y .:= 0.
fib(X,Y) :- X .:= 1, Y .:= 1.
fib(N,F) :-
    N .:= N - 1,
    N2 .:= N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F .:= F1+F2.

*** fact.pl (Ciao/Prolog)--L13--Top-----
[Ciao]
{Including /home/herme/.ciaoorc
{Including /home/herme/local/lib/ciaoapp/ciaoapp-1.0/path_init.pl
}
}
Ciao 1.11 #308: Mon Mar 14 15:23:07 CET 2005
?-
*** *Ciao/Prolog* (Ciao/Prolog/LPdoc Listener: run)--L9--All-----
Quit
```

10

Top Level Interaction Example

- File `pets.pl` contains:

```
:- module(_,_,[bf]).
```

+ *the pet example code as in previous slides.*

- Interaction with the system query evaluator (the “top level”):

```
Ciao 1.13 #0: Mon Nov 7 09:48:51 MST 2005
?- use_module(pets).
yes
?- pet(spot).
yes
?- pet(X).
X = spot ? ;
X = barry ? ;
no
?-
```

11

Simple (Top-Down) Operational Meaning of Programs

- A logic program is operationally a set of *procedure definitions* (the predicates).

- A query $\leftarrow p$ is an initial *procedure call*.

- A procedure definition with one *clause* $p \leftarrow p_1, \dots, p_m$ means:
“to execute a call to p you have to *call* p_1 and \dots and p_m ”

◊ In principle, the order in which p_1, \dots, p_n are called does not matter, but, in practical systems it is fixed.

- If several clauses (definitions) $p \leftarrow p_1, \dots, p_n$ means:
 $p \leftarrow q_1, \dots, q_m$

“to execute a call to p , call $p_1 \wedge \dots \wedge p_n$, or, alternatively, $q_1 \wedge \dots \wedge q_m$, or \dots ”

- ◊ Unique to logic programming –it is like having several alternative procedure definitions.
- ◊ Means that several possible paths may exist to a solution and they *should be explored*.
- ◊ System usually stops when the first solution found, user can ask for more.
- ◊ Again, in principle, the order in which these paths are explored does not matter (*if certain conditions are met*), but, for a given system, this is typically also fixed.

In the following we define a more precise operational semantics.

12

Unification: uses

- **Unification** is the mechanism used in procedure calls to:
 - ◊ Pass parameters.
 - ◊ “Return” values.
- It is also used to:
 - ◊ Access parts of structures.
 - ◊ Give values to variables.

Unification

- **Unifying two terms (or literals) A and B:** is asking if they can be made syntactically identical by giving (minimal) values to their variables.
 - ◊ I.e., find a **variable substitution** θ such that $A\theta = B\theta$ (or, if impossible, *fail*).
 - ◊ Only variables can be given values!
 - ◊ Two structures can be made identical only by making their arguments identical.

E.g.:

A	B	θ	$A\theta$	$B\theta$
dog	dog	\emptyset	dog	dog
X	a	$\{X = a\}$	a	a
X	Y	$\{X = Y\}$	Y	Y
f(X, g(t))	f(m(h), g(M))	$\{X=m(h), M=t\}$	f(m(h), g(t))	f(m(h), g(t))
f(X, g(t))	f(m(h), t(M))	Impossible (1)		
f(X, X)	f(Y, l(Y))	Impossible (2)		

- (1) Structures with different name and/or arity cannot be unified.
- (2) A variable cannot be given as value a term which contains that variable, because it would create an infinite term. This is known as the **occurs check**.

Unification

- Often several solutions exist, e.g.:

A	B	θ_1	$A\theta_1$ and $B\theta_1$
$f(X, g(T))$	$f(m(H), g(M))$	$\{ X=m(a), H=a, M=b, T=b \}$	$f(m(a), g(b))$
"	"	$\{ X=m(H), M=f(A), T=f(A) \}$	$f(m(H), g(f(A)))$

These are correct, but a simpler ("more general") solution exists:

A	B	θ_1	$A\theta_1$ and $B\theta_1$
