Computational Logic

A "Hands-on" Introduction to Logic Programming

Syntax: Variables, Constants, 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** (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.

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* $\boxed{\textit{operators}}$ (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.

Syntax: Atoms, Literals

- **Atoms:** an expression of the form: $\boxed{\mathtt{p}(t_1,\ t_2,\ \ldots,\ t_n)}$
  - ◇ `p` is the atom's **predicate symbol** (same convention as with functors),
  - ◇ `n` is its arity,
  - ◇ and $t_1,\ t_2,\ \ldots,\ t_n$ are *terms*.
  - ◇ The predicate symbol of an atom is also represented as `p/n`.
- Atoms and terms are syntactically identical!
  They are distinguished by context:
  <u>if</u> `dog(name(barry), color(black))` is an atom
  <u>then</u> `name(barry)` and `color(black)` are terms

  <u>if</u> `color(dog(barry,black))` is an atom
  <u>then</u> `dog(barry,black)` is a term
- I.e., atoms cannot appear inside terms; terms are the arguments of atoms.
- **Literals:** A literal is a positive (non negated) or negative (negated) atom.

Syntax: Rules

- **Rules:** A rule is an expression of the form:

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

  - ◇ The expression to the left of the arrow has to be an *atom* (no negation) and is called the **head** of the rule.
  - ◇ Those to the right of the arrow are *literals* and form the **body** of the rule.
  - ◇ Literals in the body of a rule are also called **procedure calls**.

Example:

```
meal(First, Second, Third) <-
      appetizer(First),
      main_dish(Second),
      dessert(Third).
```


Syntax: Facts, Clauses, Predicates

- **Facts:** A fact is an expression of the form: $\boxed{\texttt{p}(t_1,\ t_2,\ \ldots,\ t_n\ )\ \texttt{<-.}}$
  (i.e., a fact is a rule with an empty body).

  Examples:

```
dog(name(barry), color(black)) <-.
friends('Ann', 'John') <-.
```

- Rules and facts are both called **clauses**.
- **Predicates:** all clauses whose heads have the same name and arity form a predicate (or *procedure*) definition.

Example:

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

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

Declarative Meaning of Facts and Rules

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

- **Rules:**
  - Commas in rule bodies represent conjunction, i.e., $p \leftarrow p_1, \cdots, p_m.$ represents $p \leftarrow p_1 \wedge \cdots \wedge p_m.$
  - "$\leftarrow$" represents as usual logical implication.

  Thus, a rule $p \leftarrow p_1, \cdots, p_m.$ 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".

- **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".

Declarative Meaning of Predicates

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

  $\mathtt{p} \leftarrow \mathtt{p}_1, \ldots, \mathtt{p}_n$
  $\mathtt{p} \leftarrow \mathtt{q}_1, \ldots, \mathtt{q}_m$
  ...

  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` variables in the two clauses above are different, even if they have the same name. Variables are *local to clauses* (and are *renamed* any time a clause is used).

Programs, Queries, and Execution

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

  Example:

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

animal(spot) <-.              barks(spot) <-.
animal(barry) <-.             meows(barry) <-.
animal(hobbes) <-.            roars(hobbes) <-.
```

- **Query:** an expression of the form: $\boxed{\leftarrow p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_m}^n).}$
  (i.e., a clause without a head). A query represents a *question to the program*.
  Example: `<- pet(X).`

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

  Example: above, the system will try to find a "substitution" for `X` which makes `pet(X)` true. Intuitively, we have two possible answers: `spot` and `barry`.

  The *declarative semantics* does not specify *how* this is done – this is the role of the operational semantics.

Operational Meaning

- A logic program also has an *operational* meaning [Kowalski]:
  - A clause $p \leftarrow p_1, \ldots, p_m.$ expresses:
    "to obtain (prove) p you have to obtain (prove) p_1 and ...and p_m first"
    In principle, the order in which body literals $p_1, \ldots, p_n$ are solved does not matter, but, for a given system this may be fixed.
  - A set of clauses:
    $p \leftarrow p_1, \ldots, p_n$
    $p \leftarrow q_1, \ldots, q_m$
    ...
    expresses "to prove p, prove $p_1 \wedge \ldots \wedge p_n$, or prove $q_1 \wedge \ldots \wedge q_n$, or ..."

    The presence of several applicable clauses for a given body literal means that several possible paths exist to a solution and they should be explored.
    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 may also be fixed.

Unifi cation

- Unifying two terms is finding (the minimal) values for the variables in those terms which make them syntactically equal.
- Only variables can be given values!
- Two terms can be made identical only by making identical their arguments.

Example:

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

- (1) Terms 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**.
- All applies to unification of atoms as well!

Substitutions

- If the equation $A = B$ has a solution then A and B are **unifiable**.
- Solutions of a set of term equations are called **substitutions**.
- In a solution all values for the variables are completely explicit!
- **Substitution:** a set of equations assigning values to variables, with:
  - Only variables on the left hand side of each equation.
  - Only one equation for each left hand side variable.
  - Variables on the left hand side cannot appear on the right of any equation.

Example:

Set of Equations	**Substitution**
{ `X=f(Y), Y=f(Z)` }	NO
{ `X=f(f(Z)), Y=f(Z)` }	YES
{ `X=f(f(Z)), Z=Y` }	NO
{ `X=f(f(Z)), Y=Z` }	YES
{ `X=l(Y), Y=l(Y)` }	NO (2)
{ `X=l(Y), X=Y` }	NO!

Unifi ers

- A substitution θ which is a solution of $A = B$ is called a **unifier** of A and B.
- **Most general unifier:** one which assigns to the variables the values $\boxed{\text{strictly}}$ required to unify.
- Given two terms, if they are unifiable, then there exists a *unique* (up to variable renaming) most general unifier (m.g.u.) for them.

  Example:

Terms	**Unifiers**	**MGU (unique!)**
`f(X, g(T))`	{ `X=m(a)`, `H=a`, `M=b`, `T=b` }	{ `X=m(H)`, `M=T` }
`f(m(H), g(M))`	{ `X=m(H)`, `M=f(A)`, `T=f(A)` }	{ `X=m(A)`, `H=A`, `M=B`, `T=B` }

- Unifying two terms: find the $\boxed{\text{minimal}}$ substitution which makes them identical.
- Unification should find the (unique) m.g.u., if it exists, or *fail* otherwise.

Unifi cation 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 $\{T_1, \ldots, T_n\}$ of T and $\{S_1, \ldots, S_n\}$ of S
  - * add $\{T_1 = S_1, \ldots, T_n = S_n\}$ to E

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

Unifi cation 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) }	{}		

Unifi cation 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)
fail			

- 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
fail			

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 $A' \leftarrow B_1, \ldots, B_n$ 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 $B_1, \ldots, B_n$ 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 branches* to be tried before obtaining the solution(s); execution explores this tree (for example, breadth-first).
- (Step 2.1 also allows some freedom, but not needed.)

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

- Dealing with the fact that steps 2.1 and 2.2 are *nondeterministic*.
  A given logic *programming* system must specify how it deals with this by providing two additional rules:
  - **Computation rule:** "which literal is selected in 2.1."
  - **Search rule:** "which clause/branch is 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).
X = spot ? ;
X = barry ? ;
no
?-
```

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

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) <-.`

- $\boxed{\texttt{<- pet(P).}}$

Q	R	Clause	θ
pet(P)	*pet(P)*	C_2*	$\{P = X_1\}$
pet(X_1)	*animal(X_1)*, meows(X_1)	C_4*	$\{X_1 = \text{barry}\}$
pet(barry)	*meows(barry)*	C_7	$\{\}$
pet(barry)	—	—	—

* means that there are other clauses whose head unifies.

- A different branch (i.e., a different choice in C_2* or C_4*) can be explored to try to find another solution.

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) <-.`

- $\boxed{\texttt{<- pet(P).}}$ (different strategy)

Q	R	Clause	θ
pet(P)	*pet(P)*	C_1*	$\{P = X_1\}$
pet(X_1)	*animal(X_1)*, barks(X_1)	C_5*	$\{X_1 = \text{hobbes}\}$
pet(hobbes)	*barks(hobbes)*	???	*failure*

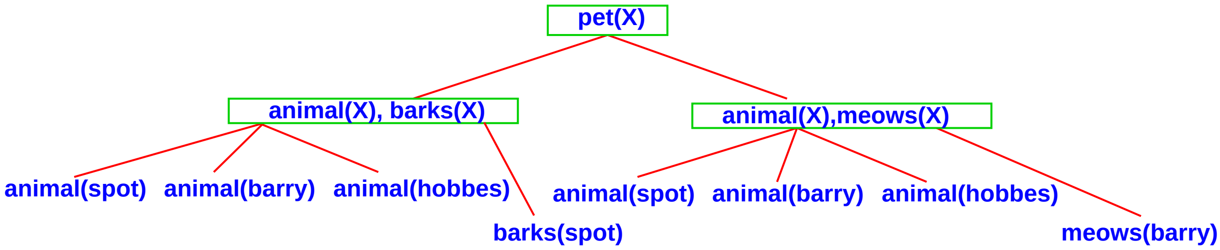
- → explore another branch (different choice in C_1* or C_5*) to find a solution.

