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>Expression</th>
<th>Term</th>
<th>Operator</th>
<th>Condition</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>
</tbody>
</table>

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:

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

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

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

  **Example:**
  \[
  \text{meal}(\text{soup, beef, coffee}) \leftarrow .
  \]
  \[
  \text{meal}(\text{First, Second, Third}) \leftarrow \text{appetizer}(\text{First}),
  \text{main_dish}(\text{Second}),
  \text{dessert}(\text{Third}).
  \]

- 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:**

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

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

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

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

  A query represents a *question to the program*.

  **Example**: `← 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 \( \text{animal(spot)} <- \) can be read as “spot is an animal”.

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

  Thus, a rule \( p \leftarrow p_1, \ldots, p_m. \) means “if \( p_1 \) and \( \ldots \) and \( p_m \) are true, then \( p \) is true”
  
  *Example*: the rule \( \text{pet(X)} \leftarrow \text{animal(X)}, \text{barks(X)}. \) 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
  
  \[ \text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X). \]
  
  \[ \text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X). \]
  
  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**:
  
  \[ \text{\textless - pet(spot).} \] asks whether \textit{spot} is a pet.
  \[ \text{\textless - pet}(X). \] asks: “Is there an \( X \) which is a pet?”
“Execution” and Semantics

• Example of a logic program:
  
  ```prolog
  pet(X) <- animal(X), barks(X).
  pet(X) <- animal(X), meows(X).
  animal(spot) <-.
  barks(spot) <-.
  animal(barry) <-.
  meows(barry) <-.
  animal(hobbes) <-.
  roars(hobbes) <-.
  ```

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

  Example: given the program above and the query `<- pet(X).` the system will try to find a “substitution” for `X` which makes `pet(X)` true.

  ◇ The declarative semantics specifies what should be computed (all possible answers).
    ⇒ Intuitively, we have two possible answers: `X = spot` and `X = barry`.

  ◇ The operational semantics specifies how answers are computed (which allows us to determine how many steps it will take).
Running 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) \( p \leftarrow p_1, \ldots, p_n \) \( p \leftarrow q_1, \ldots, q_m \) means:
  “to execute a call to \( p \), call \( p_1 \) \& \( \ldots \) \& \( p_n \), or, alternatively, \( q_1 \) \& \( \ldots \) \& \( q_n \), 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

- **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>$f(m(h), g(t))$</td>
<td>$f(m(h), g(t))$</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**.
Unification

• Often several solutions exist, e.g.:

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & \theta_1 & A\theta_1 \text{ and } B\theta_1 \\
\hline
f(X, g(T)) & f(m(H), g(M)) & \{ X=m(a), H=a, M=b, T=b \} & f(m(a), g(b)) \\
\hline
" & " & \{ X=m(H), M=f(A), T=f(A) \} & f(m(H), g(f(A))) \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & \theta_1 & A\theta_1 \text{ and } B\theta_1 \\
\hline
f(X, g(T)) & f(m(H), g(M)) & \{ X=m(H), T=M \} & f(m(H), g(M)) \\
\hline
\end{array}
\]

• 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 \( B = p(a, g(b)) \)

\[
\begin{array}{cccc}
\theta & E & T & S \\
\{\} & \{ p(X, f(Y)) = p(a, g(b)) \} & p(X, f(Y)) & p(a, g(b)) \\
\{\} & \{ X = a, f(Y) = g(b) \} & X & a \\
\{ X = a \} & \{ f(Y) = g(b) \} & f(Y) & g(b) \\
\text{fail} & & & \\
\end{array}
\]

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

\[
\begin{array}{cccc}
\theta & E & T & S \\
\{\} & \{ p(X, f(X)) = p(Z, Z) \} & p(X, f(X)) & p(Z, Z) \\
\{\} & \{ X = Z, f(X) = Z \} & X & Z \\
\{ X = Z \} & \{ f(Z) = Z \} & f(Z) & Z \\
\text{fail} & & & \\
\end{array}
\]
A (Schematic) Interpreter for Logic Programs (SLD–resolution)

Input: A logic program \( P \), a query \( Q \)
Output: \( Q_\mu \) (answer substitution) if \( Q \) is provable from \( P \), failure otherwise

Algorithm:

1. Initialize the “resolvent” \( R \) to be \( \{Q\} \)
2. While \( R \) is nonempty do:
   2.1. Take the leftmost literal \( A \) in \( R \)
   2.2. Choose a (renamed) clause \( A' \leftarrow B_1, \ldots, B_n \) from \( P \), such that \( A \) and \( A' \) unify with unifier \( \theta \) (if no such clause can be found, branch is failure; explore another branch)
   2.3. Remove \( A \) from \( R \), add \( B_1, \ldots, B_n \) to \( R \)
   2.4. Apply \( \theta \) to \( R \) and \( Q \)
3. If \( R \) is empty, output \( Q \) (a solution). Explore another branch for more sol’s.

