Computational Logic

A “Hands-on” Introduction to Pure Logic Programming
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.
Syntax: Terms

(using Prolog notation conventions)

- **Examples of terms:**

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Main functor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dad</td>
<td>constant</td>
<td>dad/0</td>
</tr>
<tr>
<td>time(min, sec)</td>
<td>structure</td>
<td>time/2</td>
</tr>
<tr>
<td>pair(Calvin, tiger(Hobbes))</td>
<td>structure</td>
<td>pair/2</td>
</tr>
<tr>
<td>Tee(Alf, rob)</td>
<td>illegal</td>
<td>—</td>
</tr>
<tr>
<td>A_good_time</td>
<td>variable</td>
<td>—</td>
</tr>
</tbody>
</table>

- **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.
Syntax: Rules and Facts (Clauses)

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

  \[ p_0(t_1, t_2, \ldots, t_n) \leftarrow p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1), \]
  
  \[ \ldots \]
  
  \[ p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m). \]

  - \( p_0(\ldots) \) to \( p_m(\ldots) \) are *syntactically* like *terms*.
  - \( p_0(\ldots) \) 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 \( p(t_1, t_2, \ldots, t_n) \) (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**.
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:**

  \[
  \text{pet(spot).} \quad \text{animal(spot).}
  \]

  \[
  \text{pet(X) :- animal(X), barks(X).} \quad \text{animal(barry).}
  \]

  \[
  \text{pet(X) :- animal(X), meows(X).} \quad \text{animal(hobbes).}
  \]

  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:

  \[
  \leftarrow p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_m}^n).
  \]

  (i.e., a clause without a head).

  A query represents a question to the program.

  **Example**: \( \leftarrow \text{pet(X)}. \)
“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.,
    \[ p \leftarrow p_1, \ldots, p_m. \text{ represents } p \leftarrow p_1 \land \cdots \land p_m. \]
  - “\(\leftarrow\)” represents as usual logical implication.

Thus, a rule \( p \leftarrow p_1, \ldots, p_m. \) means “if \( p_1 \) and \( \ldots \) 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”.

“Declarative” Meaning of Predicates and Queries

- **Predicates**: clauses in the same predicate
  
  \[
  p \leftarrow p_1, \ldots, p_n
  \]
  
  \[
  p \leftarrow q_1, \ldots, q_m
  \]
  
  ... provide different *alternatives* (for \( p \)).

  **Example**: the rules
  
  \[
  \text{pet}(X) :- \text{animal}(X), \text{barks}(X).
  \]
  
  \[
  \text{pet}(X) :- \text{animal}(X), \text{meows}(X).
  \]

  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**: 
  
  \[
  :- \text{pet}(<\text{spot}>).
  \]
  
  asks whether <\text{spot}> is a pet. 
  
  \[
  :- \text{pet}(X).
  \]
  
  asks: “Is there an \( X \) which is a pet?”
“Execution” and Semantics

- Example of a logic program:

\[
\begin{align*}
\text{pet}(X) & : - \text{animal}(X), \text{barks}(X). \\
\text{pet}(X) & : - \text{animal}(X), \text{meows}(X).
\end{align*}
\]

\[
\begin{align*}
\text{animal}(\text{spot}). \\
\text{animal}(\text{barry}). \\
\text{animal}(\text{hobbes}).
\end{align*}
\]

\[
\begin{align*}
\text{barks}(\text{spot}). \\
\text{meows}(\text{barry}). \\
\text{roars}(\text{hobbes}).
\end{align*}
\]

- 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 \(- \text{pet}(X).\), the system will try to find a “substitution” for \(X\) which makes \(\text{pet}(X)\) true.