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

The Search Tree

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

  Example: query ← `pet(X)` with the previous program generates this search tree (the boxes represent the “and” parts [except leaves]):



- The details of the operational semantics explain how the search tree will be explored during execution.
- Different query → different tree.

Role of Unifi cation in Execution

- Unification is used to *pass parameters* in procedure calls.
- Unification is used 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)`	—	—	—

- Argument positions are not fixed a priory to be input or output.

  Example: Consider query `<- pet(spot).`
- Thus, procedures can be used in different “modes”
  (different sets of arguments are input or output in each mode).
- Unification is also used to *access data*.

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

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

An Example of a Pure and Complete Logic Programming System

- The *Ciao Prolog* system includes a *pure, complete logic programming* subsystem.
  - ◇ A number of *complete* search rules are available (breadth-first, iterative deepening, ...).
  - ◇ A module can be set to *pure* mode so that no impure built-ins are accessible to the code in that module.

  In this case we have a quite good approximation of “Greene’s dream.”
- The following examples will use Ciao’s breadth-first execution mode.
- To achieve this:
  - ◇ The following must be added at the beginning of a file:

    ```
    :- use_package([bf]).
    ```
  - ◇ The *neck* (arrow) of rules must be `<-`.
  - ◇ Facts must end with `<-.`.

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:

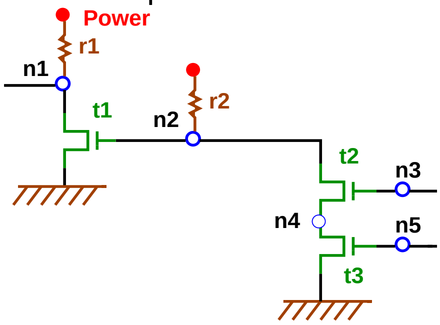
```
 <- father_of(john, peter).
Answer:  Yes
<- father_of(john, david).
Answer:  No
<- father_of(john, X).
Answer:  {X = peter}
Answer:  {X = mary}
```

```
 <- 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 (and the `'='` Predicate)

- *Data structures* are created using (complex) terms.
- Structuring data is important:
  `course(complog,wed,19,00,21,00,'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,19:00,21:00),
         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,19:0,21:00),
       lect('M.','Hermenegildo'),  loc(new,5102)) <-.
```

Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

```
course(complog,Time,Lecturer, Location) <-
        Time = t(wed,19:00,21:00),
        Lecturer = lect('M.','Hermenegildo'),
        Location = loc(new,5102).
```

- When is the Computational Logic course?

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

has solution:

```
{Time=t(wed,19:0,21:0), A=lect('M.','Hermenegildo'), B=loc(new,5102)}
```

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

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

has solution:

```
{Time=t(wed,19:0,21:0)}
```


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}`

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	→	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

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

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

- Multiple uses: `≤(s(0),s(s(0)))`, `≤(X,0),...`
- Multiple solutions: `≤(X,s(0))`, `≤(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 `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) <- Z < Y, times(Y,Q,W), plus(W,Z,X).
```

- Another possible definition:

```
mod(X,Y,X) <- 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

- 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*).

Formal object	*Cons pair syntax*	*Element syntax*
.(a,[ ])	[a|[ ]]	`[a]`
.(a,.(b,[ ]))	[a|[b|[ ]]]	`[a,b]`
.(a,.(b,.(c,[ ])))	[a|[b|[c|[ ]]]]	`[a,b,c]`
.(a,X)	[a|X]	`[a|X]`
.(a,.(b,X))	[a|[b|X]]	`[a,b|X]`

- Note that:
  `[a,b]` and `[a|X]` unify with $\{X = [b]\}$
  `[a]` and `[a,b|X]` do not unify
  `[a]` and `[a|X]` unify with $\{X = [\,]\}$
  `[ ]` and `[X]` do not unify

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).
```

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).
```


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)

Recursive Programming: Lists (Contd.)

- Concatenation of lists:
  - Base case:
    `append([],[a],[a]) <-.   append([],[a,b],[a,b]) <-.` *etc.*
    $\Rightarrow$ **append([],Ys,Ys) <- list(Ys).**
  - Rest of cases (first step):
    `append([a],[b],[a,b]) <-.`
    `append([a],[b,c],[a,b,c]) <-.` *etc.*
    $\Rightarrow$ *append([X],Ys,[X|Ys]) <- list(Ys).*
    `append([a,b],[c],[a,b,c]) <-.`
    `append([a,b],[c,d],[a,b,c,d]) <-.` *etc.*
    $\Rightarrow$ *append([X,Z],Ys,[X,Z|Ys])<- list(Ys).*

    This is still infinite $\rightarrow$ we need to generalize more.

Recursive Programming: Lists (Contd.)

