Computational Logic

A “Hands-on” Introduction to Pure Logic Programming
Syntax: Terms (Variables, Constants, and Structures)

(using Prolog notation conventions)

- **Variables:** start with uppercase character (or “_”), may include “_” and digits:
  
  _Examples_: X, Im4u, A_little_garden, _, _x, _22

- **Constants:** lowercase first character, may include “_” and digits. Also, numbers and some special characters. Quoted, any character:
  
  _Examples_: a, dog, a_big_cat, 23, ’Hungry man’, []

- **Structures:** a **functor** (the structure name, is like a constant name) followed by a fixed number of arguments between parentheses:
  
  _Example_: date(monday, Month, 1994)

Arguments can in turn be variables, constants and structures.

- **Arity:** is the number of arguments of a structure. Functors are represented as *name/arity*. A constant can be seen as a structure with arity zero.

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

(Using Prolog notation conventions)

- Examples of terms:

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

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

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

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

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

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

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

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

  **Example:**
  ```plaintext
  meal(soup, beef, coffee) \leftarrow .
  meal(First, Second, Third) \leftarrow
      appetizer(First),
      main_dish(Second),
      dessert(Third).
  ```

- Rules and facts are both called clauses.
Syntax: Predicates, Programs, and Queries

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

  *Examples:*

  \[
  \text{pet}(\text{spot}) \leftarrow . \\
  \text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X). \\
  \text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X). \\
  \text{animal}(\text{spot}) \leftarrow . \\
  \text{animal}(\text{barry}) \leftarrow . \\
  \text{animal}(\text{hobbes}) \leftarrow .
  \]

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

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

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

  A query represents a *question to the program*.  

  *Example*: \( \leftarrow \text{pet}(X) \).
“Declarative” Meaning of Facts and Rules

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

- **Facts**: state things that are true.
  (Note that a fact “p ← .” can be seen as the rule “p ← true.”)
  
  *Example*: the fact `animal(spot) ← .` can be read as “spot is an animal”.

- **Rules**:  
  - Commas in rule bodies represent conjunction, i.e.,
    
    \[ p ← p_1, \ldots, p_m. \text{ represents } p ← p_1 \land \cdots \land p_m. \]
  
  - “←” represents as usual logical implication.

  Thus, a rule \[ p ← p_1, \ldots, p_m. \] means “if \( p_1 \) and \( \ldots \) and \( p_m \) are true, then \( p \) is true”

  *Example*: the rule `pet(X) ← animal(X), barks(X).` can be read as “\( X \) is a pet if it is an animal and it barks”.
“Declarative” Meaning of Predicates and Queries

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

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

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

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

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

  **Examples**:
  
  \[
  \text{- pet(spot).}
  \]
  
  \[
  \text{- pet}(X).
  \]

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

- **Example of a logic program:**
  
  \[
  \begin{align*}
  \text{pet}(X) & \leftarrow \text{animal}(X), \text{barks}(X). \\
  \text{pet}(X) & \leftarrow \text{animal}(X), \text{meows}(X). \\
  \text{animal}(\text{spot}) & \leftarrow. \\
  \text{barks}(\text{spot}) & \leftarrow. \\
  \text{animal}(\text{barry}) & \leftarrow. \\
  \text{meows}(\text{barry}) & \leftarrow. \\
  \text{animal}(\text{hobbes}) & \leftarrow. \\
  \text{roars}(\text{hobbes}) & \leftarrow. 
  \end{align*}
  \]

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

  *Example:* given the program above and the query \( \leftarrow \text{pet}(X) \).
  the system will try to find a “substitution” for \( X \) which makes \( \text{pet}(X) \) true.

  ◦ The **declarative semantics** specifies *what* should be computed (all possible answers).
    ⇒ Intuitively, we have two possible answers: \( X = \text{spot} \) and \( X = \text{barry} \).

