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!):

\[
\begin{align*}
    a + b & \text{ is the term } +'(a, b) \text{ if } +/2 \text{ declared infix} \\
    - b & \text{ is the term } -'(b) \text{ if } -/1 \text{ declared prefix} \\
    a < b & \text{ is the term } '<(a, b) \text{ if } </2 \text{ declared infix} \\
    \text{john father mary} & \text{ is the term } father(john, mary) \text{ if } father/2 \text{ declared infix}
\end{align*}
\]

We assume that some such operator definitions are always preloaded.

Syntax: Rules and Facts (Clauses)

- Rule: an expression of the form:

\[
p_0(t_1, t_2, \ldots, t_{n_0}) \leftarrow p_1(t_{11}, t_{21}, \ldots, t_{n_1}),
\]

\[
\ldots
\]

\[
p_m(t_{1m}, t_{2m}, \ldots, t_{nm}).
\]

- \(p_0(...)\) to \(p_m(...)\) are syntactically like terms.
- \(p_0(...)\) 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:**

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:**

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

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

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

- **Query:** an expression of the form: \[ ← p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_n}^n). \]
  (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 "\[ p \leftarrow . \]" can be seen as the rule "\[ p \leftarrow \text{true}. \]")

  **Example:** the fact \[ \text{animal(spot)} \leftarrow . \]
  can be read as "spot is an animal".

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

  Thus, a rule \[ p \leftarrow p_1, \ldots, p_m. \] means "if \[ p_1 \land \cdots \land p_m \] 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
  
  \[ p \leftarrow p_1, \ldots, p_n \]
  
  \[ p \leftarrow q_1, \ldots, q_m \]
  
  provide different alternatives (for \( p \)).

**Example**: the rules

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

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

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

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

  \[
  \begin{align*}
  & \leftarrow \text{pet}(\text{spot}). & \leftarrow \text{pet}(X). \\
  \text{asks whether spot is a pet.} & \text{asks: "Is there an X which is a pet?"}
  \end{align*}
  \]

"Execution" and Semantics

- **Example of a logic program**:

  \[
  \begin{align*}
  \text{pet}(X) & \leftarrow \text{animal}(X), \text{barks}(X). \\
  \text{pet}(X) & \leftarrow \text{animal}(X), \text{meows}(X). \\
  \text{animal}(\text{spot}) & \leftarrow. \\
  \text{animal}(\text{barry}) & \leftarrow. \\
  \text{animal(\text{hobbes})} & \leftarrow. \\
  \text{barks}(\text{spot}) & \leftarrow. \\
  \text{meows(\text{barry})} & \leftarrow. \\
  \text{roars(\text{hobbes})} & \leftarrow.
  \end{align*}
  \]

- **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).
    
    \( \Rightarrow \) Intuitively, we have two possible answers: \( X = \text{spot} \) and \( X = \text{barry} \).
  
  - The **operational semantics** specifies how answers are computed (which allows us to determine how many steps it will take).
Running Pure Logic Programs: the Ciao System’s bf/af Packages

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

- Writing programs to execute in bf mode:
  - All files should start with the following line:
    ```prolog
    :- module(_,_,[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”):

  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 ← p is an initial procedure call.
- A procedure definition with one clause p ← p₁, ..., pₘ means:
  “to execute a call to p you have to call p₁ and ... and pₘ”
  ○ In principle, the order in which p₁, ..., pₘ are called does not matter, but, in practical systems it is fixed.
- If several clauses (definitions) p ← p₁, ..., pₘ means:
  p ← q₁, ..., qₙ
  “to execute a call to p, call p₁ ∧ ... ∧ pₘ, or, alternatively, q₁ ∧ ... ∧ qₙ, 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.

Unifying two terms (or literals) \( A \) and \( B \): is asking if they can be made syntactically identical by giving (minimal) values to their variables.

- I.e., find a **variable substitution** \( \theta \) such that \([A\theta = B\theta]\) (or, if impossible, fail).
- Only variables can be given values!
- Two structures can be made identical only by making their arguments identical.