$f(X, g(T))$	$f(m(H), g(M))$	$\{ X=m(H), T=M \}$	$f(m(H), g(M))$

- Always a unique (modulo variable renaming) *most general* solution exists (unless unification fails).
- This is the one that we are interested in.
- The *unification algorithm* finds this solution.

15

Unification Algorithm

- Let A and B be two terms:

- 1 $\theta = \emptyset, E = \{A = B\}$

- 2 while not $E = \emptyset$:

- 2.1 delete an equation $T = S$ from E

- 2.2 case T or S (or both) are (distinct) variables. Assuming T variable:

- * (occur check) if T occurs in the term $S \rightarrow$ halt with failure

- * substitute variable T by term S in all terms in θ

- * substitute variable T by term S in all terms in E

- * add $T = S$ to θ

- 2.3 case T and S are non-variable terms:

- * if their names or arities are different \rightarrow halt with failure

- * obtain the arguments $\{T_1, \dots, T_n\}$ of T and $\{S_1, \dots, S_n\}$ of S

- * add $\{T_1 = S_1, \dots, T_n = S_n\}$ to E

- 3 halt with θ being the m.g.u of A and B

16

Unification Algorithm Examples (I)

- Unify: $A = p(X, X)$ and $B = p(f(Z), f(W))$

θ	E	T	S
$\{\}$	$\{ p(X, X) = p(f(Z), f(W)) \}$	$p(X, X)$	$p(f(Z), f(W))$
$\{\}$	$\{ X = f(Z), X = f(W) \}$	X	$f(Z)$
$\{ X = f(Z) \}$	$\{ f(Z) = f(W) \}$	$f(Z)$	$f(W)$
$\{ X = f(Z) \}$	$\{ Z = W \}$	Z	W
$\{ X = f(W), Z = W \}$	$\{\}$		

- Unify: $A = p(X, f(Y))$ and $B = p(Z, X)$

θ	E	T	S
$\{\}$	$\{ p(X, f(Y)) = p(Z, X) \}$	$p(X, f(Y))$	$p(Z, X)$
$\{\}$	$\{ X = Z, f(Y) = X \}$	X	Z
$\{ X = Z \}$	$\{ f(Y) = Z \}$	$f(Y)$	Z
$\{ X = f(Y), Z = f(Y) \}$	$\{\}$		

17

Unification Algorithm Examples (II)

- Unify: $A = p(X, f(Y))$ and $B = p(a, g(b))$

θ	E	T	S
$\{\}$	$\{ p(X, f(Y)) = p(a, g(b)) \}$	$p(X, f(Y))$	$p(a, g(b))$
$\{\}$	$\{ X = a, f(Y) = g(b) \}$	X	a
$\{ X = a \}$	$\{ f(Y) = g(b) \}$	$f(Y)$	$g(b)$
<i>fail</i>			

- Unify: $A = p(X, f(X))$ and $B = p(Z, Z)$

θ	E	T	S
$\{\}$	$\{ p(X, f(X)) = p(Z, Z) \}$	$p(X, f(X))$	$p(Z, Z)$
$\{\}$	$\{ X = Z, f(X) = Z \}$	X	Z
$\{ X = Z \}$	$\{ f(Z) = Z \}$	$f(Z)$	Z
<i>fail</i>			

18

A (Schematic) Interpreter for Logic Programs (SLD-resolution)

Input: A logic program P , a query Q

Output: $Q\mu$ (answer substitution) if Q is provable from P , *failure* otherwise

Algorithm:

1. Initialize the “resolvent” R to be $\{Q\}$
 2. While R is nonempty do:
 - 2.1. Take the leftmost literal A in R
 - 2.2. Choose a (*renamed*) clause $A' \leftarrow B_1, \dots, B_n$ from P , such that A and A' unify with unifier θ
(if no such clause can be found, branch is *failure*; explore another branch)
 - 2.3. Remove A from R , add B_1, \dots, B_n to R
 - 2.4. Apply θ to R and Q
 3. If R is empty, output Q (a solution). Explore another branch for more sol's.
- Step 2.2 defines *alternative paths* to be explored to find answer(s); execution explores this tree (for example, breadth-first).

19

A (Schematic) Interpreter for Logic Programs (Contd.)

- Since step 2.2 is left open, a given logic *programming* system must specify how it deals with this by providing one (or more)
 - ◊ **Search rule(s):** “how are clauses/branches selected in 2.2.”
- If the search rule is not specified execution is *nondeterministic*, since choosing a different clause (in step 2.2) can lead to different solutions (finding solutions in a different order).

Example (two valid executions):

?- pet(X).	?- pet(X).
X = spot ? ;	X = barry ? ;
X = barry ? ;	X = spot ? ;