  ◦ The **operational semantics** specifies *how* answers are computed (which allows us to determine *how many steps* it will take).
Running Pure Logic Programs: the Ciao System’s bf/af Packages

- We will be using Ciao, a multiparadigm programming system which includes (as one of its “paradigms”) a pure logic programming subsystem:
  - A number of fair search rules are available (breadth-first, iterative deepening, ...): we will use “breadth-first” (bf or af).
  - Also, a module can be set to pure mode so that impure built-ins are not accessible to the code in that module.
  - This provides a reasonable first approximation of “Greene’s dream” (of course, at a cost in memory and execution time).

- Writing programs to execute in bf mode:
  - All files should start with the following line:
    ```prolog
    :- module(_,_,[bf]).  (or :- module(_,_,['bf/af']).)
    ```
    or, for “user” files, i.e., files that are not modules: :- use_package(bf).
  - The neck (arrow) of rules must be `<-`.
  - Facts must end with `<-`.
Ciao Programming Environment: file being edited and top-level
Top Level Interaction Example

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

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

  ```prolog
  Ciao 1.13 #0: Mon Nov 7 09:48:51 MST 2005
  ?- use_module(pets).
    yes
  ?- pet(spot).
    yes
  ?- pet(X).
    X = spot ? ;
    X = barry ? ;
    no
  ?-
  ```
Simple (Top-Down) Operational Meaning of Programs

- A logic program is operationally a set of *procedure definitions* (the predicates).
- A query \( \leftarrow p \) is an initial *procedure call*.
- A procedure definition with one *clause* \( p \leftarrow p_1, \ldots, p_m \) means:
  “to execute a call to \( p \) you have to *call* \( p_1 \) and \( \ldots \) and \( p_m \)”
  - In principle, the order in which \( p_1, \ldots, p_n \) are called does not matter, but, in practical systems it is fixed.
- If several clauses (definitions) \( p \leftarrow p_1, \ldots, p_n \) means:
  \( p \leftarrow q_1, \ldots, q_m \)
  “to execute a call to \( p \), call \( p_1 \wedge \ldots \wedge p_n \), or, alternatively, \( q_1 \wedge \ldots \wedge q_n \), or ...”
  - 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

- **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, \textit{fail}).
  - Only variables can be given values!
  - Two structures can be made identical only by making their arguments identical.

\[ \text{E.g.:} \]

<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 dog</td>
<td>dog dog</td>
<td>( \emptyset )</td>
<td>dog dog</td>
<td>dog dog</td>
</tr>
<tr>
<td>( X ) a</td>
<td>( X = a )</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>( X ) Y</td>
<td>( X = Y )</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>( f(X, g(t)) ) ( f(m(h), g(M)) ) ( {X=m(h), M=t} )</td>
<td>( f(m(h), g(t)) ) ( f(m(h), g(t)) )</td>
<td>( \text{Impossible (1)} )</td>
<td>( \text{Impossible (1)} )</td>
<td></td>
</tr>
<tr>
<td>( f(X, g(t)) ) ( f(m(h), t(M)) )</td>
<td>( \text{Impossible (2)} )</td>
<td>( \text{Impossible (2)} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unification

