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!):

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Main operator:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>is the term</td>
<td>’+’(a,b)</td>
</tr>
<tr>
<td>- b</td>
<td>is the term</td>
<td>’-’(b)</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>is the term</td>
<td>’&lt;’(a,b)</td>
</tr>
<tr>
<td>john father mary</td>
<td>is the term</td>
<td>father(john,mary)</td>
</tr>
</tbody>
</table>

We assume that some such operator definitions are always preloaded.
Syntax: Rules and Facts (Clauses)

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

  \[ p_0(t_1, t_2, \ldots, t_{n_0}) \leftarrow p_1(t^1_1, t^1_2, \ldots, t^1_{n_1}), \]
  \[ \ldots \]
  \[ p_m(t^m_1, t^m_2, \ldots, t^m_{n_m}). \]

  - \( p_0(\ldots) \) 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*.
  - Usually, \( \leftarrow \) is called the *neck* of the rule.

- **Fact:** an expression of the form \( p(t_1, t_2, \ldots, t_n) \).

  (i.e., a rule with empty body).

*Example:*

<table>
<thead>
<tr>
<th>expression</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>meal(soup, beef, coffee).</td>
<td>% \leftarrow A fact.</td>
</tr>
<tr>
<td>meal(First, Second, Third) :-</td>
<td>% \leftarrow A rule.</td>
</tr>
<tr>
<td>appetizer(First),</td>
<td>%</td>
</tr>
<tr>
<td>main_dish(Second),</td>
<td>%</td>
</tr>
<tr>
<td>dessert(Third).</td>
<td>%</td>
</tr>
</tbody>
</table>

- Rules and facts are both called *clauses*. 


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

  **Examples:**
  
  ```
  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. 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:**  \[ \leftarrow \text{pet}(X) . \]

  In most systems written as:  \[ ?- \text{pet}(X) . \]
“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
  
  \[
  p \leftarrow p_1, \ldots, p_n \\
  p \leftarrow q_1, \ldots, q_m \\
  \ldots
  \]

  provide different *alternatives* (for \( p \)).

  *Example*: the rules

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

  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:*

  \[
  \begin{align*}
  \text{?- pet(spot).} & \\
  \text{?- pet}(X). &
  \end{align*}
  \]

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

- Example of a logic program:

```prolog
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 = spot and X = barry.
  ◦ The operational semantics specifies how answers are computed (which allows us to determine how many steps it will take).
Running Programs in a Logic Programming System

- File `pets.pl` contains (explained later):

```prolog
:- module(_,_,['bf/bfall']).
```

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

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

```prolog
?- Ciao 1.XX ...
?- use_module(pets).
yes
?- pet(spot).
yes
?- pet(X).
X = spot ? ;
X = barry ? ;
nono
?-  
```

See the part on Developing Programs with a Logic Programming System for more details on the particular system used in the course (Ciao).
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) \( p \leftarrow p_1, \ldots, p_n \), \( p \leftarrow q_1, \ldots, q_n \) means:
  "to execute a call to \( p \), call \( p_1 \land \ldots \land p_n \), or, alternatively, \( q_1 \land \ldots \land q_n \), 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 *instantiating* them with 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>( f(m(h), g(t)) )</td>
<td>( f(m(h), g(t)) )</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*. (See, however, *cyclic terms* later.)
Unification

- 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>&quot;&quot;</td>
<td>&quot;&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 \( E = \emptyset \):
   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 \( S \) → 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 \( \theta \)
   2.3 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>{}</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>{}</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>{}</td>
<td></td>
<td></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>{}</td>
<td>{ p(X, f(Y)) = p(Z, X) }</td>
<td>p(X, f(Y))</td>
<td>p(Z, X)</td>
</tr>
<tr>
<td>{}</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>{}</td>
<td></td>
<td></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>
</tbody>
</table>