no	no
?-	?-

- In fact, there is also some freedom in step 2.1, i.e., a system may also specify:
 - ◊ **Computation rule(s):** “how are literals selected in 2.1.”

20

Running programs

C_1 : `pet(X) <- animal(X), barks(X).`
 C_2 : `pet(X) <- animal(X), meows(X).`
 C_3 : `animal(spot) <-.`
 C_4 : `animal(barry) <-.`
 C_5 : `animal(hobbes) <-.`
 C_6 : `barks(spot) <-.`
 C_7 : `meows(barry) <-.`
 C_8 : `roars(hobbes) <-.`

- `<- pet(P).`

Q	R	Clause	θ
<code>pet(P)</code>	<code><u>pet(P)</u></code>	C_2^*	$\{P = X_1\}$
<code>pet(X₁)</code>	<code><u>animal(X₁)</u>, meows(X₁)</code>	C_4^*	$\{X_1 = \text{barry}\}$
<code>pet(barry)</code>	<code><u>meows(barry)</u></code>	C_7	$\{\}$
<code>pet(barry)</code>	—	—	—

* means there is a *choice-point*, i.e., there are other clauses whose head unifies.

- System response: `P = barry ?`
- If we type “;” after the ? prompt (i.e., we ask for another solution) the system can go and execute a different branch (i.e., a different choice in C_2^* or C_4^*).

21

Running programs (different strategy)

C_1 : `pet(X) <- animal(X), barks(X).`
 C_2 : `pet(X) <- animal(X), meows(X).`
 C_3 : `animal(spot) <-.`
 C_4 : `animal(barry) <-.`
 C_5 : `animal(hobbes) <-.`
 C_6 : `barks(spot) <-.`
 C_7 : `meows(barry) <-.`
 C_8 : `roars(hobbes) <-.`

- `<- pet(P).` (different strategy)

Q	R	Clause	θ
<code>pet(P)</code>	<code><u>pet(P)</u></code>	C_1^*	$\{P = X_1\}$
<code>pet(X₁)</code>	<code><u>animal(X₁)</u>, barks(X₁)</code>	C_5^*	$\{X_1 = \text{hobbes}\}$
<code>pet(hobbes)</code>	<code><u>barks(hobbes)</u></code>	???	<i>failure</i>

→ explore another branch (different choice in C_1^* or C_5^*) to find a solution.
We take C_3 instead of C_5 :

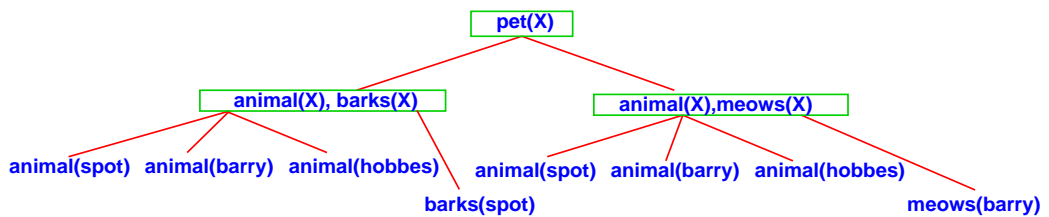
Q	R	Clause	θ
<code>pet(P)</code>	<code><u>pet(P)</u></code>	C_1^*	$\{P = X_1\}$
<code>pet(X₁)</code>	<code><u>animal(X₁)</u>, barks(X₁)</code>	C_3^*	$\{X_1 = \text{spot}\}$
<code>pet(spot)</code>	<code><u>barks(spot)</u></code>	C_6	$\{\}$
<code>pet(spot)</code>	—	—	—

22

The Search Tree

- A query + a logic program together specify a *search tree*.

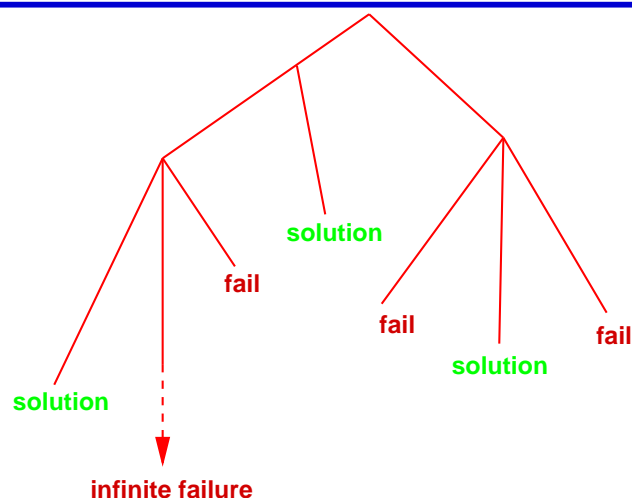
Example: query $\leftarrow \text{pet}(X)$ with the previous program generates this search tree (the boxes represent the “and” parts [except leaves]):



- Different query \rightarrow different tree.
- The search and computation rules explain how the search tree will be explored during execution.
- How can we achieve completeness (guarantee that all solutions will be found)?

23

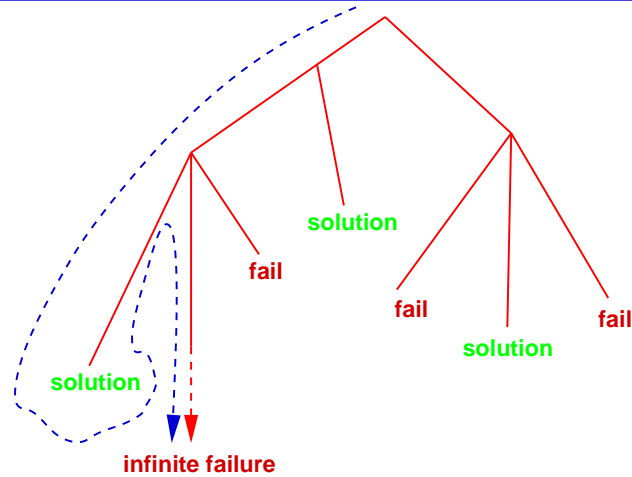
Characterization of The Search Tree



- All solutions are at *finite depth* in the tree.
- Failures can be at finite depth or, in some cases, be an infinite branch.

24

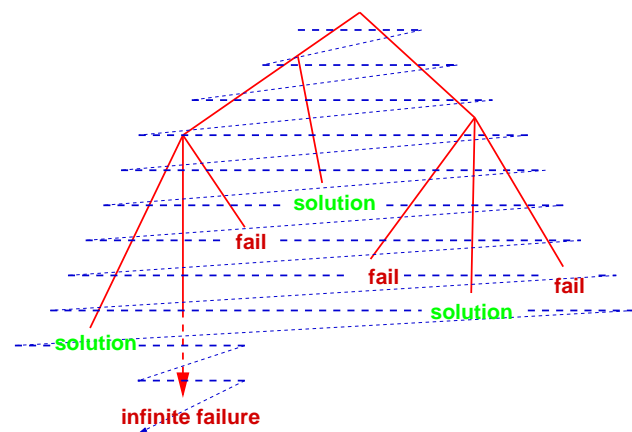
Depth-First Search



- Incomplete: may fall through an infinite branch before finding all solutions.
- But very efficient: it can be implemented with a call stack, very similar to a traditional programming language.

25

Breadth-First Search



- Will find all solutions before falling through an infinite branch.
- But costly in terms of time and memory.
- Used in all the following examples (via Ciao's bf package).

26

Role of Unification in Execution and Modes

- As mentioned before, unification used to *access data* and *give values to variables*.

Example: Consider query `<- animal(A), named(A,Name)` . with:

```
animal(dog(barry)) <- .    named(dog(Name),Name) <- .
```

- Also, unification is used to *pass parameters* in procedure calls and to *return values* upon procedure exit.

Q	R	Clause	θ
pet(P)	pet(P)	C_1^*	$\{P=X_1\}$
pet(X_1)	animal(X_1), barks(X_1)	C_3^*	$\{X_1=spot\}$
pet(spot)	barks(spot)	C_6	$\{\}$
pet(spot)	—	—	—

- In fact, argument positions are not fixed a priory to be input or output.

Example: Consider query `<- pet(spot)` . vs. `<- pet(X)` .

or `<- add(s(0),s(s(0)),Z)` . vs. `<- add(s(0),Y,s(s(s(0))))` .

- Thus, procedures can be used in different **modes** (different sets of arguments are input or output in each mode).

27

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}

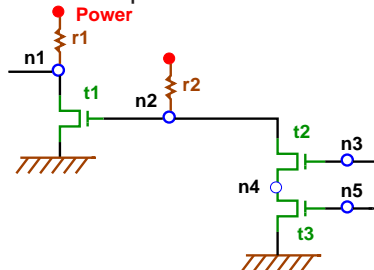
<- grandfather_of(X, michael).
Answer: {X = john}
<- grandfather_of(X, Y).
Answer: {X = john, Y = michael}
Answer: {X = john, Y = david}
<- grandfather_of(X, X).
Answer: No
```