- Often several solutions exist, e.g.:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( \theta_1 )</th>
<th></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></th>
<th></th>
<th>( \theta_1 )</th>
<th></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>${ p(X,X) = p(f(Z),f(W)) }$</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>
<tr>
<td><strong>fail</strong></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><strong>fail</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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):

```prolog
?- pet(X).
X = spot ? ;
X = barry ? ;
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) <-.

* 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 C₂* or C₄*).
Running programs (different strategy)

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). (different strategy)

<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>C₅*</td>
<td>{X₁ = hobbes}</td>
</tr>
<tr>
<td>pet(hobbes)</td>
<td>barks(hobbes)</td>
<td>???</td>
<td>failure</td>
</tr>
</tbody>
</table>

→ explore another branch (different choice in C₁* or C₅*) to find a solution. We take C₃ instead of C₅:

<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>
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 → 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).
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:
  
  \[ \text{animal(dog(barry))} <- . \quad \text{named(dog(Name),Name)} <- . \]

- 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>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=\text{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>

- In fact, argument positions are not fixed a priori to be input or output. **Example:** Consider query `<- pet(spot).` vs. `<- pet(X).` or `<- add(s(0),s(s(0)),Z).` vs. `<- add(s(0),Y,s(s(s(0))))`.

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

- A Logic Database is a set of facts and rules (i.e., a logic program):

\[
\begin{align*}
\text{father_of}(\text{john}, \text{peter}) & \leftarrow. \\
\text{father_of}(\text{john}, \text{mary}) & \leftarrow. \\
\text{father_of}(\text{peter}, \text{michael}) & \leftarrow. \\
\text{mother_of}(\text{mary}, \text{david}) & \leftarrow. \\
\text{grandfather_of}(L,M) & \leftarrow \text{father_of}(L,N), \text{father_of}(N,M). \\
\text{grandfather_of}(X,Y) & \leftarrow \text{father_of}(X,Z), \text{mother_of}(Z,Y).
\end{align*}
\]

- Given such database, a logic programming system can answer questions (queries) such as:

\[
\begin{align*}
\leftarrow \text{father_of}(\text{john}, \text{peter}). \\
\text{Answer: } \text{Yes} \\
\leftarrow \text{father_of}(\text{john}, \text{david}). \\
\text{Answer: } \text{No} \\
\leftarrow \text{father_of}(\text{john}, X). \\
\text{Answer: } \{X = \text{peter}\} \\
\text{Answer: } \{X = \text{mary}\} \\
\leftarrow \text{grandfather_of}(\text{john}, \text{michael}). \\
\text{Answer: } \{X = \text{john}\} \\
\leftarrow \text{grandfather_of}(\text{john}, Y). \\
\text{Answer: } \{X = \text{john}, Y = \text{michael}\} \\
\text{Answer: } \{X = \text{john}, Y = \text{david}\} \\
\leftarrow \text{grandfather_of}(\text{john}, X). \\
\text{Answer: } \text{No}
\end{align*}
\]

- Rules for \text{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,19,00,20,30,’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:

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

• When is the Computational Logic course?

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

has solution:

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

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

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

has solution:

\[
\{\text{Time}=t(\text{wed}, 18:30, 20:30)\}\]
Structured Data and Data Abstraction (Contd.)

- The circuit example revisited:

  ```
  resistor(r1,power,n1) <-.
  transistor(t1,n2,ground,n1) <-.
  resistor(r2,power,n2) <-.
  transistor(t2,n3,n4,n2) <-.
  transistor(t3,n5,ground,n4) <-.
  inverter(inv(T,R),Input,Output) <-
    transistor(T,Input,ground,Output), resistor(R,power,Output).
  nand_gate(nand(T1,T2,R),Input1,Input2,Output) <-
    transistor(T1,Input1,X,Output), transistor(T2,Input2,ground,X),
    resistor(R,power,Output).
  and_gate(and(N,I),Input1,Input2,Output) <-
    nand_gate(N,Input1,Input2,X), inverter(I,X,Output).
  ```

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

Traditional → Codd's Relational Model
- File → Relation
- Record → Tuple
- Field → Attribute

Table
Row
Column

Example:

<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>

<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>

Person

Lived-in

- The order of the rows is immaterial.
- (Duplicate rows are not allowed)
Logic Programs and the Relational DB Model (Contd.)

<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 (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>

- **Example:**
  
  person(brown,20,male) <-.
  person(jones,21,female) <-.
  person(smith,36,male) <-.

- **Example:**
  
  lived_in(brown,london,15) <-.
  lived_in(brown,york,5) <-.
  lived_in(jones,paris,21) <-.
  lived_in(smith,brussels,15) <-.
  lived_in(smith,santander,5) <-.

<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>

<table>
<thead>
<tr>
<th>Name</th>
<th>Town</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>London</td>
<td>15</td>
</tr>
<tr>
<td>Brown</td>
<td>York</td>
<td>5</td>
</tr>
<tr>
<td>Jones</td>
<td>Paris</td>
<td>21</td>
</tr>
<tr>
<td>Smith</td>
<td>Brussels</td>
<td>15</td>
</tr>
<tr>
<td>Smith</td>
<td>Santander</td>
<td>5</td>
</tr>
</tbody>
</table>
The operations of the relational model are easily implemented as rules.

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

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

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

- **Projection:**
  
  \[
  r_{13}(X_1, X_3) \leftarrow r(X_1, X_2, X_3).
  \]

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

• Derived operations – some can be expressed more directly in LP:
  
  ◦ Intersection:
    
    $$r \text{ \_meet\_s}(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n).$$
  
  ◦ Join:
    
    $$r \text{ \_joinX2\_s}(X_1, \ldots, X_n) \leftarrow r(X_1, X_2, X_3, \ldots, X_n), s(X'_1, X_2, X'_3, \ldots, X'_n).$$

• Duplicates an issue: see “setof” later in Prolog.
Deductive Databases

The subject of “deductive databases” uses these ideas to develop *logic-based databases*.

- Often syntactic restrictions (a subset of definite programs) used (e.g. “Datalog” – no functors, no existential variables).
- Variations of a “bottom-up” execution strategy used: Use the $T_p$ operator (explained in the theory part) to compute the model, restrict to the query.
Recursive Programming

- **Example**: ancestors.
  
  ```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:
  