- The declarative semantics specifies what should be computed (all possible answers).
  \[\Rightarrow\] 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).
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:
  ◦ Files should start with the following line:
    :- module(_,_,[’bf/bfall’]).
    or, for “user” files, i.e., files that are not modules: :- use_package(bf).
Ciao Programming Environment: file being edited and top-level
Top Level Interaction Example

- File `pets.pl` contains:
  ```prolog
  :- module(_,_,[bf]).
  + the pet example code as in previous slides.
  ```

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

  ```bash
  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
  ?-
  ```
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* \( p \leftarrow p_1, \ldots, p_m \). means:
  "to execute a call to \( p \) you have to *call* \( p_1 \) and \( \ldots \) and \( p_m \)”
  - In principle, the order in which \( p_1, \ldots, p_n \) are called does not matter, but, in practical systems it is fixed.
- If several clauses (definitions) \( p \leftarrow p_1, \ldots, p_n \) means:
  \( p \leftarrow q_1, \ldots, q_m \)
  "to execute a call to \( p \), call \( p_1 \land \ldots \land p_n \), or, alternatively, \( q_1 \lor \ldots \lor q_n \), or \ldots”
  - 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.
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 is a procedure to solve equations on data structures.
  - As usual, it returns a minimal solution to the equation (or the equation system).
  - As many equation solving procedures it is based on isolating variables and then substituting them by their values.
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** \( \theta \) 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.**:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( \theta )</th>
<th>( A\theta )</th>
<th>( B\theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
<td>dog</td>
<td>( \emptyset )</td>
<td>dog</td>
<td>dog</td>
</tr>
<tr>
<td>X</td>
<td>a</td>
<td>( { X = a } )</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>( { X = Y } )</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>( f(X, g(t)) )</td>
<td>( f(m(h), g(M)) )</td>
<td>( { X = m(h), M = t } )</td>
<td>( f(m(h), g(t)) )</td>
<td>( f(m(h), g(t)) )</td>
</tr>
<tr>
<td>( f(X, g(t)) )</td>
<td>( f(m(h), t(M)) )</td>
<td>Impossible (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( f(X, X) )</td>
<td>( f(Y, l(Y)) )</td>
<td>Impossible (2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (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**.
• Often several solutions exist, e.g.:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta_1$</th>
<th>$A\theta_1$ and $B\theta_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X, g(T))$</td>
<td>$f(m(H), g(M))$</td>
<td>${X=m(a), H=a, M=b, T=b}$</td>
<td>$f(m(a), g(b))$</td>
</tr>
<tr>
<td>$\text{&quot;} \quad \text{&quot;}$</td>
<td>$\text{&quot;} \quad \text{&quot;}$</td>
<td>${X=m(H), M=f(A), T=f(A)}$</td>
<td>$f(m(H), g(f(A)))$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta_1$</th>
<th>$A\theta_1$ and $B\theta_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X, g(T))$</td>
<td>$f(m(H), g(M))$</td>
<td>${X=m(H), T=M}$</td>
<td>$f(m(H), g(M))$</td>
</tr>
</tbody>
</table>

• 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.
Unification Algorithm

- Let $A$ and $B$ be two terms:

1. $\theta = \emptyset$, $E = \{A = B\}$
2. while not $E = \emptyset$:
   1. delete an equation $T = S$ from $E$
   2. case $T$ or $S$ (or both) are (distinct) variables. Assuming $T$ variable:
      * (occur check) if $T$ occurs in the term $S$ → halt with failure
      * substitute variable $T$ by term $S$ in all terms in $\theta$
      * substitute variable $T$ by term $S$ in all terms in $E$
      * add $T = S$ to $\theta$
   3. case $T$ and $S$ are non-variable terms:
      * if their names or arities are different → 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 $\theta$ being the m.g.u of $A$ and $B$
Unification Algorithm Examples (I)

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

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>$E$</th>
<th>$T$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>${ p(X, X) = p(f(Z), f(W)) }$</td>
<td>$p(X, X)$</td>
<td>$p(f(Z), f(W))$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${ X = f(Z), X = f(W) }$</td>
<td>$X$</td>
<td>$f(Z)$</td>
</tr>
<tr>
<td>${ X = f(Z) }$</td>
<td>${ f(Z) = f(W) }$</td>
<td>$f(Z)$</td>
<td>$f(W)$</td>
</tr>
<tr>
<td>${ X = f(Z) }$</td>
<td>${ Z = W }$</td>
<td>$Z$</td>
<td>$W$</td>
</tr>
<tr>
<td>${ X = f(W), Z = W }$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>$E$</th>
<th>$T$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>${ p(X, f(Y)) = p(Z, X) }$</td>
<td>$p(X, f(Y))$</td>
<td>$p(Z, X)$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${ X = Z, f(Y) = X }$</td>
<td>$X$</td>
<td>$Z$</td>
</tr>
<tr>
<td>${ X = Z }$</td>
<td>${ f(Y) = Z }$</td>
<td>$f(Y)$</td>
<td>$Z$</td>
</tr>
<tr>
<td>${ X = f(Y), Z = f(Y) }$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Unification Algorithm Examples (II)