**E.g.:**

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

- (1) Structures with different name and/or arity cannot be unified.
- (2) A variable cannot be given as value a term which contains that variable, because it would create an infinite term. This is known as the **occurs check**.
Unification

• Often several solutions exist, e.g.:

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

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

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

• Always a unique (modulo variable renaming) most general solution exists (unless unification fails).

• This is the one that we are interested in.

• The unification algorithm finds this solution.

Unification Algorithm

• Let $A$ and $B$ be two terms:

1. $\theta = \emptyset$, $E = \{ A = B \}$

2. while not $E = \emptyset$:

   2.1 delete an equation $T = S$ from $E$

   2.2 case $T$ or $S$ (or both) are (distinct) variables. Assuming $T$ variable:

      * (occur check) if $T$ occurs in the term $S \rightarrow$ halt with failure

      * substitute variable $T$ by term $S$ in all terms in $\theta$

      * substitute variable $T$ by term $S$ in all terms in $E$

      * add $T = S$ to $\theta$

   2.3 case $T$ and $S$ are non-variable terms:

      * if their names or arities are different $\rightarrow$ halt with failure

      * obtain the arguments $\{ T_1, \ldots, T_n \}$ of $T$ and $\{ S_1, \ldots, S_n \}$ of $S$

      * add $\{ T_1 = S_1, \ldots, T_n = S_n \}$ to $E$

3. halt with $\theta$ being the m.g.u of $A$ and $B$
Unification Algorithm Examples (I)

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

\[
\begin{array}{c|c|c|c}
\theta & E & T & S \\
\hline
\{\} & \{p(X,X)=p(f(Z),f(W))\} & p(X,X) & p(f(Z),f(W)) \\
\{\} & \{X=f(Z),X=f(W)\} & X & f(Z) \\
\{X=f(Z)\} & \{f(Z)=f(W)\} & f(Z) & f(W) \\
\{X=f(Z)\} & \{Z=W\} & Z & W \\
\{X=f(W),Z=W\} & \{\} & \\
\end{array}
\]

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

\[
\begin{array}{c|c|c|c}
\theta & E & T & S \\
\hline
\{\} & \{p(X,f(Y))=p(Z,X)\} & p(X,f(Y)) & p(Z,X) \\
\{\} & \{X=Z,f(Y)=X\} & X & Z \\
\{X=Z\} & \{f(Y)=Z\} & f(Y) & Z \\
\{X=f(Y),Z=f(Y)\} & \{\} & \\
\end{array}
\]

Unification Algorithm Examples (II)

• Unify: $A = p(X,f(Y))$ and $B = p(a,g(b))$

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

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

\[
\begin{array}{c|c|c|c}
\theta & E & T & S \\
\hline
\{\} & \{p(X,f(X))=p(Z,Z)\} & p(X,f(X)) & p(Z,Z) \\
\{\} & \{X=Z,f(X)=Z\} & X & Z \\
\{X=Z\} & \{f(Z)=Z\} & f(Z) & Z \\
\text{fail} & & & \\
\end{array}
\]
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).

A (Schematic) Interpreter for Logic Programs (Contd.)

- Since step 2.2 is left open, a given logic programming system must specify how it deals with this by providing one (or more)
  - Search rule(s): “how are clauses/branches selected in 2.2.”
- If the search rule is not specified execution 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).  ?- pet(X).
  X = spot ? ;  X = barry ? ;
  X = barry ? ;  X = spot ? ;
  no  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

$C_1$: pet(X) <- animal(X), barks(X).
$C_2$: pet(X) <- animal(X), meows(X).
$C_3$: animal(spot) <-.
$C_4$: animal(barry) <-.
$C_5$: animal(hobbes) <-.
$C_6$: barks(spot) <-.
$C_7$: meows(barry) <-.
$C_8$: roars(hobbes) <-.

$\leftarrow$ pet(P).