- Rules for grandmother_of(X, Y)?

28

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}`

29

Structured Data and Data Abstraction (and the '=' Predicate)

- *Data structures* are created using (complex) terms.
- Structuring data is important:


```
course(complog,wed,19,00,20,30,'M.', 'Hermenegildo',new,5102) <- .
```
- When is the Computational Logic course?


```
<- course(complog,Day,StartH,StartM,FinishH,FinishM,C,D,E,F).
```
- Structured version:


```
course(complog,Time,Lecturer, Location) <-
  Time = t(wed,18:30,20:30),
  Lecturer = lect('M.', 'Hermenegildo'),
  Location = loc(new,5102).
```

Note: "X=Y" is equivalent to "'='(X,Y)" where the predicate `=/2` is defined as the fact `"=''(X,X) <- ."` – Plain unification!

- Equivalent to:


```
course(complog, t(wed,18:30,20:30),
  lect('M.', 'Hermenegildo'), loc(new,5102)) <- .
```

30

Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

```
course(complog,Time,Lecturer, Location) <-  
  Time = t(wed,18:30,20:30),  
  Lecturer = lect('M.', 'Hermenegildo'),  
  Location = loc(new,5102).
```

- When is the Computational Logic course?

```
<- course(complog,Time, A, B).
```

has solution:

```
{Time=t(wed,18:30,20:30), A=lect('M.', 'Hermenegildo'), B=loc(new,5102)}
```

- Using the *anonymous variable* ("_"):

```
<- course(complog,Time, _, _).
```

has solution:

```
{Time=t(wed,18:30,20:30)}
```

31

Structured Data and Data Abstraction (Contd.)