- We note that:
  `append([a,b],Ys,[a|[b|Ys]])` $\equiv$ `append([a,b],Ys,[a|Zs])`
  `with Zs = [b|Ys]`
  `append([a,b,c],Ys,[a|[b|Ys]])` $\equiv$ `append([a,b,c],Ys,[a|Zs])`
  `with Zs = [b|Ws], Ws = [c|Ys].`
  $\Rightarrow$ **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])`

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.

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

- Defining `pre_order(Tree,Order)`:

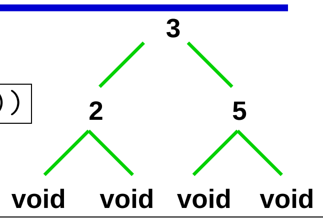
```
pre_order(void,[]) <-.
pre_order(tree(X,Left,Right),Order) <-
     pre_order(Left,OrderLeft),
     pre_order(Right,OrderRight),
     append([X|OrderLeft],OrderRight,Order).
```

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

Creating a Binary Tree in Pascal and Prolog

- 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)) <- 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`

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.)

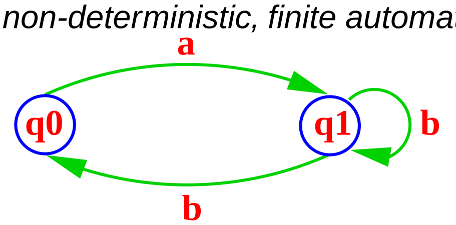
- Symbolic differentiation: `deriv(Expression, X, DifferentiatedExpression)`

```
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: 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.)

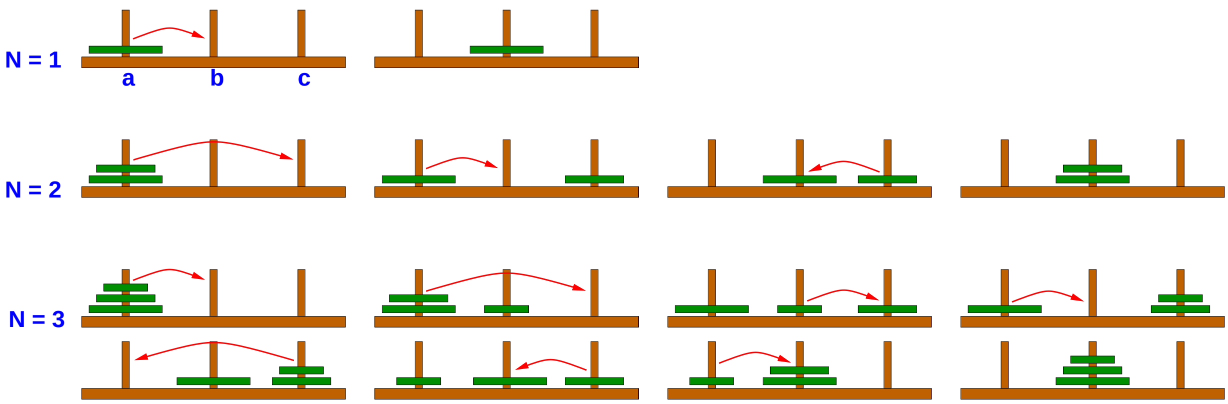
- A *nondeterministic, stack, finite automaton* (NDSFA):

```
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?

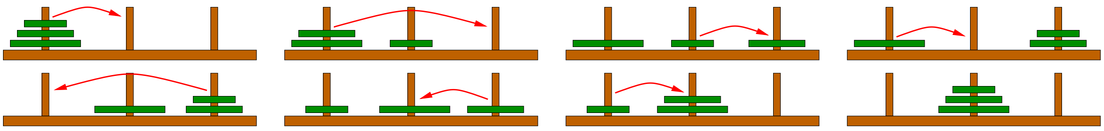
Recursive Programming: Towers of Hanoi

- *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:*

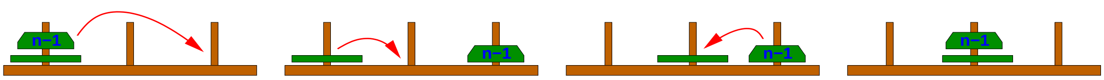


is represented by:

```
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 `hanoi(N,A,B,C,Moves)` where “`Moves` contains the moves needed to move a tower of `N` disks from peg `A` to peg `B`, with the help of peg `C`.”

```
hanoi(s(0),A,B,C,[move(A, B)]) <-.
hanoi(s(N),A,B,C,Moves) <-
    hanoi(N,A,C,B,Moves1),
    hanoi(N,C,B,A,Moves2),
    append(Moves1,[move(A, B)|Moves2],Moves).
```

- And we simply call this predicate:

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

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.