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$R$</th>
<th>Clause</th>
<th>$\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>$C_1^*$</td>
<td>${P = X_1}$</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), meows(X_1)</td>
<td>$C_4^*$</td>
<td>${X_1 = \text{barry}}$</td>
</tr>
<tr>
<td>pet(barry)</td>
<td>meows(barry)</td>
<td>$C_7$</td>
<td>{}</td>
</tr>
<tr>
<td>pet(barry)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- System response: $P = \text{barry}$ ?
- If we type ":[;" after the ? prompt (i.e., we ask for another solution) the system can go and execute a different branch (i.e., a different choice in $C_5^*$ or $C_4^*$).

Running programs (different strategy)

$C_1$: pet(X) <- animal(X), barks(X).
$C_2$: pet(X) <- animal(X), meows(X).
$C_3$: animal(spot) <-.
$C_4$: animal(barry) <-.
$C_5$: animal(hobbes) <-.
$C_6$: barks(spot) <-.
$C_7$: meows(barry) <-.
$C_8$: roars(hobbes) <-.

$\leftarrow$ 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>$C_1^*$</td>
<td>${P = X_1}$</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>$C_5^*$</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 $C_1^*$ or $C_5^*$) to find a solution. We take $C_1$ instead of $C_5$:

<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>$C_1^*$</td>
<td>${P = X_1}$</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>$C_4^*$</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>
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]):

```
  pet(X)  
  /  
animal(X), barks(X)  animal(X), meows(X)  
  /  
animal(spot)  animal(barry)  animal(hobbes)  
  /  
animal(spot)  animal(barry)  animal(hobbes)  
  /  
barks(spot)  
```

- 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:
  \[
  \text{animal(dog(barry))} \leftarrow . \quad \text{named(dog(Name)}, \text{Name}) \leftarrow .
  \]
- Also, unification is used to pass parameters in procedure calls and to return values upon procedure exit.

<table>
<thead>
<tr>
<th>Q</th>
<th>R</th>
<th>Clause</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>C₁*</td>
<td>{ P=X₁ }</td>
</tr>
<tr>
<td>pet(X₁) animal(X₁), barks(X₁)</td>
<td>C₂*</td>
<td>{ X₁=spot }</td>
<td></td>
</tr>
<tr>
<td>pet(spot) barks(spot)</td>
<td>C₃*</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):
  
  \[
  \begin{align*}
  \text{father_of(john, peter)} & \leftarrow . \\
  \text{father_of(john, mary)} & \leftarrow . \\
  \text{father_of(peter, michael)} & \leftarrow . \\
  \text{mother_of(mary, david)} & \leftarrow . \\
  \text{grandfather_of(L,M)} & \leftarrow \text{father_of(L,N)}, \\
  & \quad \text{father_of(N,M)}. \\
  \text{grandfather_of(X,Y)} & \leftarrow \text{father_of(X,Z)}, \\
  & \quad \text{mother_of(Z,Y)}. \\
  \end{align*}
  \]
- Given such database, a logic programming system can answer questions (queries) such as:
  
  \(-\) father_of(john, peter).
  Answer: Yes

  \(-\) father_of(john, david).
  Answer: No

  \(-\) father_of(john, X).
  Answer: \{X = peter\}
  Answer: \{X = mary\}

  \(-\) grandfather_of(X, Y)?
  Answer: \{X = john\}
  Answer: \{X = john, Y = michael\}
  Answer: \{X = john, Y = david\}
  Answer: No
Database Programming (Contd.)

- Another example:

```
resistor(power,n1) <-.
resistor(power,n2) <-.
transistor(n2,ground,n1) <-.
transistor(n3,n4,n2) <-.
transistor(n5,ground,n4) <-.
```

```
inverter(Input,Output) <-
    transistor(Input,ground,Output), resistor(power,Output).
```

```
nand_gate(Input1,Input2,Output) <-
    transistor(Input1,X,Output), transistor(Input2,ground,X),
    resistor(power,Output).
```

```
and_gate(Input1,Input2,Output) <-
    nand_gate(Input1,Input2,X), inverter(X, Output).
```

- Query `and_gate(In1,In2,Out)` has solution: \{In1=n3, In2=n5, Out=n1\}

Structured Data and Data Abstraction (and the ‘=’ Predicate)

- Data structures are created using (complex) terms.

