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>Exp</th>
<th>Term</th>
<th>Type</th>
<th>Main functor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>is the term</td>
<td>’+’(a, b)</td>
<td>if +/-2 declared infix</td>
</tr>
<tr>
<td>- b</td>
<td>is the term</td>
<td>’-’(b)</td>
<td>if -/1 declared prefix</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>is the term</td>
<td>’&lt;’(a, b)</td>
<td>if &lt;/2 declared infix</td>
</tr>
<tr>
<td>john father mary</td>
<td>is the term</td>
<td>father(john, mary)</td>
<td>if father/2 declared infix</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:

  \[
  p_0(t_1, t_2, \ldots, t_{n_0}) \leftarrow p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1), \ldots, p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m).
  \]

  ◦ \(p_0(\ldots)\) 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**.
  ◦ Usually, \(-\) 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:*

  | meal(soup, beef, coffee).       | % \leftarrow A fact. |
  | meal(First, Second, Third) :-  | % \leftarrow A rule.  |
  | 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:*

  
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(spot)</td>
<td>animal(spot)</td>
</tr>
<tr>
<td>pet(X) :- animal(X), barks(X)</td>
<td>animal(barry)</td>
</tr>
<tr>
<td>pet(X) :- animal(X), meows(X)</td>
<td>animal(hobbes)</td>
</tr>
</tbody>
</table>

  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:* `:- pet(X).`

  In most systems written as: `?- 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.,
    
    \( p \leftarrow p_1, \ldots, p_m \) 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
  
  ```prolog
  pet(X) :- animal(X), barks(X).
  pet(X) :- animal(X), meows(X).
  ```

  express two ways for \(X\) to be a pet.

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

- **A query** represents a *question to the program*.

  *Examples*:
  
  ```prolog
  ?- pet(spot).
  ?- pet(X).
  ```

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

- Example of a logic program:

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

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

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

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

  ◦ The **declarative semantics** specifies *what* should be computed (all possible answers).
    ⇒ Intuitively, we have two possible answers: `X = 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 ;
      no
  ?- 
  ```

  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) \( p \leftarrow p_1, \ldots, p_n \) means:
  \( p \leftarrow q_1, \ldots, q_m \)
  "to execute a call to \( p \), call \( p_1 \land \ldots \land p_n \), or, alternatively, \( q_1 \land \ldots \land q_n \), 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;</td>
<td>&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 $S \rightarrow$ 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 $\rightarrow$ 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 $\theta$ 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>{}</td>
<td>$p(X,X)$</td>
<td>$p(f(Z), f(W))$</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>$X = f(Z), X = f(W)$</td>
<td>$X$</td>
</tr>
<tr>
<td>{$X = f(Z)$}</td>
<td>{}</td>
<td>$f(Z) = f(W)$</td>
<td>$f(Z)$</td>
</tr>
<tr>
<td>{$X = f(Z)$}</td>
<td>{}</td>
<td>$Z = W$</td>
<td>$Z$</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>{}</td>
<td>$p(X, f(Y))$</td>
<td>$p(Z, X)$</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>$X = Z, f(Y) = X$</td>
<td>$X$</td>
</tr>
<tr>
<td>{$X = Z$}</td>
<td>{}</td>
<td>$f(Y) = Z$</td>
<td>$f(Y)$</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>
<tr>
<td>fail</td>
<td></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>fail</td>
<td></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 failed; 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).
A (Schematic) Interpreter for Logic Programs (Contd.)

- Since step 2.2 is left open, a given logic programming system must specify how it deals with this by providing one (or more)
  - **Search rule(s):** “how are clauses/branches selected in 2.2.”

- If the search rule is not specified execution can be *nondeterministic*, since choosing a different clause (in step 2.2) could lead to different solutions (finding solutions in a different order).

*Example* (two valid executions):

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

- 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

\( C_1 \):  pet(X) :- animal(X), barks(X).

\( C_2 \):  pet(X) :- animal(X), meows(X).

\( C_3 \):  animal(spot).

\( C_4 \):  animal(barry).

\( C_5 \):  animal(hobbes).

\( C_6 \):  barks(spot).

\( C_7 \):  meows(barry).

\( C_8 \):  roars(hobbes).

\( :- \) pet(P).

<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>( C_8 )</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)

\( C_1: \) 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).

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

\[
\begin{array}{|c|c|c|c|}
\hline
Q & R & \text{Clause} & \theta \\
\hline
\text{pet(P)} & \text{pet(P)} & C_1^* & \{P = X_1\} \\
\text{pet(X)} & \text{animal(X)} \land \text{barks(X)} & C_5^* & \{X_1 = \text{hobbes}\} \\
\text{pet(hobbes)} & \text{barks(hobbes)} & \text{???} & \text{failure} \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|c|c|}
\hline
Q & R & \text{Clause} & \theta \\
\hline
\text{pet(P)} & \text{pet(P)} & C_1^* & \{P = X_1\} \\
\text{pet(X)} & \text{animal(X)} \land \text{barks(X)} & C_3^* & \{X_1 = \text{spot}\} \\
\text{pet(spot)} & \text{barks(spot)} & C_6 & \{\} \\
\text{pet(spot)} & \therefore & \therefore & \therefore \\
\hline
\end{array}
\]
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)?
Characterization of The Search Tree

- All solutions are at *finite depth* in the tree.
- Failures can be at finite depth or, in some cases, be an infinite branch.
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).
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/bfall’]).
    ```
  - 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

• As mentioned before, unification used to *access data* and *give values to variables*.  
  
  *Example:* Consider query 
  
  \[ \text{:- } \text{animal(A), named(A,Name).} \] 
  
  with: 
  
  \[ \text{animal(dog(barry)). named(dog(Name),Name).} \] 

• 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>$\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>$C_1^*$</td>
<td>{ P=X_1 }</td>
</tr>
<tr>
<td>pet($X_1$)</td>
<td>animal($X_1$), barks($X_1$)</td>
<td>$C_3^*$</td>
<td>{ $X_1$=spot }</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>$C_6$</td>
<td>{}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
“Modes”

• In fact, argument positions are not fixed a priori to be input or output.

   Example: Consider query \( \text{:- pet(spot).} \) vs. \( \text{:- pet(X).} \)

   or \( \text{:- plus( s(0), s(s(0)), Z).} \) % Adds

   vs. \( \text{:- plus( s(0), Y, s(s(s(0))))}. \) % Subtracts

• Thus, procedures can be used in different **modes** s.t. different sets of arguments are input or output in each mode.

• We sometimes use \( + \) and \( - \) to refer to, respectively, and argument being an input or an output, e.g.:

   \( \text{plus(} +X, +Y, -Z) \) means we call \( \text{plus} \) with

   ◦ \( X \) instantiated,

   ◦ \( Y \) instantiated, and

   ◦ \( Z \) free.
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)?
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,18,30,20,30,’M.’,’Hermenegildo’,new,5102).

- When is the Computational Logic course?


- Structured version:

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

**Note:** “X=Y” is equivalent to “’=(X,Y)” where the predicate =/2 is defined as the fact “’=(X,X).” – Plain unification!

- Equivalent to:

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

- Given:

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

- When is the Computational Logic course?

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

has solution:

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

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

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

has solution:

```
Time=t(wed,18:30,20:30)
```
Terms as Data Structures with Pointers

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

```
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:

```prolog
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

  ```prolog```

  ```prolog
  :- and_gate(G,In1,In2,Out).
  ```

  has solution:

  ```prolog
  G=and(nand(t2,t3,r2),inv(t1,r1)), In1=n3, In2=n5, 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"

<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"

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).
Logic Programs and the Relational DB Model (Contd.)

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

  - **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 $\leq/2$)

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

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

  - **Join**: 
    
    \[
    \text{r\_join\_x2\_s}(X_1, \ldots, X_n) \leftarrow r(X_1, X_2, X_3, \ldots, X_n), s(X'_1, X_2, X'_3, \ldots, X'_n).
    \]

- Duplicates an issue: see “setof” later in Prolog.
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)
  → powerful knowledge representation and reasoning systems.
Recursive Programming

• Example: ancestors.

\[
\text{parent}(X,Y) :- \text{father}(X,Y).
\]

\[
\text{parent}(X,Y) :- \text{mother}(X,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{parent}(Z,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{parent}(Z,W), \text{parent}(W,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{parent}(Z,W), \text{parent}(W,K), \text{parent}(K,Y).
\]

• Defining ancestor recursively:

\[
\text{parent}(X,Y) :- \text{father}(X,Y).
\]

\[
\text{parent}(X,Y) :- \text{mother}(X,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{ancestor}(Z,Y).
\]

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{ancestor}(Z,Y).
\]

• 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:
    - weekday('Monday').
    - weekday('Tuesday'). ...

- **Example**: Date (weekday * day in the month):
  - Set of terms to represent: date('Monday',23), date('Tuesday',24), ...
  - Type definition:
    - 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: \(0, s(0), s(s(0)), \ldots\)
  - 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: \(0, s(0), -s(0), \ldots\)
  - 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:

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

  ◇ Multiple uses (modes):

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

  ◇ Multiple solutions:

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

- Addition:

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

  ◇ Multiple uses (modes):

  \[
  \text{plus}(s(s(0)), s(0), Z), \text{plus}(s(s(0)), Y, s(0))
  \]

  ◇ Multiple solutions:

  \[
  \text{plus}(X, Y, s(s(s(0)))), \text{etc.}
  \]
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)` \( (Z = X \times Y) \), \( \text{exp}(N,X,Y) \) \( (Y = X^N) \), \( \text{factorial}(N,F) \) \( (F = N!) \), \( \text{minimum}(N1,N2,Min) \), ...
Recursive Programming: Arithmetic

- Definition of \( \text{mod}(X, Y, Z) \)
  “Z is the remainder from dividing X by Y”
  \[ \exists Q \text{s.t. } X = Y \times Q + Z \land Z < Y \]
  \[ \Rightarrow \]

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

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

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

- Another possible definition:

  \[
  \text{mod}(X, Y, X) :- \ \text{less}(X, Y).
  \]

  \[
  \text{mod}(X, Y, Z) :- \ \text{plus}(X1, Y, X), \ \text{mod}(X1, Y, Z).
  \]

- 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:

\[
\begin{align*}
\text{ackermann}(0, N) &= s(N) \\
\text{ackermann}(s(M1), 0) &= \text{ackermann}(M1, s(0)) \\
\text{ackermann}(s(M1), s(N1)) &= \text{ackermann}(M1, \text{ackermann}(s(M1), N1))
\end{align*}
\]

• Can be defined as:

\[
\begin{align*}
\text{ackermann}(0, N, s(N)). \\
\text{ackermann}(s(M1), 0, \text{Val}) &:= \text{ackermann}(M1, s(0), \text{Val}). \\
\text{ackermann}(s(M1), s(N1), \text{Val}) &:= \text{ackermann}(s(M1), N1, \text{Val1}), \\
&\quad \text{ackermann}(M1, \text{Val1}, \text{Val}).
\end{align*}
\]

• 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:

```prolog
:- use_package(functional).
:- op(500, fy, s).

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

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

```prolog
nat(0).
nat(s(X)) :- nat(X).
```

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

```prolog
nat := 0.
nat := s(X) :- nat(X).
```
Moving body call to head using the ~ notation ("evaluate and replace with result"):

```prolog
nat := 0.
nat := s(~nat).
```

"~" not needed with functional package if inside its own definition:

```prolog
nat := 0.
nat := s(nat).
```

Using an `:- op(500, fy, s)` declaration to define s as a *prefix operator*:

```prolog
nat := 0.
nat := s nat.
```

Using "|" (disjunction):

```prolog
nat := 0 | s nat.
```

Which is exactly equivalent to:

```prolog
nat(0).
nat(s(X) :- nat(X).
```
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* \([\ ]\) (\(\rightarrow\) denotes the empty list).
  - A functor of arity 2: traditionally the dot “.” (which is overloaded).

- Syntactic sugar: the term \((X,Y)\) is denoted by \([X\mid Y]\) (\(X\) 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\mid[]])</td>
<td>([a])</td>
</tr>
<tr>
<td>(.a,.(b,[]))</td>
<td>([a\mid[b\mid[]]])</td>
<td>([a,b])</td>
</tr>
<tr>
<td>(.a,.((b,.(c,[]))))</td>
<td>([a\mid[b\mid[c\mid[]]]])</td>
<td>([a,b,c])</td>
</tr>
<tr>
<td>(.a,X)</td>
<td>([a\mid X])</td>
<td>([a\mid X])</td>
</tr>
<tr>
<td>(.a,.(b,X))</td>
<td>([a\mid[b\mid X]])</td>
<td>([a,b\mid X])</td>
</tr>
</tbody>
</table>

- Note that:
  - \([a,b]\) and \([a\mid X]\) unify with \(\{X = [b]\}\)
  - \([a]\) and \([a\mid X]\) unify with \(\{X = [\]\}\)
  - \([a]\) and \([a,b\mid X]\) do not unify
  - \([\]\) and \([X]\) do not unify
Recursive Programming: Lists (Contd.)

- Type definition (no syntactic sugar):

```prolog
list([]).
list.(X,Y)) :- list(Y).
```

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

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

- Type definition, using also functional package:

```prolog
list := [] | [_|list].
```

- “Exploring” the type:

```prolog
?- list(L).
L = [] ? ;
L = [_] ? ;
L = [_,_] ? ;
L = [_,_,_] ? ;
L = [_,_,_,_] ? ;
...
```
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).