- 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}
```

32

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)

33

Logic Programs and the Relational DB Model (Contd.)

Relational Database → Logic Programming

Relation Name	→ Predicate symbol
Relation	→ Procedure consisting of ground facts (facts without variables)
Tuple	→ Ground fact
Attribute	→ 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

34

Logic Programs and the Relational DB Model (Contd.)

- The operations of the relational model are easily implemented as rules.
 - ◇ *Union:*
 $r_union_s(X_1, \dots, X_n) \leftarrow r(X_1, \dots, X_n).$
 $r_union_s(X_1, \dots, X_n) \leftarrow s(X_1, \dots, X_n).$
 - ◇ *Set Difference:*
 $r_diff_s(X_1, \dots, X_n) \leftarrow r(X_1, \dots, X_n), \text{ not } s(X_1, \dots, X_n).$
 $r_diff_s(X_1, \dots, X_n) \leftarrow s(X_1, \dots, X_n), \text{ not } r(X_1, \dots, X_n).$
(we postpone the discussion on *negation* until later.)
 - ◇ *Cartesian Product:*
 $r_X_s(X_1, \dots, X_m, X_{m+1}, \dots, X_{m+n}) \leftarrow r(X_1, \dots, X_m), s(X_{m+1}, \dots, X_{m+n}).$
 - ◇ *Projection:*
 $r13(X_1, X_3) \leftarrow r(X_1, X_2, X_3).$
 - ◇ *Selection:*
 $r_selected(X_1, X_2, X_3) \leftarrow r(X_1, X_2, X_3), \leq(X_2, X_3).$
(see later for definition of $\leq/2$)

35

Logic Programs and the Relational DB Model (Contd.)

- Derived operations – some can be expressed more directly in LP:
 - ◇ *Intersection:*
 $r_meet_s(X_1, \dots, X_n) \leftarrow r(X_1, \dots, X_n), s(X_1, \dots, X_n).$
 - ◇ *Join:*
 $r_joinX2_s(X_1, \dots, X_n) \leftarrow r(X_1, X_2, X_3, \dots, X_n), s(X'_1, X_2, X'_3, \dots, X'_n).$
- Duplicates an issue: see “setof” later in Prolog.

36

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.

37

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.

38

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) <- .
```

39

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).
```

40

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

- 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

- Definition of $\text{mod}(X, Y, Z)$
“Z is the remainder from dividing X by Y”
($\exists Q$ s.t. $X = Y * Q + Z$ and $Z < Y$):
 $\text{mod}(X, Y, Z) \leftarrow \text{less}(Z, Y), \text{times}(Y, Q, W), \text{plus}(W, Z, X).$

 $\text{less}(0, s(X)) \leftarrow \text{nat}(X).$
 $\text{less}(s(X), s(Y)) \leftarrow \text{less}(X, Y).$
- Another possible definition:
 $\text{mod}(X, Y, X) \leftarrow \text{less}(X, Y).$
 $\text{mod}(X, Y, Z) \leftarrow \text{plus}(X1, Y, X), \text{mod}(X1, Y, Z).$
- The second is much more efficient than the first one
(compare the size of the proof trees).

43

Recursive Programming: Arithmetic/Functions

- The Ackermann function:
 $\text{ackermann}(0, N) = N + 1$
 $\text{ackermann}(M, 0) = \text{ackermann}(M - 1, 1)$
 $\text{ackermann}(M, N) = \text{ackermann}(M - 1, \text{ackermann}(M, N - 1))$
- In Peano arithmetic:
 $\text{ackermann}(0, N) = s(N)$
 $\text{ackermann}(s(M), 0) = \text{ackermann}(M, s(0))$
 $\text{ackermann}(s(M), s(N)) = \text{ackermann}(M, \text{ackermann}(s(M), N))$
- Can be defined as:
 $\text{ackermann}(0, N, s(N)) \leftarrow .$
 $\text{ackermann}(s(M), 0, \text{Val}) \leftarrow \text{ackermann}(M, s(0), \text{Val}).$
 $\text{ackermann}(s(M), s(N), \text{Val}) \leftarrow \text{ackermann}(s(M), N, \text{Val1}),$
 $\text{ackermann}(M, \text{Val1}, \text{Val}).$
- 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).

44

Recursive Programming: Lists

- Binary structure: first argument is *element*, second argument is *rest* of the list.
- We need:
 - ◊ a constant symbol: the empty list denoted by the *constant* `[]`
 - ◊ a functor of arity 2: traditionally the dot `.` (which is overloaded).
- Syntactic sugar: the term `.(X,Y)` is denoted by `[X|Y]` (X is the *head*, Y is the *tail*).

<i>Formal object</i>	<i>Cons pair syntax</i>	<i>Element syntax</i>
<code>.(a,[])</code>	<code>[a []]</code>	<code>[a]</code>
<code>.(a,.(b,[]))</code>	<code>[a [b []]]</code>	<code>[a,b]</code>
<code>.(a,.(b,.(c,[])))</code>	<code>[a [b [c []]]]</code>	<code>[a,b,c]</code>
<code>.(a,X)</code>	<code>[a X]</code>	<code>[a X]</code>
<code>.(a,.(b,X))</code>	<code>[a [b X]]</code>	<code>[a,b X]</code>

- Note that:

<code>[a,b]</code> and <code>[a X]</code> unify with $\{X = [b]\}$	<code>[a]</code> and <code>[a X]</code> unify with $\{X = []\}$
<code>[a]</code> and <code>[a,b X]</code> do not unify	<code>[]</code> and <code>[X]</code> do not unify

45

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).
```