\textit{fail}

- 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>
</tbody>
</table>

\textit{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 $\theta$
       (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 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
?- 
```

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.”
Running programs

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

• :- pet(P).

<table>
<thead>
<tr>
<th>Q</th>
<th>R</th>
<th>Clause</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>C₂*</td>
<td>{P = X₁}</td>
</tr>
<tr>
<td>pet(X₁)</td>
<td>animal(X₁), meows(X₁)</td>
<td>C₄*</td>
<td>{X₁ = barry}</td>
</tr>
<tr>
<td>pet(barry)</td>
<td>meows(barry)</td>
<td>C₇</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 = 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 C₂* or C₄*).
Running programs (different strategy)

\[C_1: \quad \text{pet}(X) :- \text{animal}(X), \text{barks}(X).\]
\[C_2: \quad \text{pet}(X) :- \text{animal}(X), \text{meows}(X).\]
\[C_3: \quad \text{animal}(\text{spot}).\]
\[C_4: \quad \text{animal}(\text{barry}).\]
\[C_5: \quad \text{animal}(\text{hobbes}).\]
\[C_6: \quad \text{barks}(\text{spot}).\]
\[C_7: \quad \text{meows}(\text{barry}).\]
\[C_8: \quad \text{roars}(\text{hobbes}).\]

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

\[Q \quad R \quad \text{Clause} \quad \theta\]
\[\text{pet}(P) \quad \text{pet}(P) \quad C_1^* \quad \{P = X_1\}\]
\[\text{pet}(X_1) \quad \text{animal}(X_1), \text{barks}(X_1) \quad C_5^* \quad \{X_1 = \text{hobbes}\}\]
\[\text{pet}(\text{hobbes}) \quad \text{barks}(\text{hobbes}) \quad ??? \quad \text{failure}\]

\[\rightarrow \text{ explore another branch (different choice in } C_1^* \text{ or } C_5^* \text{) to find a solution.}\]
\[\text{We take } C_3 \text{ instead of } C_5:\]

\[\begin{array}{|c|c|c|c|}
\hline
Q & R & \text{Clause} & \theta \\
\hline
\text{pet}(P) & \text{pet}(P) & C_1^* & \{P = X_1\} \\
\text{pet}(X_1) & \text{animal}(X_1), \text{barks}(X_1) & C_3^* & \{X_1 = \text{spot}\} \\
\text{pet(\text{spot})} & \text{barks(\text{spot})} & C_6 & \{} \\
\text{pet(\text{spot})} & \text{---} & \text{---} & \text{---} \\
\hline
\end{array}\]
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]):

- 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)?
• 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 \texttt{bf} package).
Selecting breadth-first or depth-first search

- In the Ciao system we can select the search rule using the packages mechanism.

- Files should start with the following line:
  - To execute in *breadth-first* mode:
    ```prolog
    :- module(_,_,[’bf/bfail’]).
    ```
  - To execute in *depth-first* mode:
    ```prolog
    :- module(_,_,[]).
    ```

See the part on Developing Programs with a Logic Programming System for more details on the particular system used in the course (Ciao).
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</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>C₁ *</td>
<td>{ P=X₁ }</td>
</tr>
<tr>
<td>pet(X₁)</td>
<td>animal(X₁), barks(X₁)</td>
<td>C₃ *</td>
<td>{ X₁=spot }</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>C₆</td>
<td>{}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>—</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 `:- plus(s(0),s(s(0)),Z).` vs. `:- plus(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).
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).
yes
?- father_of(john, david).
no
?- father_of(john, X).
X = peter ;
X = mary

?- grandfather_of(X, michael).
X = john
?- grandfather_of(X, Y).
X = john, Y = michael ;
X = john, Y = david
?- grandfather_of(X, X).
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:

  ```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:**

  \[
  \text{course(complog,Time,Lecturer, Location) :-}
  \]
  \[
  \begin{align*}
  \text{Time} & = t(\text{wed},18:30,20:30), \\
  \text{Lecturer} & = \text{lect('M.','Hermenegildo')}, \\
  \text{Location} & = \text{loc(new,5102)}. 
  \end{align*}
  \]

- **When is the Computational Logic course?**

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

  has solution:

  \[
  \begin{align*}
  \text{Time} & = t(\text{wed},18:30,20:30), \\
  \text{A} & = \text{lect('M.','Hermenegildo')}, \\
  \text{B} & = \text{loc(new,5102)}. 
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \text{course(complog,Time, _, _).}
  \end{align*}
  \]

  has solution:

  \[
  \begin{align*}
  \text{Time} & = t(\text{wed},18:30,20:30)
  \end{align*}
  \]
Terms as Data Structures with Pointers

- **main** below is a procedure, that:
  - creates some data structures, with *pointers* and *aliasing*.
  - calls other procedures, passing pointers to these structures to them.

```
main :-
    X=f(K, g(K)),
    Y=a,
    Z=g(L),
    W=h(b, L),
    % Heap memory at this point →
    p(X, Y),
    q(Y, Z),
    r(W).
```

- Terms are data structures with pointers.
- Logical variables are *declarative* pointers.
  - Declarative: they can only be assigned once.
Structured Data and Data Abstraction (Contd.)