  ```
  is_weekday('Monday') <-.
  is_weekday('Tuesday') <-. ...
  ```

**Example**: Date (weekday * day in the month):

- Set of terms to represent: date('Monday',23), date(Tuesday,24), ...
- Type definition:
  
  ```
  is_date(date(W,D)) <- is_weekday(W), is_day_of_month(D).
  is_day_of_month(1) <-.
  is_day_of_month(2) <-.
  ...
  is_day_of_month(31) <-.  ```
Recursive Programming: Recursive Types

- **Recursive types**: defined by recursive logic programs.
- **Example**: natural numbers (simplest recursive data type):
  - Set of terms to represent: \(0, s(0), s(s(0)), \ldots\)
  - Type definition:
    - \(\text{nat}(0) \leftarrow .\)
    - \(\text{nat}(s(X)) \leftarrow \text{nat}(X).\)

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

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

- **Example**: integers:
  - Set of terms to represent: \(0, s(0), -s(0), \ldots\)
  - Type definition:
    - \(\text{integer}(X) \leftarrow \text{nat}(X).\)
    - \(\text{integer}(-X) \leftarrow \text{nat}(X).\)
Recursive Programming: Arithmetic

- Defining the natural order ($\leq$) of natural numbers:
  
  less_or_equal(0,X) <- nat(X).
  less_or_equal(s(X),s(Y)) <- less_or_equal(X,Y).

- Multiple uses: less_or_equal(s(0),s(s(0))), less_or_equal(X,0), ...

- Multiple solutions: less_or_equal(X,s(0)), less_or_equal(s(s(0)),Y), etc.

- Addition:
  
  plus(0,X,X) <- nat(X).
  plus(s(X),Y,s(Z)) <- plus(X,Y,Z).

- Multiple uses: plus(s(s(0)),s(0),Z), plus(s(s(0)),Y,s(0))

- Multiple solutions: plus(X,Y,s(s(s(0)))), etc.
Recursive Programming: Arithmetic

- Another possible definition of addition:
  
  \[
  \text{plus}(X,0,X) \leftarrow \text{nat}(X).
  \]
  
  \[
  \text{plus}(X,s(Y),s(Z)) \leftarrow \text{plus}(X,Y,Z).
  \]

- The meaning of \text{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: \text{times}(X,Y,Z) \ (Z = X \times Y), \ \text{exp}(N,X,Y) \ (Y = X^N),

  \[
  \text{factorial}(N,F) \ (F = N!), \ \text{minimum}(N1,N2,Min), \ldots
  \]
Recursive Programming: Arithmetic

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

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

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

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

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

- The second is much more efficient than the first one (compare the size of the proof trees).
Recursive Programming: Arithmetic/Functions

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

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

- Can be defined as:
  
  \[ \text{ackermann}(0,N,s(N)) \leftarrow . \]
  \[ \text{ackermann}(s(M),0,\text{Val}) \leftarrow \text{ackermann}(M,s(0),\text{Val}). \]
  \[ \text{ackermann}(s(M),s(N),\text{Val}) \leftarrow \text{ackermann}(s(M),N,\text{Val1}), \]
  \[ \text{ackermann}(M,\text{Val1},\text{Val}). \]

- In general, \textit{functions} can be coded as a predicate with one more argument, which represents the output (and additional syntactic sugar often available).

- Syntactic support available (see, e.g., the Ciao \textit{functions} package).
Recursive Programming: Lists

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

<table>
<thead>
<tr>
<th>Formal object</th>
<th>Cons pair syntax</th>
<th>Element syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>((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

• Type definition (no syntactic sugar):
  \[
  \text{list}([], []) <- .
  \text{list}([], (X, Y)) <- \text{list}(Y).
  \]

• Type definition (with syntactic sugar):
  \[
  \text{list}([], []) <- .
  \text{list}([], [X|Y]) <- \text{list}(Y).
  \]
Recursive Programming: Lists (Contd.)

- X is a member of the list Y:

  \[
  \begin{align*}