46

Recursive Programming: Lists (Contd.)

- X is a *member* of the list Y:

```
member(a, [a]) <- member(b, [b]) <- etc.      => member(X, [X]) <- .
member(a, [a,c]) <- member(b, [b,d]) <- etc.  => member(X, [X,Y]) <- .
member(a, [a,c,d]) <- member(b, [b,d,l]) <- etc. => member(X, [X,Y,Z]) <- .
=> member(X, [X|Y]) <- list(Y).
```

```
member(a, [c,a]), member(b, [d,b]). etc.      => member(X, [Y,X]).
member(a, [c,d,a]). member(b, [s,t,b]). etc.  => member(X, [Y,Z,X]).
=> member(X, [Y|Z]) <- member(X,Z).
```

- Resulting definition:

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

47

Recursive Programming: Lists (Contd.)

- Resulting definition:

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

- Uses of member(X,Y):

- ◇ checking whether an element is in a list (member(b, [a,b,c]))
- ◇ finding an element in a list (member(X, [a,b,c]))
- ◇ finding a list containing an element (member(a, Y))

- Define: prefix(X,Y) (the list X is a prefix of the list Y), e.g.
prefix([a, b], [a, b, c, d])
- Define: suffix(X,Y), sublist(X,Y), ...
- Define length(Xs,N) (N is the length of the list Xs)

48

Recursive Programming: Lists (Contd.)

- Concatenation of lists:

- ◇ Base case:

`append([], [a], [a]) <- .` `append([], [a,b], [a,b]) <- .` *etc.*
⇒ `append([], Ys, Ys) <- list(Ys).`

- ◇ Rest of cases (first step):

`append([a], [b], [a,b]) <- .`
`append([a], [b,c], [a,b,c]) <- .` *etc.*
⇒ `append([X], Ys, [X|Ys]) <- list(Ys).`
`append([a,b], [c], [a,b,c]) <- .`
`append([a,b], [c,d], [a,b,c,d]) <- .` *etc.*
⇒ `append([X,Z], Ys, [X,Z|Ys]) <- list(Ys).`

This is still infinite → we need to generalize more.

49

Recursive Programming: Lists (Contd.)

- Second generalization:

`append([X], Ys, [X|Ys]) <- list(Ys).`
`append([X,Z], Ys, [X,Z|Ys]) <- list(Ys).`
`append([X,Z,W], Ys, [X,Z,W|Ys]) <- list(Ys).`
⇒ `append([X|Xs], Ys, [X|Zs]) <- append(Xs, Ys, Zs).`

- So, we have:

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

- Uses of `append`:

- ◇ concatenate two given lists: `<- append([a,b], [c], Z)`
 - ◇ find differences between lists: `<- append(X, [c], [a,b,c])`
 - ◇ split a list: `<- append(X, Y, [a,b,c])`

50

Recursive Programming: Lists (Contd.)

- `reverse(Xs, Ys)`: `Ys` is the list obtained by reversing the elements in the list `Xs`
It is clear that we will need to traverse the list `Xs`
For each element `X` of `Xs`, we must put `X` at the end of the rest of the `Xs` list already reversed:

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

How can we stop?

```
reverse([], []) <-.
```

- As defined, `reverse(Xs, Ys)` is very inefficient. Another possible definition:

```
reverse(Xs, Ys) <- reverse(Xs, [], Ys).
```

```
reverse([], Ys, Ys) <-.
```

```
reverse([X|Xs], Acc, Ys) <- reverse(Xs, [X|Acc], Ys).
```

- Find the differences in terms of efficiency between the two definitions.