- Step 2.2 defines alternative paths to be explored to find answer(s); execution explores this tree (for example, breadth-first).
Since step 2.2 is left open, a given logic *programming* system must specify how it deals with this by providing one (or more)

- **Search rule(s):** “how are clauses/branches selected in 2.2.”

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

**Example** (two valid executions):

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

In fact, there is also some freedom in step 2.1, i.e., a system may also specify:

- **Computation rule(s):** “how are literals selected in 2.1.”
Running programs

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

•  <- pet(P).

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

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

• System response:  P = barry  ?

• If we type “;” after the ? prompt (i.e., we ask for another solution) the system can go and execute a different branch (i.e., a different choice in C2* or C4*).
Running programs (different strategy)

\[ \text{C}_1: \quad \text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X). \]
\[ \text{C}_2: \quad \text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X). \]
\[ \text{C}_3: \quad \text{animal(spot)} \leftarrow. \]
\[ \text{C}_4: \quad \text{animal(barry)} \leftarrow. \]
\[ \text{C}_5: \quad \text{animal(hobbes)} \leftarrow. \]
\[ \text{C}_6: \quad \text{barks(spot)} \leftarrow. \]
\[ \text{C}_7: \quad \text{meows(barry)} \leftarrow. \]
\[ \text{C}_8: \quad \text{roars(hobbes)} \leftarrow. \]

\( \text{← pet}(P). \) (different strategy)

<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>( \text{C}_1^* )</td>
<td>( { P = X_1 } )</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>( \text{C}_5^* )</td>
<td>( { X_1 = \text{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 \( \text{C}_1^* \) or \( \text{C}_5^* \)) to find a solution. We take \( \text{C}_3 \) instead of \( \text{C}_5 \):

<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>( \text{C}_1^* )</td>
<td>( { P = X_1 } )</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>( \text{C}_3^* )</td>
<td>( { X_1 = \text{spot} } )</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>( \text{C}_6 )</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:
  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>

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

- Thus, procedures can be used in different modes
  (different sets of arguments are input or output in each mode).
A Logic Database is a set of facts and rules (i.e., a logic program):

\[
\begin{align*}
\text{father_of}(\text{john}, \text{peter}) & \iff \\
\text{father_of}(\text{john}, \text{mary}) & \iff \\
\text{father_of}(\text{peter}, \text{michael}) & \iff \\
\text{mother_of}(\text{mary}, \text{david}) & \iff
\end{align*}
\]

\[
\begin{align*}
\text{grandfather_of}(L, M) & \iff \text{father_of}(L, N), \\
& \quad \text{father_of}(N, M).
\end{align*}
\]

\[
\begin{align*}
\text{grandfather_of}(X, Y) & \iff \text{father_of}(X, Z), \\
& \quad \text{mother_of}(Z, Y).
\end{align*}
\]

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

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

Rules for \text{grandmother_of}(X, Y)?
Another example:

\[
\begin{align*}
\text{resistor(power,n1)} & \leftarrow. \\
\text{resistor(power,n2)} & \leftarrow. \\
\text{transistor(n2,ground,n1)} & \leftarrow. \\
\text{transistor(n3,n4,n2)} & \leftarrow. \\
\text{transistor(n5,ground,n4)} & \leftarrow. \\
\text{inverter(Input,Output)} & \leftarrow \text{transistor(Input,ground,Output), resistor(power,Output)}. \\
\text{nand\_gate(Input1,Input2,Output)} & \leftarrow \text{transistor(Input1,X,Output), transistor(Input2,ground,X), resistor(power,Output)}. \\
\text{and\_gate(Input1,Input2,Output)} & \leftarrow \text{nand\_gate(Input1,Input2,X), inverter(X, Output)}. 
\end{align*}
\]