- Structuring data is important:

```
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 “=/2(X,Y)” where the predicate “=/2” is defined as the fact “=/2(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:
  
  \[
  \text{course(complog,Time,Lecturer, Location) } \leftarrow \\
  \text{Time = } t(\text{wed},18:30,20:30), \\
  \text{Lecturer = lect(’M.’,’Hermenegildo’),} \\
  \text{Location = loc(new,5102).}
  \]

- When is the Computational Logic course?
  
  \[
  \leftarrow \text{course(complog,Time, A, B).}
  \]
  
  has solution:
  
  \[
  \{ \text{Time=t(wed,18:30,20:30), A=lect(’M.’,’Hermenegildo’), B=loc(new,5102)} \}
  \]

- Using the anonymous variable (“.”):
  
  \[
  \leftarrow \text{course(complog,Time, , ).}
  \]
  
  has solution:
  
  \[
  \{ \text{Time=t(wed,18:30,20:30)} \}
  \]

Structured Data and Data Abstraction (Contd.)

- The circuit example revisited:
  
  \[
  \begin{align*}
  \text{resistor(r1,power,n1) } & \leftarrow . \\
  \text{resistor(r2,power,n2) } & \leftarrow . \\
  \text{transistor(t1,n2,ground,n1) } & \leftarrow . \\
  \text{transistor(t2,n3,n4,n2) } & \leftarrow . \\
  \text{transistor(t3,n5,ground,n4) } & \leftarrow .
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \text{nand_gate(nand(T1,T2,R),Input1,Input2,Output) } & \leftarrow . \\
  \text{transistor(T1,Input1,X,Output), transistor(T2,Input2,ground,X),} \\
  \text{resistor(R,power,Output).}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{and_gate(and(N,I),Input1,Input2,Output) } & \leftarrow . \\
  \text{nand_gate(N,Input1,Input2,X), inverter(I,X,Output).}
  \end{align*}
  \]

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

**Traditional $\rightarrow$ Codd's Relational Model**

<table>
<thead>
<tr>
<th>File</th>
<th>Relation</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>Tuple</td>
<td>Row</td>
</tr>
<tr>
<td>Field</td>
<td>Attribute</td>
<td>Column</td>
</tr>
</tbody>
</table>

- **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>

- The order of the rows is immaterial.
- (Duplicate rows are not allowed)

Logic Programs and the Relational DB Model (Contd.)

**Relational Database $\rightarrow$ Logic Programming**

- **Relation Name** $\rightarrow$ Predicate symbol
- **Relation** $\rightarrow$ Procedure consisting of ground facts (facts without variables)
- **Tuple** $\rightarrow$ Ground fact
- **Attribute** $\rightarrow$ 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) <.-`
The operations of the relational model are easily implemented as rules.

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

- **Set Difference:**
  \[ \text{r} \setminus \text{s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, \ldots, X_n), \neg \text{s}(X_1, \ldots, X_n). \]
  \[ \text{r} \setminus \text{s}(X_1, \ldots, X_n) \leftarrow \text{s}(X_1, \ldots, X_n), \neg \text{r}(X_1, \ldots, X_n). \]
  (we postpone the discussion on negation until later.)

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

- **Projection:**
  \[ \text{r}_{1,3}(X_1, X_3) \leftarrow \text{r}(X_1, X_2, X_3). \]

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

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

- **Join:**
  \[ \text{r} \cdot \text{s}(X_1, \ldots, X_n) \leftarrow \text{r}(X_1, X_2, X_3, \ldots, X_n), \text{s}(X'_1, X'_2, X'_3, \ldots, X'_n). \]

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.

  parent(X,Y) <- father(X,Y).
  parent(X,Y) <- mother(X,Y).

  ancestor(X,Y) <- parent(X,Y).
  ancestor(X,Y) <- parent(X,Z), parent(Z,Y).
  ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,Y).
  ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,K), parent(K,Y).
  ...

- Defining ancestor recursively:

  parent(X,Y) <- father(X,Y).
  parent(X,Y) <- mother(X,Y).

  ancestor(X,Y) <- parent(X,Y).
  ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).

- Exercise: define “related”, “cousin”, “same generation”, etc.
Types

- **Type**: a (possibly infinite) set of terms.
- **Type definition**: A program defining a type.
- **Example**: Weekday:
  - Set of terms to represent: Monday, Tuesday, Wednesday, ...
  - Type definition:
    - 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:
    - nat(0) <-.
    - nat(s(X)) <- nat(X).

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

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

- Defining the natural order (≤) 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 → 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 \cdot 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 \( Y \)”
  \( (\exists \ Q \text{ s.t. } X = Y*Q + Z \text{ and } Z < Y) \):
  \[
  \text{mod}(X,Y,Z) \leftarrow \text{less}(Z,Y), \text{times}(Y,Q,W), \text{plus}(W,Z,X).
  \]

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

- In general, \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>[a]</td>
</tr>
<tr>
<td>.(a..(b,[]))</td>
<td>[a][b][ ]</td>
<td>[a,b]</td>
</tr>
<tr>
<td>.(a..(b..(c,[])))</td>
<td>[a][b][c][ ]</td>
<td>[a,b,c]</td>
</tr>
<tr>
<td>.(a,X)</td>
<td>[a][X]</td>
<td>[a][X]</td>
</tr>
<tr>
<td>.(a..(b,X))</td>
<td>[a][b][X]</td>
<td>[a,b][X]</td>
</tr>
</tbody>
</table>

- Note that:
  - \([a,b]\) and \([a][X]\) unify with \(\{X = b\}\)
  - \([a]\) and \([a][X]\) unify with \(\{X = \_\}\)
  - \([a]\) and \([a,b][X]\) do not unify
  - \([\_]\) and \([X]\) do not unify

Recursive Programming: Lists

- Type definition (no syntactic sugar):
  ```
  list([]) <- .
  list(.(X,Y)) <- list(Y).
  ```
- Type definition (with syntactic sugar):
  ```
  list([]) <- .
  list([X|Y]) <- list(Y).
  ```
Recursive Programming: Lists (Contd.)

• X is a member of the list Y:

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

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

• Resulting definition:

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

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

This is still infinite \(\rightarrow\) we need to generalize more.

---

Recursive Programming: Lists (Contd.)

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

- \texttt{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:

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

  How \ can \ we \ stop?

  \begin{verbatim}
  reverse([],[]) <-.
  \end{verbatim}

- As \ defined, \ \texttt{reverse(Xs,Ys)} \ is \ very \ inefficient. \ Another \ possible \ definition:

  \begin{verbatim}
  reverse(Xs,Ys) <- reverse(Xs,[],Ys).
  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 \ \texttt{tree(Element,Left,Right)}.

- Empty \ tree \ represented \ by \ \texttt{void}.

- Definition:

  \begin{verbatim}
  binary_tree(void) <- .
  binary_tree(tree(Element,Left,Right)) <-
      binary_tree(Left),
      binary_tree(Right).
  \end{verbatim}

- Defining \ \texttt{tree_member(Element,Tree)}:

  \begin{verbatim}
  tree_member(X,tree(X,Left,Right)) <-
      binary_tree(Left),
      binary_tree(Right).
  tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Left).
  tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Right).
  \end{verbatim}
Recursive Programming: Binary Trees

- Defining `pre_order(Tree,Order)`:  
  ```prolog
  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:  
  ```prolog
  T = tree(3, tree(2,void,void), tree(5,void,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 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!

  - \( \leftarrow \text{lt_member}(X, [b,a,c]). \)
    \[ X = b \text{;} X = a \text{;} X = c \]
  - \( \leftarrow \text{lt_member}(X, \text{tree}(b, \text{tree}(a, \text{void}, \text{void}), \text{tree}(c, \text{void}, \text{void}))). \)
    \[ X = b \text{;} X = a \text{;} X = c \]