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

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>( E )</th>
<th>( T )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p(X,f(Y)) = p(a,g(b)) }</td>
<td>p(X,f(Y))</td>
<td>p(a,g(b))</td>
</tr>
<tr>
<td>{ }</td>
<td>{ X=a, f(Y)=g(b) }</td>
<td>X</td>
<td>a</td>
</tr>
<tr>
<td>{ X=a }</td>
<td>{ f(Y)=g(b) }</td>
<td>f(Y)</td>
<td>g(b)</td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>( E )</th>
<th>( T )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p(X,f(X)) = p(Z,Z) }</td>
<td>p(X,f(X))</td>
<td>p(Z,Z)</td>
</tr>
<tr>
<td>{ }</td>
<td>{ X=Z, f(X)=Z }</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>{ X=Z }</td>
<td>{ f(Z)=Z }</td>
<td>f(Z)</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\theta$
       (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 $\theta$ 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).
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 step 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):

```prolog
?- pet(X).
X = spot ;
X = barry ;
no
?- pet(X).
X = barry ;
X = spot ;
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 step 2.1.”
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).

\[\text{\vdash pet(P)}\]

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(R)</th>
<th>Clause</th>
<th>(\theta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>(C_2^*)</td>
<td>({P = X_1})</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), meows(X_1)</td>
<td>(C_4^*)</td>
<td>({X_1 = \text{barry}})</td>
</tr>
<tr>
<td>pet(barry)</td>
<td>meows(barry)</td>
<td>(C_7)</td>
<td>{}</td>
</tr>
<tr>
<td>pet(barry)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

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

System response: \(P = \text{barry} \; ?\)

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 \(C_4^*\)).
Running programs (different strategy)

C₁: \( \text{pet}(X) \) \(-\) \( \text{animal}(X) \), \( \text{barks}(X) \).
C₂: \( \text{pet}(X) \) \(-\) \( \text{animal}(X) \), \( \text{meows}(X) \).
C₃: \( \text{animal}(\text{spot}) \).
C₄: \( \text{animal}(\text{barry}) \).
C₅: \( \text{animal}(\text{hobbes}) \).
C₆: \( \text{barks}(\text{spot}) \).
C₇: \( \text{meows}(\text{barry}) \).
C₈: \( \text{roars}(\text{hobbes}) \).

\( \text{:- pet}(P) \). \text{(different strategy)}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\( Q \) & \( R \) & \text{Clause} & \( \theta \) \\
\hline
\( \text{pet}(P) \) & \( \text{pet}(P) \) & \( \text{C₁} \) & \( \{ P = X₁ \} \) \\
\( \text{pet}(X₁) \) & \( \text{animal}(X₁), \text{barks}(X₁) \) & \( \text{C₅} \) & \( \{ X₁ = \text{hobbes} \} \) \\
\( \text{pet}(\text{hobbes}) \) & \( \text{barks(\text{hobbes})} \) & \( ???? \) & \text{failure} \\
\hline
\end{tabular}
\end{center}

\( \rightarrow \) explore another branch (different choice in \( \text{C₁} \) * or \( \text{C₅} \) *) to find a solution.
We take \( \text{C₃} \) instead of \( \text{C₅} \):

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\( Q \) & \( R \) & \text{Clause} & \( \theta \) \\
\hline
\( \text{pet}(P) \) & \( \text{pet}(P) \) & \( \text{C₁} \) & \( \{ P = X₁ \} \) \\
\( \text{pet}(X₁) \) & \( \text{animal}(X₁), \text{barks}(X₁) \) & \( \text{C₃} \) & \( \{ X₁ = \text{spot} \} \) \\
\( \text{pet}(\text{spot}) \) & \( \text{barks(\text{spot})} \) & \( \text{C₆} \) & \( \{ \} \) \\
\( \text{pet}(\text{spot}) \) & \( \text{---} \) & \( \text{---} \) & \( \text{---} \) \\
\hline
\end{tabular}
\end{center}
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]):