- The circuit example revisited:

  \[
  \text{resistor}(r1, \text{power}, n1). \quad \text{transistor}(t1, n2, \text{ground}, n1). \hfill \\
  \text{resistor}(r2, \text{power}, n2). \quad \text{transistor}(t2, n3, n4, n2). \hfill \\
  \text{transistor}(t3, n5, \text{ground}, n4). \hfill \\
  \text{inverter}(\text{inv}(T, R), \text{Input}, \text{Output}) \leftarrow \hfill \\
  \quad \text{transistor}(T, \text{Input}, \text{ground}, \text{Output}), \hfill \\
  \quad \text{resistor}(R, \text{power}, \text{Output}). \hfill \\
  \]

  \[
  \text{nand\_gate}(\text{nand}(T1, T2, R), \text{Input1}, \text{Input2}, \text{Output}) \leftarrow \hfill \\
  \quad \text{transistor}(T1, \text{Input1}, X, \text{Output}), \hfill \\
  \quad \text{transistor}(T2, \text{Input2}, \text{ground}, X), \hfill \\
  \quad \text{resistor}(R, \text{power}, \text{Output}). \hfill \\
  \]

  \[
  \text{and\_gate}(\text{and}(N, I), \text{Input1}, \text{Input2}, \text{Output}) \leftarrow \hfill \\
  \quad \text{nand\_gate}(N, \text{Input1}, \text{Input2}, X), \hfill \\
  \quad \text{inverter}(I, X, \text{Output}). \hfill \\
  \]

- The query \( \text{:- and\_gate}(G, \text{In1}, \text{In2}, \text{Out}) \) has solution: \( G = \text{and}(\text{nand}(t2, t3, r2), \text{inv}(t1, r1)), \text{In1} = n3, \text{In2} = n5, \text{Out} = n1 \)
Logic Programs and the Relational DB Model

<table>
<thead>
<tr>
<th>Relational Database</th>
<th>Logic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation Name</td>
<td>→ Predicate symbol</td>
</tr>
<tr>
<td>Relation</td>
<td>→ Procedure consisting of ground facts</td>
</tr>
<tr>
<td></td>
<td>(facts without variables)</td>
</tr>
<tr>
<td>Tuple</td>
<td>→ Ground fact</td>
</tr>
<tr>
<td>Attribute</td>
<td>→ Argument of predicate</td>
</tr>
</tbody>
</table>

<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>

**“Person”**

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

<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>

**“Lived in”**

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

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

- **Set Difference:**
  
  \[ \text{r} \text{\_diff\_s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, \ldots, X_n), \text{not} \text{\_s}(X_1, \ldots, X_n). \]

  \[ \text{r} \text{\_diff\_s}(X_1, \ldots, X_n) \leftarrow \text{s}(X_1, \ldots, X_n), \text{not} \text{\_r}(X_1, \ldots, X_n). \]

  (we postpone the discussion on negation until later.)

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

- **Projection:**
  
  \[ \text{r}\_1\_3(X_1, X_3) \leftarrow \text{r}(X_1, X_2, X_3). \]

- **Selection:**
  
  \[ \text{r}\_\text{selected}(X_1, X_2, X_3) \leftarrow \text{r}(X_1, X_2, X_3), \leq(X_2, X_3). \]

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

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

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

  - **Join:**
    
    \[ \text{r} \text{\_join\_x2\_s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, X_2, X_3, \ldots, X_n), \text{s}(X_1', X_2, X_3', \ldots, X_n'). \]

- 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.
  - Powerful notions of negation supported: S-models → Answer Set Programming. ASP).
Recursive Programming

- Example: ancestors.

```prolog
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:

```prolog
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
    weekday('Monday').
    weekday('Tuesday'). ...
    ```
- **Example**: Date (weekday * day in the month):
  - Set of terms to represent: date('Monday',23), date(Tuesday,24), ...
  - Type definition:
    ```prolog
    date(date(W,D)) :- weekday(W), day_of_month(D).
    day_of_month(1).
    day_of_month(2).
    ...
    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:

    \[
    \begin{align*}
    \text{nat}(0) & , \\
    \text{nat}(s(X)) & : = \text{nat}(X). \\
    \end{align*}
    \]

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

- Types are *runnable* and can be used to check or produce values:
  - $?- \text{nat}(X) \Rightarrow X=0; X=s(0); X=s(s(0)); \ldots$

- We can reason about *complexity*, for a given *class of queries* (“mode”).
  E.g., for mode $\text{nat}(\text{ground})$ complexity is *linear* in size of number.