Query \text{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:**

  
  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)}
• The circuit example revisited:

\[
\begin{align*}
\text{resistor}(r_1, \text{power}, n_1) & \leftarrow \text{transistor}(t_1, n_2, \text{ground}, n_1) \leftarrow. \\
\text{resistor}(r_2, \text{power}, n_2) & \leftarrow \text{transistor}(t_2, n_3, n_4, n_2) \leftarrow. \\
\text{transistor}(t_3, n_5, \text{ground}, n_4) & \leftarrow.
\end{align*}
\]

\[
\begin{align*}
\text{inverter}(\text{inv}(T, R), \text{Input}, \text{Output}) & \leftarrow \\
\text{transistor}(T, \text{Input}, \text{ground}, \text{Output}), \text{resistor}(R, \text{power}, \text{Output}).
\end{align*}
\]

\[
\begin{align*}
\text{nand}_\text{gate}(\text{nand}(T_1, T_2, R), \text{Input}_1, \text{Input}_2, \text{Output}) & \leftarrow \\
\text{transistor}(T_1, \text{Input}_1, X, \text{Output}), \text{transistor}(T_2, \text{Input}_2, \text{ground}, X), \\
\text{resistor}(R, \text{power}, \text{Output}).
\end{align*}
\]

\[
\begin{align*}
\text{and}_\text{gate}(\text{and}(N, I), \text{Input}_1, \text{Input}_2, \text{Output}) & \leftarrow \\
\text{nand}_\text{gate}(N, \text{Input}_1, \text{Input}_2, X), \text{inverter}(I, X, \text{Output}).
\end{align*}
\]

• The query \( \leftarrow \text{and}_\text{gate}(G, \text{In}_1, \text{In}_2, \text{Out}). \)

has solution: \( \{ G=\text{and}(\text{nand}(t_2, t_3, r_2), \text{inv}(t_1, r_1)), \text{In}_1=n_3, \text{In}_2=n_5, \text{Out}=n_1 \} \)
Logic Programs and the Relational DB Model

Traditional → Codd’s Relational Model

File → Relation
Record → Tuple
Field → Attribute

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.)

Relational Database  →  Logic Programming
Relation Name  →  Predicate symbol
Relation  →  Procedure consisting of ground facts
                      (facts without variables)
Tuple  →  Ground fact
Attribute  →  Argument of predicate

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

- **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) <-.
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**:
    
    \[
    \text{r\_1\_3}(X_1, X_3) \leftarrow r(X_1, X_2, X_3).
    \]
  - **Selection**:
    
    \[
    \text{r\_selected}(X_1, X_2, X_3) \leftarrow r(X_1, X_2, X_3), \leq (X_2, X_3).
    \]
    (see later for definition of $\leq/2$)
Logic Programs and the Relational DB Model (Contd.)

- Derived operations – some can be expressed more directly in LP:
  - Intersection:
    \[
    \text{r\_meet\_s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, \ldots, X_n), \text{s}(X_1, \ldots, X_n).
    \]
  - Join:
    \[
    \text{r\_joinX2\_s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, X_2, X_3, \ldots, X_n), \text{s}(X'_1, X_2, X'_3, \ldots, X'_n).
    \]
- Duplicates an issue: see “setof” later in Prolog.
Deductive Databases

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

- Example: ancestors.

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

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

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

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

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

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

... 

- Defining ancestor recursively:

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

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

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

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

- 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:
    is_weekday('Monday') <-.
    is_weekday('Tuesday') <-. ...

• **Example**: Date (weekday * day in the month):
  ◦ Set of terms to represent: date('Monday',23), date(Tuesday,24), ...
  ◦ Type definition:
    is_date(date(W,D)) <- is_weekday(W), is_day_of_month(D).
    is_day_of_month(1) <-.
    is_day_of_month(2) <-.
    ...
    is_day_of_month(31) <-.
Recursive Programming: Recursive Types

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

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

- We can reason about *complexity*, for a given class of queries (“mode”). E.g., for mode \texttt{nat(ground)} complexity is *linear* in size of number.
- **Example**: integers:
  - Set of terms to represent: 0, s(0), -s(0), …
  - Type definition:
    - \texttt{integer(X) <- nat(X).}
    - \texttt{integer(-X) <- nat(X).}
Recursive Programming: Arithmetic

- Defining the natural order (\(\leq\)) of natural numbers:
  
  \[
  \text{less_or_equal}(0,X) \leftarrow \text{nat}(X).
  \]
  
  \[
  \text{less_or_equal}(s(X),s(Y)) \leftarrow \text{less_or_equal}(X,Y).
  \]

- **Multiple uses:** \text{less_or_equal}(s(0),s(s(0))), \text{less_or_equal}(X,0), ...

- **Multiple solutions:** \text{less_or_equal}(X,s(0)), \text{less_or_equal}(s(s(0)),Y), etc.