```
animal(spot) animal(barry) animal(hobbes)
```

```
animal(hobbes)animal(barry)animal(spot)
```

```
animal(X), barks(X)
```

```
animal(X), meows(X)
```

```
barks(spot)
```

```
meows(barry)
```

• Different query → 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)?
All solutions are at *finite depth* in the tree.

Failures can be at finite depth or, in some cases, be an infinite branch.
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.
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).
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.

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(R)</th>
<th>Clause (\theta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>(C_1^*) {P=X_1}</td>
</tr>
<tr>
<td>pet((X_1))</td>
<td>animal((X_1)), barks((X_1))</td>
<td>(C_3^*) {X_1=\text{spot}}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>(C_6) {}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

• In fact, argument positions are not fixed a priori 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).
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\}

Rules for grandmother_of(X, Y)?
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:
  
  ```prolog
course(complog,wed,19,00,20,30,’M.’,’Hermenegildo’,new,5102).
  ```

- When is the Computational Logic course?
  
  ```prolog
  ```

- Structured version:

  ```prolog
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:

  ```prolog
course(complog, t(wed,18:30,20:30),
    lect(’M.’,’Hermenegildo’), loc(new,5102)).
  ```
Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

  \[
  \text{course(complog,Time,Lecturer, Location)} :- \\
  \quad \text{Time} = t(\text{wed},18:30,20:30), \\
  \quad \text{Lecturer} = \text{lect(‘M.’,’Hermenegildo’),} \\
  \quad \text{Location} = \text{loc(new,5102)}.
  \]

- When is the Computational Logic course?

  \[
  :- \text{course(complog,Time, A, B)}.
  \]

  has solution:

  \[
  \{\text{Time}=t(\text{wed},18:30,20:30), \ A=\text{lect(‘M.’,’Hermenegildo’),} \ B=\text{loc(new,5102)}\}
  \]

- Using the **anonymous variable** (“_”):

  \[
  :- \text{course(complog,Time, _, _)}. 
  \]

  has solution:

  \[
  \{\text{Time}=t(\text{wed},18:30,20:30)\}
  \]
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
Record → Tuple
Field → Attribute
Table
Row
Column

• Example:

Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>M</td>
</tr>
<tr>
<td>Jones</td>
<td>21</td>
<td>F</td>
</tr>
<tr>
<td>Smith</td>
<td>36</td>
<td>M</td>
</tr>
</tbody>
</table>

Lived–in

<table>
<thead>
<tr>
<th>Name</th>
<th>Town</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>London</td>
<td>15</td>
</tr>
<tr>
<td>Brown</td>
<td>York</td>
<td>5</td>
</tr>
<tr>
<td>Jones</td>
<td>Paris</td>
<td>21</td>
</tr>
<tr>
<td>Smith</td>
<td>Brussels</td>
<td>15</td>
</tr>
<tr>
<td>Smith</td>
<td>Santander</td>
<td>5</td>
</tr>
</tbody>
</table>

• 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:**
  
<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>M</td>
</tr>
<tr>
<td>Jones</td>
<td>21</td>
<td>F</td>
</tr>
<tr>
<td>Smith</td>
<td>36</td>
<td>M</td>
</tr>
</tbody>
</table>

- **Example:**
  
<table>
<thead>
<tr>
<th>Name</th>
<th>Town</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>London</td>
<td>15</td>
</tr>
<tr>
<td>Brown</td>
<td>York</td>
<td>5</td>
</tr>
<tr>
<td>Jones</td>
<td>Paris</td>
<td>21</td>
</tr>
<tr>
<td>Smith</td>
<td>Brussels</td>
<td>15</td>
</tr>
<tr>
<td>Smith</td>
<td>Santander</td>
<td>5</td>
</tr>
</tbody>
</table>

Example:

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

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).
The operations of the relational model are easily implemented as rules.