51

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

52

Recursive Programming: Binary Trees

- 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).
```

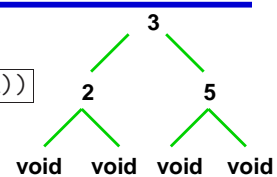
- Define `in_order(Tree,Order)`, `post_order(Tree,Order)`.

53

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;
...
```

54

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)) <- lt_member(X, L).  
lt_member(X, tree(Y,L,R)) <- 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 (somewhat surprising): `<- lt_member(M, T)`.

55

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).
```

56

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*D.V) <- deriv(U,X,DU), deriv(V,X,DV) .
deriv(U/V,X,(DU*V-U*D.V)/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) .
...

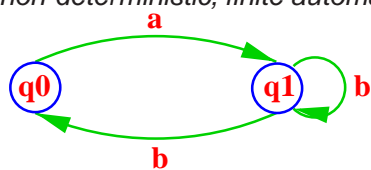
```

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

57

Recursive Programming: Automata (Graphs)

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



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) .

```

58

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?

59

Recursive Programming: Towers of Hanoi

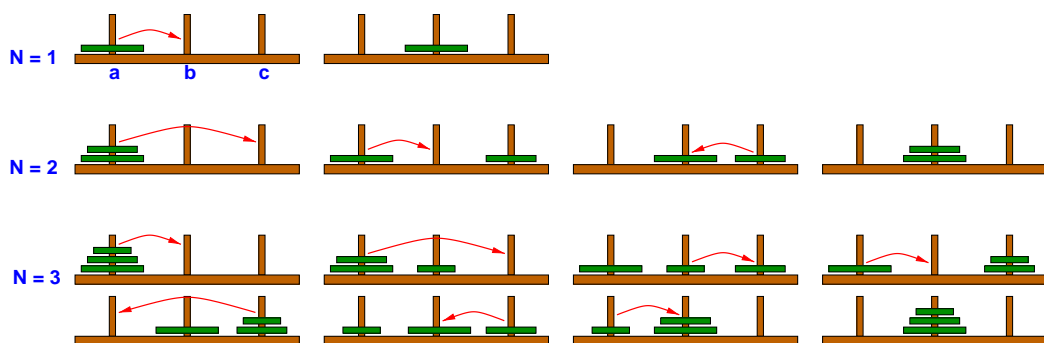
- Objective:

◇ 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.

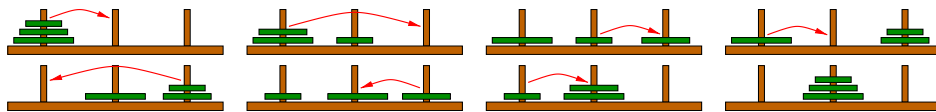
◇ A larger disk can never be placed on top of a smaller disk.



60

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:*



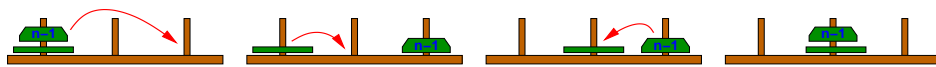
is represented by:

```
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) ] )
```

61

Recursive Programming: Towers of Hanoi (Contd.)

- A general rule:



- We capture this in a predicate `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) .
```

62

Learning to Compose Recursive Programs

- To some extent it is a simple question of practice.
- By induction (as in the previous examples): elegant, but generally difficult – not the way most people do it.
- State first the base case(s), and then think about the general recursive case(s).
- Sometimes it may help to compose programs with a given use in mind (e.g., “forwards execution”), making sure it is declaratively correct. Consider also if alternative uses make declarative sense.
- Sometimes it helps to look at well-written examples and use the same “schemas”.
- Global top-down design approach:
 - ◊ state the general problem
 - ◊ break it down into subproblems
 - ◊ solve the pieces
- Again, best approach: practice.