  \text{member}(a, [a]) & \leftarrow. \text{member}(b, [b]) \leftarrow. \text{etc.} \quad \Rightarrow \text{member}(X, [X]) \leftarrow. \\
  \text{member}(a, [a, c]) & \leftarrow. \text{member}(b, [b, d]) \leftarrow. \text{etc.} \quad \Rightarrow \text{member}(X, [X, Y]) \leftarrow. \\
  \text{member}(a, [a, c, d]) & \leftarrow. \text{member}(b, [b, d, 1]) \leftarrow. \text{etc.} \quad \Rightarrow \text{member}(X, [X, Y, Z]) \leftarrow. \\
  \Rightarrow \text{member}(X, [X|Y]) & \leftarrow \text{list}(Y). \\
  \end{align*}
  \]

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

- Resulting definition:
  \[
  \begin{align*}
  \text{member}(X, [X|Y]) & \leftarrow \text{list}(Y). \\
  \text{member}(X, [\_|T]) & \leftarrow \text{member}(X, T). \\
  \end{align*}
  \]
Recursive Programming: Lists (Contd.)

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

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

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

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

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

- Concatenation of lists:
  - Base case:
    ```prolog
    append([], [a], [a]) <-. append([], [a, b], [a, b]) <-. etc.
    ⇒ append([], Ys, Ys) <- list(Ys).
    ```
  - Rest of cases (first step):
    ```prolog
    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:
  \[
  \text{append([X],Ys,[X\mid Ys])} \leftarrow \text{list(Ys)}. \\
  \text{append([X,Z],Ys,[X,Z\mid Ys])} \leftarrow \text{list(Ys)}. \\
  \text{append([X,Z,W],Ys,[X,Z,W\mid Ys])} \leftarrow \text{list(Ys)}. \\
  \Rightarrow \text{append([X\mid Xs],Ys,[X\mid Zs])} \leftarrow \text{append(Xs,Ys,Zs)}. 
  \]

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

• Uses of append:
  ◦ concatenate two given lists: \(\leftarrow \text{append([a,b],[c],Z)}\)
  ◦ find differences between lists: \(\leftarrow \text{append(X,[c],[a,b,c])}\)
  ◦ split a list: \(\leftarrow \text{append(X,Y,[a,b,c])}\)
Recursive Programming: Lists (Contd.)

- \texttt{reverse(Xs,Ys)}: \texttt{Ys} is the list obtained by reversing the elements in the list \texttt{Xs}
  - It is clear that we will need to traverse the list \texttt{Xs}
  - For each element \texttt{X} of \texttt{Xs}, we must put \texttt{X} at the end of the rest of the \texttt{Xs} list already reversed:
    \begin{verbatim}
    reverse([X|Xs],Ys <-
    reverse(Xs,Zs),
    append(Zs,[X],Ys).
    \end{verbatim}
  - How can we stop?
    \texttt{reverse([],[])} <-.

- As defined, \texttt{reverse(Xs,Ys)} is very inefficient. Another possible definition:
  \begin{verbatim}
  reverse(Xs,Ys) <- reverse(Xs,[],Ys).
  \end{verbatim}

  \begin{verbatim}
  reverse([],Ys,Ys) <-.
  reverse([X|Xs],Acc,Ys) <- reverse(Xs,[X|Acc],Ys).
  \end{verbatim}

- 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:
  
  \[
  \begin{align*}
  \text{binary_tree(void)} & \leftarrow . \\
  \text{binary_tree(tree(Element,Left,Right))} & \leftarrow \\
  & \quad \text{binary_tree(Left)}, \\
  & \quad \text{binary_tree(Right)}. \\
  \end{align*}
  \]

- Defining `tree_member(Element,Tree)`:
  
  \[
  \begin{align*}
  \text{tree_member}(X,\text{tree}(X,\text{Left},\text{Right})) & \leftarrow \\
  & \quad \text{binary_tree(Left)}, \\
  & \quad \text{binary_tree(Right)}. \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  \text{tree_member}(X,\text{tree}(Y,\text{Left},\text{Right})) & \leftarrow \text{tree_member}(X,\text{Left}). \\
  \text{tree_member}(X,\text{tree}(Y,\text{Left},\text{Right})) & \leftarrow \text{tree_member}(X,\text{Right}). \\
  \end{align*}
  \]
Recursive Programming: Binary Trees

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

- Define `in_order(Tree,Order)`, `post_order(Tree,Order)`. 
Creating a Binary Tree in Pascal and LP

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

- In Pascal:

```pascal
type tree = ^treerec;
  treerec = record
    data : integer;
    left : tree;
    right: tree;
  end;

var t : tree;
```

```pascal
...new(t);
new(t^left);
new(t^right);
t^left^left := nil;
t^left^right := nil;
t^right^left := nil;
t^right^right := nil;
t^data := 3;
t^left^data := 2;
t^right^data := 5;
...```
Polymorphism

- Note that the two definitions of member/2 can be used \textit{simultaneously}:

\begin{verbatim}
lt_member(X,[X|Y]) <- list(Y).
ltd_member(X,[_|T]) <- ltd_member(X,T).

lt_member(X,tree(X,L,R)) <- binary_tree(L), binary_tree(R).
ltd_member(X,tree(Y,L,R)) <- ltd_member(X,L).
ltd_member(X,tree(Y,L,R)) <- ltd_member(X,R).
\end{verbatim}

Lists only unify with the first two clauses, trees with clauses 3–5!

- \texttt{<- lt_member(X,[b,a,c]).}
  
  \texttt{X = b ; X = a ; X = c}

- \texttt{<- lt_member(X,tree(b,tree(a,void,void),tree(c,void,void))).}
  
  \texttt{X = b ; X = a ; X = c}

- Also, try (somewhat surprising): \texttt{<- lt_member(M,T).}
Recursive Programming: Manipulating Symbolic Expressions

- Recognizing polynomials in some term X:
  ◦ X is a polynomial in X
  ◦ a constant is a polynomial in X
  ◦ sums, differences and products of polynomials in X are polynomials
  ◦ also polynomials raised to the power of a natural number and the quotient of a polynomial by a constant

polynomial(X,X) <-.
polynomial(Term,X) <- pconstant(Term).
polynomial(Term1+Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1-Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1*Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1/Term2,X) <- polynomial(Term1,X), pconstant(Term2).
polynomial(Term1^N,X) <- polynomial(Term1,X), nat(N).
Recursive Programming: Manipulating Symb. Expressions (Contd.)