  - 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

  \[ \text{polynomial}(X,X) \leftarrow. \]
  \[ \text{polynomial}(\text{Term},X) \leftarrow \text{pconstant}(\text{Term}). \]
  \[ \text{polynomial}(\text{Term1}+\text{Term2},X) \leftarrow \text{polynomial}(\text{Term1},X), \text{polynomial}(\text{Term2},X). \]
  \[ \text{polynomial}(\text{Term1} \text{-} \text{Term2},X) \leftarrow \text{polynomial}(\text{Term1},X), \text{polynomial}(\text{Term2},X). \]
  \[ \text{polynomial}(\text{Term1} \times \text{Term2},X) \leftarrow \text{polynomial}(\text{Term1},X), \text{polynomial}(\text{Term2},X). \]
  \[ \text{polynomial}(\text{Term1} \div \text{Term2},X) \leftarrow \text{polynomial}(\text{Term1},X), \text{pconstant}(\text{Term2}). \]
  \[ \text{polynomial}(\text{Term1}^N,X) \leftarrow \text{polynomial}(\text{Term1},X), \text{nat}(N). \]
Recursive Programming: Manipulating Symb. Expressions (Contd.)

- Symbolic differentiation: \( \text{deriv}(\text{Expression}, \ X, \ \text{DifferentiatedExpression}) \)

\[
\text{deriv}(X, X, s(0)) \leftarrow.
\]
\[
\text{deriv}(C, X, 0) \leftarrow \text{pconstant}(C).
\]
\[
\text{deriv}(U+V, X, DU+DV) \leftarrow \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV).
\]
\[
\text{deriv}(U-V, X, DU-DV) \leftarrow \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV).
\]
\[
\text{deriv}(U*V, X, DU*V+U*DV) \leftarrow \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV).
\]
\[
\text{deriv}(U/V, X, (DU*V-U*DV)/V^s(s(0))) \leftarrow \text{deriv}(U, X, DU), \ \text{deriv}(V, X, DV).
\]
\[
\text{deriv}(U^s(N), X, s(N)*U^N*DU) \leftarrow \text{deriv}(U, X, DU), \ \text{nat}(N).
\]
\[
\text{deriv}(\log(U), X, DU/U) \leftarrow \text{deriv}(U, X, DU).
\]
...

\[
\leftarrow \text{deriv}(s(s(s(0)))*x+s(s(0)), x, Y).
\]

- A simplification step can be added.

Recursive Programming: Automata (Graphs)

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

\[
\begin{array}{ccc}
q_0 & a & q_1 \\
& b & \end{array}
\]

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

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

- Program:

\[
\begin{align*}
\text{initial}(q_0) & \leftarrow. \\
\text{delta}(q_0, a, q_1) & \leftarrow. \\
\text{delta}(q_1, b, q_0) & \leftarrow. \\
\text{final}(q_0) & \leftarrow. \\
\text{delta}(q_1, b, q_1) & \leftarrow.
\end{align*}
\]

\[
\text{accept}(S) \leftarrow \text{initial}(Q), \ \text{accept_from}(S, Q).
\]
\[
\text{accept_from}([], Q) \leftarrow \text{final}(Q).
\]
\[
\text{accept_from}([X]|Xs, Q) \leftarrow \text{delta}(Q, X, \text{NewQ}), \ \text{accept_from}(Xs, \text{NewQ}).
\]
Recursive Programming: Automata (Graphs) (Contd.)

• A nondeterministic, stack, finite automaton (NDSFA):

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

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

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

initial(q0) \leftarrow.
final(q1) \leftarrow.

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

\( N = 1 \)

\( N = 2 \)

\( N = 3 \)
Recursive Programming: Towers of Hanoi (Contd.)

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

  ![Diagram of 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:

  ![Diagram of Towers of Hanoi](image)

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

  ```prolog
  hanoi(s(0),Orig,Dest,_Help,[move(Orig, Dest)]) <- .
  hanoi(s(N),Orig,Dest,Help,Moves) <-
      hanoi(N,Orig,Help,Dest,Moves1),
      hanoi(N,Help,Dest,Orig,Moves2),
      append(Moves1,[move(Orig, Dest)|Moves2],Moves).
  ```

  - And we simply call this predicate:

    ```prolog
    hanoi_moves(N,Moves) <-
        hanoi(N,a,b,c,Moves).
    ```
Learning to Compose Recursive Programs

- To some extent it is a simple question of practice.
- By 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.