- **Union:**
  \[
  \text{r\_union\_s}(X_1,\ldots,X_n) \leftarrow r(X_1,\ldots,X_n).
  \]
  \[
  \text{r\_union\_s}(X_1,\ldots,X_n) \leftarrow s(X_1,\ldots,X_n).
  \]

- **Set Difference:**
  \[
  \text{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow r(X_1,\ldots,X_n), \text{not } s(X_1,\ldots,X_n).
  \]
  \[
  \text{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow s(X_1,\ldots,X_n), \text{not } r(X_1,\ldots,X_n).
  \]
  
  (we postpone the discussion on negation until later.)

- **Cartesian Product:**
  \[
  \text{r\_X\_s}(X_1,\ldots,X_m,X_{m+1},\ldots,X_{m+n}) \leftarrow r(X_1,\ldots,X_m),s(X_{m+1},\ldots,X_{m+n}).
  \]

- **Projection:**
  \[
  r13(X_1,X_3) \leftarrow r(X_1,X_2,X_3).
  \]

- **Selection:**
  \[
  \text{r\_selected}(X_1,X_2,X_3) \leftarrow r(X_1,X_2,X_3),\leq(X_2,X_3).
  \]
  
  (see later for definition of \(\leq/2\))
Logic Programs and the Relational DB Model (Contd.)

- Derived operations – some can be expressed more directly in LP:
  - Intersection:
    \[
    r \_meet \_s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n).
    \]
  - Join:
    \[
    r \_joinX2 \_s(X_1, \ldots, X_n) \leftarrow r(X_1, X_2, X_3, \ldots, X_n), s(X'_1, X_2, X'_3, \ldots, X'_n).
    \]
- 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:
    ```prolog
    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:
    ```prolog
    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: $0, s(0), s(s(0)), \ldots$
  
  ◦ Type definition:
    
    ```prolog
    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: $0, s(0), -s(0), \ldots$
  
  ◦ Type definition:
    
    ```prolog
    integer( X) :- nat(X).
    integer(-X) :- nat(X).
    ```
Recursive Programming: Arithmetic

• Defining the natural order (≤) of natural numbers:

\[
\text{less\_or\_equal}(0,X) :- \text{nat}(X).
\]
\[
\text{less\_or\_equal}(s(X),s(Y)) :- \text{less\_or\_equal}(X,Y).
\]

• Multiple uses: \text{less\_or\_equal}(s(0),s(s(0))), \text{less\_or\_equal}(X,0),...

• Multiple solutions: \text{less\_or\_equal}(X,s(0)), \text{less\_or\_equal}(s(s(0)),Y), etc.

• Addition:

\[
\text{plus}(0,X,X) :- \text{nat}(X).
\]
\[
\text{plus}(s(X),Y,s(Z)) :- \text{plus}(X,Y,Z).
\]

• Multiple uses: \text{plus}(s(s(0)),s(0),Z), \text{plus}(s(s(0)),Y,s(0))

• Multiple solutions: \text{plus}(X,Y,s(s(s(0)))), etc.
Recursive Programming: Arithmetic

- Another possible definition of addition:
  