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

  ```prolog
  deriv(X,X,s(0)) <-.
  deriv(C,X,0) <- pconstant(C).
  deriv(U+V,X,DU+DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U-V,X,DU-DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U*V,X,DU*V+U*DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U/V,X,(DU*V-U*DV)/V^s(s(0))) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U^s(N),X,s(N)*U^N*DU) <- deriv(U,X,DU), nat(N).
  deriv(log(U),X,DU/U) <- deriv(U,X,DU).
  ...
  
  <- deriv(s(s(s(0)))*x+s(s(0)),x,Y).
  
  A simplification step can be added.
  ```
Recognizing the sequence of characters accepted by the following non-deterministic, finite automaton (NDFA):

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

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

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

Program:

\[
\begin{align*}
\text{initial}(q_0) & \leftarrow. \\
\text{delta}(q_0,a,q_1) & \leftarrow. \\
\text{delta}(q_1,b,q_0) & \leftarrow. \\
\text{final}(q_0) & \leftarrow. \\
\text{delta}(q_1,b,q_1) & \leftarrow. \\
\text{accept}(S) & \leftarrow \text{initial}(Q), \text{accept}_{\text{from}}(S,Q). \\
\text{accept}_{\text{from}}([],Q) & \leftarrow \text{final}(Q). \\
\text{accept}_{\text{from}}([X|Xs],Q) & \leftarrow \text{delta}(Q,X,\text{NewQ}), \text{accept}_{\text{from}}(Xs,\text{NewQ}).
\end{align*}
\]
A nondeterministic, stack, finite automaton (NDSFA):

\[
\begin{align*}
\text{accept}(S) & \leftarrow \text{initial}(Q), \text{accept}_\text{from}(S,Q,[]). \\
\text{accept}_\text{from}([],Q,[]) & \leftarrow \text{final}(Q). \\
\text{accept}_\text{from}([X|Xs],Q,S) & \leftarrow \text{delta}(Q,X,S,NewQ,NewS), \\
& \quad \text{accept}_\text{from}(Xs,NewQ,NewS).
\end{align*}
\]

\[
\begin{align*}
\text{initial}(q0) & \leftarrow. \\
\text{final}(q1) & \leftarrow.
\end{align*}
\]

\[
\begin{align*}
\text{delta}(q0,X,Xs,q0,[X|Xs]) & \leftarrow. \\
\text{delta}(q0,X,Xs,q1,[X|Xs]) & \leftarrow. \\
\text{delta}(q0,X,Xs,q1,Xs) & \leftarrow. \\
\text{delta}(q1,X,[X|Xs],q1,Xs) & \leftarrow.
\end{align*}
\]

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.

- $N = 1$
- $N = 2$
- $N = 3$
Recursive Programming: Towers of Hanoi (Contd.)

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

  ![Diagram of the Towers of Hanoi](image)

  is represented by:

  ```prolog
  hanoi_moves( s(s(s(0))),
               [ move(a,b), move(a,c), move(b,c), move(a,b),
                 move(c,a), move(c,b), move(a,b) ])
  ```
Recursive Programming: Towers of Hanoi (Contd.)

- A general rule:

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

\[
\text{hanoi}(s(0), \text{Orig}, \text{Dest}, _, \text{Help}, [\text{move}(\text{Orig}, \text{Dest})]) \leftarrow .
\]
\[
\text{hanoi}(s(N), \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) \leftarrow
\]
\[
\quad \text{hanoi}(N, \text{Orig}, \text{Help}, \text{Dest}, \text{Moves1}),
\quad \text{hanoi}(N, \text{Help}, \text{Dest}, \text{Orig}, \text{Moves2}),
\quad \text{append}(\text{Moves1}, [\text{move}(\text{Orig}, \text{Dest}) | \text{Moves2}], \text{Moves}).
\]

- And we simply call this predicate:

\[
\text{hanoi_moves}(N, \text{Moves}) \leftarrow
\]
\[
\quad \text{hanoi}(N, \text{a}, \text{b}, \text{c}, \text{Moves}).
\]
Learning to Compose Recursive Programs

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