- 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:
  
  \[
  \text{len}([], 0).
  \text{len}([H|T], s(LT)) :- \text{len}(T, LT)
  \]

- Adding all elements of a list:
  
  \[
  \text{sumlist}([], 0).
  \text{sumlist}([H|T], S) :- \text{sumlist}(T, ST), \text{plus}(ST, H, S).
  \]

- The type of lists of natural numbers:
  
  \[
  \text{natlist}([], 0).
  \text{natlist}([H|T]) :- \text{natlist}(T, ST), \textbf{nat}(ST, H, S).
  \]

  or:
  
  \[
  \text{natlist} := [\textbf{^\_nat}|\text{natlist}].
  \]
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:
    
    \[
    \text{append([], [a], [a]). append([], [a,b], [a,b]). etc.}
    \]
    
    \[
    \Rightarrow \text{append([], Ys, Ys) :- list(Ys).}
    \]

  - Rest of cases (first step):
    
    \[
    \text{append([a], [b], [a,b]). append([a], [b,c], [a,b,c]). etc.}
    \]
    
    \[
    \Rightarrow \text{append([X], Ys, [X|Ys]) :- list(Ys).}
    \]
    
    \[
    \text{append([a,b], [c], [a,b,c]). append([a,b], [c,d], [a,b,c,d]). etc.}
    \]
    
    \[
    \Rightarrow \text{append([X,Z], Ys, [X,Z|Ys]) :- list(Ys).}
    \]

This is still infinite → we need to generalize more.
Recursive Programming: Lists (Contd.)

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

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

- Another way of reasoning: thinking inductively.
  ✷ The base case is: \text{append}([], Ys, Ys) :- \text{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:
    ```prolog
    ?- append([a,b,c],[d,e],L).
    L = [a,b,c,d,e] ?
    ```
  - Find differences between lists:
    ```prolog
    ?- append(D,[d,e],[a,b,c,d,e]).
    D = [a,b,c] ?
    ```
  - Split a list:
    ```prolog
    ?- 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] ?
    ...
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:

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

  How can we stop?

  ```prolog
  reverse([], []).
  ```

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

  ```prolog
  reverse(Xs, Ys) :- reverse(Xs, [], Ys).
  ```

  ```prolog
  reverse([], Ys, Ys).
  ```

  ```prolog
  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:

```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.
  
  ```prolog
  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).
literal_member(X, [_|T]) :- lt_member(X, T).
```

```
lt_member(X, tree(X,L,R)) :- binary_tree(L), binary_tree(R).
literal_member(X, tree(Y,L,R)) :- lt_member(X, L).
literal_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).`
Recursive Programming: Manipulating Symbolic Expressions

- 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

```
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: \( \text{deriv}(\text{Expression}, \ X, \ \text{DifferentiatedExpression}) \)

  \[
  \begin{align*}
  \text{deriv}(X, X, s(0)) & . \\
  \text{deriv}(C, X, 0) & : \text{pconstant}(C). \\
  \text{deriv}(U + V, X, DU + DV) & : \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV). \\
  \text{deriv}(U - V, X, DU - DV) & : \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV). \\
  \text{deriv}(U \cdot V, X, DU \cdot V + U \cdot DV) & : \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV). \\
  \text{deriv}(U/V, X, (DU \cdot V - U \cdot DV)/V^{s(s(0)))} & : \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV). \\
  \text{deriv}(U^{s(N)}, X, s(N) \cdot U^{\cdot N} \cdot DU) & : \text{deriv}(U, X, DU), \ \text{nat}(N). \\
  \text{deriv}(\log(U), X, DU/U) & : \text{deriv}(U, X, DU). \\
  \ldots
  \end{align*}
  \]

- \( \text{deriv}(s(s(s(0))) \cdot x + s(s(0)), x, Y) \).

- A simplification step can be added.
Recursive Programming: Automata (Graphs)

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

  \[ q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0 \]

  where \( q_0 \) is both the *initial* and the *final* state.

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

- Program:
  
  ```prolog
  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).
  ```
A *nondeterministic, stack, finite automaton* (NDSFA):

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

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

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

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

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

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

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

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

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

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 showing the progression of moving disks from peg a to peg b through peg c for N=1, N=2, and N=3 cases.](image-url)
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:**

```
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) ])
```

is represented by:

![Diagram of the Towers of Hanoi with seven disks and moves highlighted in red.](Image)
Recursive Programming: Towers of Hanoi (Contd.)

- 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 \( \text{Help} \)."

\[
\begin{align*}
\text{hanoi}(s(0), \text{Orig}, \text{Dest}, \_\text{Help}, [\text{move}(\text{Orig}, \text{Dest})]). \\
\text{hanoi}(s(N), \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) & : - \\
& \quad \text{hanoi}(N, \text{Orig}, \text{Help}, \text{Dest}, \text{Moves1}), \\
& \quad \text{hanoi}(N, \text{Help}, \text{Dest}, \text{Orig}, \text{Moves2}), \\
& \quad \text{append} (\text{Moves1}, [\text{move}(\text{Orig}, \text{Dest}) | \text{Moves2}], \text{Moves}).
\end{align*}
\]

- And we simply call this predicate:

\[
\begin{align*}
\text{hanoi\textunderscore moves}(N, \text{Moves}) & : - \\
& \quad \text{hanoi}(N, a, b, c, \text{Moves}).
\end{align*}
\]
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.