- **Example**: integers:
  - Set of terms to represent: $0$, $s(0)$, $-s(0)$, ...
  
  - Type definition:

    \[
    \begin{align*}
    \text{integer}( X) & : = \text{nat}(X) , \\
    \text{integer}(-X) & : = \text{nat}(X). \\
    \end{align*}
    \]
Recursive Programming: Arithmetic

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

```prolog
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:

```prolog
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.
```
• Another possible definition of addition:

\[
\begin{align*}
\text{plus}(X, 0, X) & :\text{- nat}(X). \\
\text{plus}(X, s(Y), s(Z)) & :\text{- plus}(X, Y, Z).
\end{align*}
\]

• The meaning of \text{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: \text{times}(X, Y, Z) (Z = X \times Y), \text{exp}(N, X, Y) (Y = X^N), \text{factorial}(N, F) (F = N!), \text{minimum}(N1, N2, Min), ...
Recursive Programming: Arithmetic

- Definition of \texttt{mod}(X, Y, Z)
  
  “Z is the remainder from dividing X by Y”

  \[ \exists Q \text{ s.t. } X = Y \times Q + Z \land Z < Y \]

  \[ \Rightarrow \]

  \texttt{mod}(X, Y, Z) :- \ \texttt{less}(Z, Y), \ \texttt{times}(Y, Q, W), \ \texttt{plus}(W, Z, X).

- Another possible definition:
  \texttt{mod}(X, Y, X) :- \ \texttt{less}(X, Y).
  \texttt{mod}(X, Y, Z) :- \ \texttt{plus}(X_1, Y, X), \ \texttt{mod}(X_1, 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:

  \[
  \text{ackermann}(0,N) = N+1 \\
  \text{ackermann}(M,0) = \text{ackermann}(M-1,1) \\
  \text{ackermann}(M,N) = \text{ackermann}(M-1,\text{ackermann}(M,N-1))
  \]

- In Peano arithmetic:

  \[
  \text{ackermann}(0,N) = s(N) \\
  \text{ackermann}(s(M1),0) = \text{ackermann}(M1,s(0)) \\
  \text{ackermann}(s(M1),s(N1)) = \text{ackermann}(M1,\text{ackermann}(s(M1),N1))
  \]

- Can be defined as:

  \[
  \text{ackermann}(0,N,s(N)). \\
  \text{ackermann}(s(M1),0,Val) :- \text{ackermann}(M1,s(0),Val). \\
  \text{ackermann}(s(M1),s(N1),Val) :- \text{ackermann}(s(M1),N1,Val1), \text{ackermann}(M1,Val1,Val).
  \]

- In general, *functions* can be coded as a predicate with one more argument, which represents the output (and additional syntactic sugar often available).
Recursive Programming: Arithmetic/Functions (Functional Syntax)

- Syntactic support available (see, e.g., the Ciao **fsyntax** and **functional** packages).

- The Ackermann function (Peano) in Ciao’s functional Syntax and defining \( s \) as a prefix operator:

\[
\begin{align*}
\text{:- use_package(functional).} \\
\text{:- op(500,fy,s).} \\
\text{ackermann( 0, N) := s N.} \\
\text{ackermann(s M, 0) := ackermann(M, s 0).} \\
\text{ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).}
\end{align*}
\]

- Convenient in other cases – e.g. for defining types:

\[
\begin{align*}
\text{nat(0).} \\
\text{nat(s(X) :- nat(X).}
\end{align*}
\]

Using special := notation for the “return” (last) the argument:

\[
\begin{align*}
\text{nat := 0.} \\
\text{nat := s(X) :- nat(X).}
\end{align*}
\]
Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the \( \sim \) notation (“evaluate and replace with result”):

\[
\begin{align*}
\text{nat} & := 0. \\
\text{nat} & := \text{s}(\sim \text{nat}).
\end{align*}
\]

“\( \sim \)” not needed with functional package if inside its own definition:

\[
\begin{align*}
\text{nat} & := 0. \\
\text{nat} & := \text{s}(\text{nat}).
\end{align*}
\]

Using an \texttt{:- op(500, fy, s)} declaration to define \texttt{s} as a \textit{prefix operator}:

\[
\begin{align*}
\text{nat} & := 0. \\
\text{nat} & := \text{s nat}.
\end{align*}
\]

Using “\( | \)” (disjunction):

\[
\begin{align*}
\text{nat} & := 0 \mid \text{s nat}.
\end{align*}
\]

Which exactly equivalent to:

\[
\begin{align*}
\text{nat}(0). \\
\text{nat}(\text{s}(X)) & := \text{nat}(X).
\end{align*}
\]
Recursive Programming: Lists

- Binary structure: first argument is *element*, second argument is *rest* of the list.
- We need:
  - A constant symbol: we use the constant \([\ ]\) (→ denotes the empty list).
  - 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>((a,[]))</td>
<td>([a</td>
<td>[]])</td>
</tr>
<tr>
<td>((a,.(b,[])))</td>
<td>([a</td>
<td>[b</td>
</tr>
<tr>
<td>((a,.(b,(c,[]))))</td>
<td>([a</td>
<td>[b</td>
</tr>
<tr>
<td>((a,X))</td>
<td>([a</td>
<td>X])</td>
</tr>
<tr>
<td>((a,.(b,X)))</td>
<td>([a</td>
<td>[b</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 (Contd.)

- Type definition (no syntactic sugar):
  
  ```
  list([],).
  list([X,Y]) :- list(Y).
  ```

- Type definition, with some syntactic sugar ([ ] notation):

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

- Type definition, using also functional package:

  ```
  list := [] | [X|list].
  ```

- “Exploring” the type:

  ```
  ?- list(L).
  L = [] ? ;
  L = [X] ? ;
  L = [X,X] ? ;
  L = [X,X,X] ? ;
  L = [X,X,X,X] ? ;
  ...
  ```
Recursive Programming: Lists (Contd.)

- X is a member of the list Y:
  
  \[
  \begin{align*}
  &\text{member}(a,[a]). \quad \text{member}(b,[b]). \quad \text{etc.} \quad \Rightarrow \text{member}(X,[X]). \\
  &\text{member}(a,[a,c]). \quad \text{member}(b,[b,d]). \quad \text{etc.} \quad \Rightarrow \text{member}(X,[X,Y]). \\
  &\text{member}(a,[a,c,d]). \quad \text{member}(b,[b,d,l]). \quad \text{etc.} \quad \Rightarrow \text{member}(X,[X,Y,Z]).
  \end{align*}
  \]

  \[\Rightarrow \text{member}(X,[X|Y]) \leftarrow \text{list}(Y).\]

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

  \[\Rightarrow \text{member}(X,[Y|Z]) \leftarrow \text{member}(X,Z).\]

- Resulting definition:

  \[
  \begin{align*}
  &\text{member}(X,[X|Y]) \leftarrow \text{list}(Y). \\
  &\text{member}(X,[\_|T]) \leftarrow \text{member}(X,T).
  \end{align*}
  \]

- 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))
• Combining lists and naturals:

  ◇ Computing the length of a list:

  ```prolog
  len([], 0).
  len([H|T], s(LT)) :- len(T, LT)
  ```

  ◇ Adding all elements of a list:

  ```prolog
  sumlist([], 0).
  sumlist([H|T], S) :- sumlist(T, ST), plus(ST, H, S).
  ```

  ◇ The type of lists of natural numbers:

  ```prolog
  natlist([], 0).
  natlist([H|T]) :- natlist(T, ST), nat(ST, H, S).
  ```

  or:

  ```prolog
  natlist := [nat | natlist].
  ```
Recursive Programming: Lists (Contd.)

- Exercises:
  - 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: \( \text{suffix}(X, Y) \), \( \text{sublist}(X, Y) \), ...
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.
Recursive Programming: Lists (Contd.)

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

- So, we have:

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

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

- Another way of reasoning: thinking inductively.

  ◦ The base case is:

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

  ◦ If we assume that \text{append(Zs, Ys, Zs)} works for some iteration, then, in the next one, the following holds: \text{append(s(Zs), Ys, s(Zs))}. 
Recursive Programming: Lists (Contd.)

- Uses of append:
  - Concatenate two given lists:
    \[\text{?- append}([a,b,c],[d,e],L).}\]
    \[L = [a,b,c,d,e]?\]
  - Find differences between lists:
    \[\text{?- append}(D,[d,e],[a,b,c,d,e]).}\]
    \[D = [a,b,c]?\]
  - Split a list:
    \[\text{?- append}(A,B,[a,b,c,d,e]).}\]
    \[A = [],\]
    \[B = [a,b,c,d,e]?;\]
    \[A = [a],\]
    \[B = [b,c,d,e]?;\]
    \[A = [a,b],\]
    \[B = [c,d,e]?;\]
    \[A = [a,b,c],\]
    \[B = [d,e]?;\]
    \[...\]
• 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:

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

How can we stop?

\[
\text{reverse}([], []). 
\]

• As defined, reverse(Xs, Ys) is very inefficient. Another possible definition: (uses an *accumulating parameter*)

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

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

⇒ 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:

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

- Defining `tree_member(Element,Tree)`:  

  ```prolog
  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,Elements)`: Elements is a list containing the elements of Tree traversed in *preorder*.

```
pre_order(void,[]).
pre_order(tree(X,Left,Right),Elements) :-
    pre_order(Left,ElementsLeft),
    pre_order(Right,ElementsRight),
    append([X|ElementsLeft],ElementsRight,Elements).
```

• Exercise – define:
  ◦ `in_order(Tree,Elements)`
  ◦ `post_order(Tree,Elements)`
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).
Recognizing (and generating!) 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).
```
Symbolic differentiation: deriv(Expression, X, DifferentiatedExpression)

```prolog
deriv(X,X,s(0)).
derv(C,X,0) :- pconstant(C).
derv(U+V,X,DU+DV) :- deriv(U,X,DU), deriv(V,X,DV).
derv(U-V,X,DU-DV) :- deriv(U,X,DU), deriv(V,X,DV).
derv(U*V,X,DU*V+U*DV) :- deriv(U,X,DU), deriv(V,X,DV).
derv(U/V,X,(DU*V-U*DV)/V^s(s(0))) :- deriv(U,X,DU), deriv(V,X,DV).
derv(U^s(N),X,s(N)*U^N*DU) :- deriv(U,X,DU), nat(N).
derv(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.
Recognizing the sequence of characters accepted by the following *non-deterministic, finite automaton* (NDFA):

where \texttt{q0} is both the *initial* and the *final* state.

Strings are represented as lists of constants (e.g., \texttt{[a,b,b]}).

Program:

\begin{verbatim}
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).
\end{verbatim}
A nondeterministic, stack, finite automaton (NDSFA):

\[
\text{accept}(S) \leftarrow \text{initial}(Q), \text{accept}\_\text{from}(S,Q,[]) .
\]

\[
\text{accept}\_\text{from}([],Q,[]) \leftarrow \text{final}(Q) .
\]

\[
\text{accept}\_\text{from}([X|Xs],Q,S) \leftarrow \text{delta}(Q,X,S,NewQ,NewS), \\
\quad \text{accept}\_\text{from}(Xs,NewQ,NewS) .
\]

\[
\text{initial}(q0) .
\]

\[
\text{final}(q1) .
\]

\[
\text{delta}(q0,X,Xs,q0,[X|Xs]) .
\]

\[
\text{delta}(q0,X,Xs,q1,[X|Xs]) .
\]

\[
\text{delta}(q0,X,Xs,q1,Xs) .
\]

\[
\text{delta}(q1,X,[X|Xs],q1,Xs) .
\]

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:**

![Diagram of the Towers of Hanoi]

is represented by:

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

![Diagram of the Towers of Hanoi puzzle]

• 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).
```
Learning to Compose Recursive Programs

- To some extent it is a simple question of practice.
- By generalization (as in the previous examples): elegant, but sometimes difficult? (Not the way most people do it.)
- Think inductively: 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 then also if alternative uses make sense.
- Sometimes it helps to look at well-written examples and use the same “schemas.”
- Using a global top-down design approach can help (in general, not just for recursive programs):
  ◦ State the general problem.
  ◦ Break it down into subproblems.
  ◦ Solve the pieces.
- Again, the best approach: practice, practice, practice.