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

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

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

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

• The meaning of \text{plus} is the same if both definitions are combined.

• Not recommended: several proof trees for the same query \rightarrow not efficient, not concise. We look for minimal axiomatizations.

• The art of logic programming: finding compact and computationally efficient formulations!

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

• Definition of $\text{mod}(X, Y, Z)$
  “$Z$ is the remainder from dividing $X$ by $Y$”
  $(\exists \ Q \text{ s.t. } X = Y\times Q + Z \text{ and } Z < Y)$:
  $\text{mod}(X, Y, Z) \gets \text{less}(Z, Y), \text{times}(Y, Q, W), \text{plus}(W, Z, X)$.

  $\text{less}(0, s(X)) \gets \text{nat}(X)$.
  $\text{less}(s(X), s(Y)) \gets \text{less}(X, Y)$.

• Another possible definition:
  $\text{mod}(X, Y, X) \gets \text{less}(X, Y)$.
  $\text{mod}(X, Y, Z) \gets \text{plus}(X, Y, X), \text{mod}(X, Y, Z)$.

• 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:
  \[
  \text{ackermann}(0,N,s(N)) \leftarrow . \\
  \text{ackermann}(s(M),0,Val) \leftarrow \text{ackermann}(M,s(0),Val). \\
  \text{ackermann}(s(M),s(N),Val) \leftarrow \text{ackermann}(s(M),N,Val1), \\
  \text{ackermann}(M,Val1,Val).
  \]

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

- Syntactic support available (see, e.g., the Ciao *functions* package).
Recursive Programming: Lists

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

<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):
  
  \[
  \text{list}([]) \leftarrow .
  \]
  
  \[
  \text{list}(.(X,Y)) \leftarrow \text{list}(Y). 
  \]

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

- X is a *member* of the list Y:

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

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

- 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:
  member(X, [X|Y]) <- list(Y).
  member(X, [_|T]) <- member(X, T).

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

• Define: prefix(X, Y) (the list X is a prefix of the list Y), e.g.
  prefix([[a, b], [a, b, c, d]])

• Define: suffix(X, Y), sublist(X, Y), ...

• Define length(Xs, N) (N is the length of the list Xs)
Recursive Programming: Lists (Contd.)

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

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

• Second generalization:
  \[
  \text{append}([X], Ys, [X|Ys]) \leftarrow \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z], Ys, [X,Z|Ys]) \leftarrow \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z,W], Ys, [X,Z,W|Ys]) \leftarrow \text{list}(Ys).
  \]

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

• So, we have:

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

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

• 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.)

- `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:

  ```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.
Represented by a ternary functor $\text{tree}(\text{Element}, \text{Left}, \text{Right})$.

Empty tree represented by $\text{void}$.

Definition:

\[
\text{binary_tree(\text{void})} \leftarrow \).
\text{binary_tree(\text{tree}(\text{Element}, \text{Left}, \text{Right}))} \leftarrow
\text{binary_tree(\text{Left})}, \text{binary_tree(\text{Right})}.
\]

Defining $\text{tree_member(\text{Element}, \text{Tree})}$:

\[
\text{tree_member(X,tree(X,\text{Left},\text{Right}))} \leftarrow
\text{binary_tree(\text{Left})}, \text{binary_tree(\text{Right})}.
\text{tree_member(X,tree(Y,\text{Left},\text{Right}))} \leftarrow \text{tree_member(X,\text{Left})}.
\text{tree_member(X,tree(Y,\text{Left},\text{Right}))} \leftarrow \text{tree_member(X,\text{Right})}.
\]
Recursive Programming: Binary Trees

- Defining `pre_order(Tree,Order)

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

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

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

- **In Pascal:**
  ```pascal
  type tree = ^treerec;
  treerec = record
    data : integer;
    left : tree;
    right: tree;
  end;

  var t : tree;

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

• Note that the two definitions of member/2 can be used simultaneously:

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

  \[
  \text{lt_member}(X, [_|T]) \leftarrow \text{lt_member}(X, T).
  \]

  \[
  \text{lt_member}(X, \text{tree}(X, L, R)) \leftarrow \text{binary_tree}(L), \text{binary_tree}(R).
  \]

  \[
  \text{lt_member}(X, \text{tree}(Y, L, R)) \leftarrow \text{lt_member}(X, L).
  \]

  \[
  \text{lt_member}(X, \text{tree}(Y, L, R)) \leftarrow \text{lt_member}(X, R).
  \]