  ```prolog
  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 → 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”
  \[(\exists \, Q \text{l.s.t. } X = Y\cdot Q + Z \text{ and } Z < Y):\]
  \[
  \text{mod}(X,Y,Z) \leftarrow \text{less}(Z, Y), \text{times}(Y,Q,W), \text{plus}(W,Z,X).
  \]

  \[
  \text{less}(0,s(X)) \leftarrow \text{nat}(X).
  \]
  \[
  \text{less}(s(X),s(Y)) \leftarrow \text{less}(X,Y).
  \]

- 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).
Recursive Programming: Arithmetic/Functions

- The Ackermann function:
  \[
  \begin{align*}
  \text{ackermann}(0, N) &= N+1 \\
  \text{ackermann}(M, 0) &= \text{ackermann}(M-1, 1) \\
  \text{ackermann}(M, N) &= \text{ackermann}(M-1, \text{ackermann}(M, N-1))
  \end{align*}
  \]

- In Peano arithmetic:
  \[
  \begin{align*}
  \text{ackermann}(0, N) &= s(N) \\
  \text{ackermann}(s(M), 0) &= \text{ackermann}(M, s(0)) \\
  \text{ackermann}(s(M), s(N)) &= \text{ackermann}(M, \text{ackermann}(s(M), N))
  \end{align*}
  \]

- Can be defined as:
  \[
  \begin{align*}
  \text{ackermann}(0, N, s(N)) &. \\
  \text{ackermann}(s(M), 0, \text{Val}) &:- \text{ackermann}(M, s(0), \text{Val}). \\
  \text{ackermann}(s(M), s(N), \text{Val}) &:- \text{ackermann}(s(M), N, \text{Val1}), \\
  &\quad \text{ackermann}(M, \text{Val1}, \text{Val}).
  \end{align*}
  \]

- In general, \textit{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 \textit{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 \([X|Y]\) \((X\ is\ the\ head,\ Y\ is\ the\ tail)\).

<table>
<thead>
<tr>
<th>Formal object</th>
<th>Cons pair syntax</th>
<th>Element syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(.([\ ]))</td>
<td>([a</td>
<td>[\ ]])</td>
</tr>
<tr>
<td>(.([a</td>
<td>([b</td>
<td>[\ ]])]))</td>
</tr>
<tr>
<td>(.([a</td>
<td>([b</td>
<td>([c</td>
</tr>
<tr>
<td>(.([a</td>
<td>X]))</td>
<td>([a</td>
</tr>
<tr>
<td>(.([a</td>
<td>([b</td>
<td>X])))</td>
</tr>
</tbody>
</table>

- Note that:
  - \([a,b]\) and \([a|X]\) unify with \(\{X = [b]\}\)
  - \([a]\) and \([a|X]\) unify with \(\{X = [\]\}\)
  - \([a]\) and \([a,b|X]\) do not unify
  - \([\ ]\) 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).
• X is a *member* of the list Y:

\[
\text{member}(a, [a]). \quad \text{member}(b, [b]). \quad \text{etc.} \quad \Rightarrow \text{member}(X, [X]).
\]
\[
\text{member}(a, [a,c]). \quad \text{member}(b, [b,d]). \quad \text{etc.} \quad \Rightarrow \text{member}(X, [X,Y]).
\]
\[
\text{member}(a, [a,c,d]). \quad \text{member}(b, [b,d,l]). \quad \text{etc.} \quad \Rightarrow \text{member}(X, [X,Y,Z]).
\]

⇒ \text{member}(X, [X|Y]) :- \text{list}(Y).

\[
\text{member}(a, [c,a]), \quad \text{member}(b, [d,b]). \quad \text{etc.} \quad \Rightarrow \text{member}(X, [Y,X]).
\]
\[
\text{member}(a, [c,d,a]), \quad \text{member}(b, [s,t,b]). \quad \text{etc.} \quad \Rightarrow \text{member}(X, [Y,Z,X]).
\]

⇒ \text{member}(X, [Y|Z]) :- \text{member}(X,Z).

• Resulting definition:

\[
\text{member}(X, [X|Y]) :- \text{list}(Y).
\]
\[
\text{member}(X, [-|T]) :- \text{member}(X,T).
\]
Recursive Programming: Lists (Contd.)

- Resulting definition:
  
  \[
  \text{member}(X, [X \mid Y]) \leftarrow \text{list}(Y).
  \text{member}(X, [\_ \mid T]) \leftarrow \text{member}(X, T).
  \]

- Uses of \text{member}(X,Y):
  
  - checking whether an element is in a list (\text{member}(b, [a, b, c]))
  - finding an element in a list (\text{member}(X, [a, b, c]))
  - finding a list containing an element (\text{member}(a, Y))

- Define: \text{prefix}(X,Y) (the list \(X\) is a prefix of the list \(Y\)), e.g.
  \[
  \text{prefix}([a, b], [a, b, c, d])
  \]

- Define: \text{suffix}(X,Y), \text{sublist}(X,Y),…

- Define \text{length}(Xs,N) (N is the length of the list \(Xs\))
Recursive Programming: Lists (Contd.)

- Concatenation of lists:
  
  ◇ Base case:
  
  \[
  \text{append}([], [a], [a]). \quad \text{append}([], [a,b], [a,b]). \quad \text{etc.}
  \]

  \[\Rightarrow \text{append}([], Ys, Ys) :- \text{list}(Ys).\]

  ◇ Rest of cases (first step):
  
  \[
  \text{append}([a], [b], [a,b]).
  \]

  \[
  \text{append}([a], [b,c], [a,b,c]). \quad \text{etc.}
  \]

  \[\Rightarrow \text{append}([X], Ys, [X|Ys]) :- \text{list}(Ys).\]

  \[
  \text{append}([a,b], [c], [a,b,c]).
  \]

  \[
  \text{append}([a,b], [c,d], [a,b,c,d]). \quad \text{etc.}
  \]

  \[\Rightarrow \text{append}([X,Z], Ys, [X,Z|Ys]) :- \text{list}(Ys).\]

  This is still infinite \[\Rightarrow\] we need to generalize more.
Recursive Programming: Lists (Contd.)

- Second generalization:
  \[
  \text{append}([X], Ys, [X|Ys]) :- \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z], Ys, [X,Z|Ys]) :- \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z,W], Ys, [X,Z,W|Ys]) :- \text{list}(Ys).
  \]
  \[
  \Rightarrow \text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs).
  \]

- So, we have:
  \[
  \text{append}([], Ys, Ys) :- \text{list}(Ys).
  \]
  \[
  \text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs).
  \]

- Uses of append:
  - concatenate two given lists: \(\text{:- append}([a,b], [c], Z)\)
  - find differences between lists: \(\text{:- append}(X, [c], [a,b,c])\)
  - split a list: \(\text{:- append}(X, Y, [a,b,c])\)
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:

  \[
  \text{reverse}([X|Xs],Ys) : - \\
  \text{reverse}(Xs,Zs), \\
  \text{append}(Zs,[X],Ys).
  \]

  How can we stop?

  `reverse([],[]).`

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

  \[
  \text{reverse}(Xs,Ys) : - \text{reverse}(Xs,[],Ys).
  \]

  `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)) :-
    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).
```
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 LP