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

• $\leftarrow \text{lt_member}(X, [b,a,c])$.
  $X = b$ ; $X = a$ ; $X = c$

• $\leftarrow \text{lt_member}(X, \text{tree}(b, \text{tree}(a, \text{void}, \text{void}), \text{tree}(c, \text{void}, \text{void})))$.
  $X = b$ ; $X = a$ ; $X = c$

• Also, try (somewhat surprising): $\leftarrow \text{lt_member}(M, T)$. 
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

  \[
  \text{polynomial}(X,X) \leftarrow .
  \text{polynomial}(\text{Term},X) \leftarrow \text{pconstant}(\text{Term}).
  \text{polynomial}(\text{Term}_1+\text{Term}_2,X) \leftarrow \text{polynomial}(\text{Term}_1,X), \text{polynomial}(\text{Term}_2,X).
  \text{polynomial}(\text{Term}_1-\text{Term}_2,X) \leftarrow \text{polynomial}(\text{Term}_1,X), \text{polynomial}(\text{Term}_2,X).
  \text{polynomial}(\text{Term}_1\times\text{Term}_2,X) \leftarrow \text{polynomial}(\text{Term}_1,X), \text{polynomial}(\text{Term}_2,X).
  \text{polynomial}(\text{Term}_1/\text{Term}_2,X) \leftarrow \text{polynomial}(\text{Term}_1,X), \text{pconstant}(\text{Term}_2).
  \text{polynomial}(\text{Term}_1^N,X) \leftarrow \text{polynomial}(\text{Term}_1,X), \text{nat}(N).
  \]
Recursive Programming: Manipulating Symb. Expressions (Contd.)

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

  deriv(X,X,s(0)) <-.
  deriv(C,X,0) <- pconstant(C).
  deriv(U+V,X,DU+DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U-V,X,DU-DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U*V,X,DU*V+U*DV) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U/V,X,(DU*V-U*DV)/V^s(s(0))) <- deriv(U,X,DU), deriv(V,X,DV).
  deriv(U^s(N),X,s(N)*U^N*DU) <- deriv(U,X,DU), nat(N).
  deriv(log(U),X,DU/U) <- deriv(U,X,DU).

  ...

- <- deriv(s(s(s(0)))*x+s(s(0)),x,Y).

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

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

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

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

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

- Program:

  \[
  \begin{align*}
  \text{initial}(q0) & \leftarrow. \\
  \text{delta}(q0,a,q1) & \leftarrow. \\
  \text{delta}(q1,b,q0) & \leftarrow. \\
  \text{final}(q0) & \leftarrow. \\
  \text{delta}(q1,b,q1) & \leftarrow. \\
  \text{accept}(S) & \leftarrow \text{initial}(Q), \text{accept_from}(S,Q). \\
  \text{accept_from}([],Q) & \leftarrow \text{final}(Q). \\
  \text{accept_from}([X|Xs],Q) & \leftarrow \text{delta}(Q,X,NewQ), \text{accept_from}(Xs,NewQ).
  \end{align*}
  \]
Recursive Programming: Automata (Graphs) (Contd.)

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

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

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

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

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

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

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

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

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

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

- **What sequence does it recognize?**
Recursive Programming: Towers of Hanoi

- **Objective:**
  - Move tower of N disks from peg a to peg b, with the help of peg c.

- **Rules:**
  - Only one disk can be moved at a time.
  - A larger disk can never be placed on top of a smaller disk.

![Diagram of Towers of Hanoi for N = 1, 2, 3]
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 $\text{move}(A, B)$ represents that the top disk in $A$ should be moved to $B$.
- Example:

$$\text{hanoi_moves}( \text{s}(\text{s}(\text{s}(0))),
\begin{array}{l}
\text{[ move(a,b), move(a,c), move(b,c), move(a,b),}
\text{ move(c,a), move(c,b), move(a,b) ]}
\end{array}$$
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 $\text{Help}$.”

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

  \[
  \text{hanoi}(s(N), \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) \leftarrow
  \]

  \[
  \quad \text{hanoi}(N, \text{Orig}, \text{Help}, \text{Dest}, \text{Moves1}),
  \quad \text{hanoi}(N, \text{Help}, \text{Dest}, \text{Orig}, \text{Moves2}),
  \quad \text{append}([\text{move}(\text{Orig}, \text{Dest})], \text{Moves1}, \text{Moves2}, \text{Moves})
  \]

• And we simply call this predicate:

  \[
  \text{hanoi}_{\text{moves}}(N, \text{Moves}) \leftarrow
  \]

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