- In Prolog:
  \[
  T = \text{tree}(3, \text{tree}(2,\text{void},\text{void}), \text{tree}(5,\text{void},\text{void}))
  \]

- In Pascal:

```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;
...```

```pascal```
Polymorphism

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

  ```prolog
  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 (somewat surprising): :- lt_member(M,T).
  ```
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

```prolog
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):

  ![Automaton Diagram](image)

  where \( q_0 \) 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):

  ```prolog
  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.

![Diagram of towers of Hanoi](image_url)
Recursive Programming: Towers of Hanoi (Contd.)

- We will call the main predicate \( \text{hanoi \_ moves}(N, \text{Moves}) \)
- \( N \) is the number of disks and \( \text{Moves} \) the corresponding list of “moves”.
- Each move \( \text{move}(A, B) \) represents that the top disk in \( A \) should be moved to \( B \).
- \textbf{Example}:

  \[
  \text{hanoi\_moves( } s(s(s(0))), \[
  [ \text{move}(a,b), \text{move}(a,c), \text{move}(b,c), \text{move}(a,b), \[
  \text{move}(c,a), \text{move}(c,b), \text{move}(a,b) ])
  \]
  \]
A general rule:

We capture this in a predicate \( \text{hanoi}(N, \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) \) where “Moves contains the moves needed to move a tower of \( N \) disks from peg \( \text{Orig} \) to peg \( \text{Dest} \), with the help of peg \( \text{Help} \).”

\[
\text{hanoi}(s(0), \text{Orig}, \text{Dest}, _, \text{Help}, [\text{move}(\text{Orig}, \text{Dest})]).
\]

\[
\text{hanoi}(s(N), \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) :-
\text{hanoi}(N, \text{Orig}, \text{Help}, \text{Dest}, \text{Moves1}),
\text{hanoi}(N, \text{Help}, \text{Dest}, \text{Orig}, \text{Moves2}),
\text{append}(\text{Moves1}, [\text{move}(\text{Orig}, \text{Dest})|\text{Moves2}], \text{Moves}).
\]

And we simply call this predicate:

\[
\text{hanoi_moves}(N, \text{Moves}) :-
\text{hanoi}(N, a, b, c, \text{Moves}).
\]
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.