Syntax: Terms (Variables, Constants, and Structures)

- **Variables**: start with uppercase character (or "_"), may include "_" and digits.
  - Examples: X, x, _12345, a_thing, date(Monday, Month, 1994)

- **Constants**: lowercase first character, may include "_" and digits. Also, numbers.
  - Examples: a, b, c, 1.23, '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**. The data structures of a logic program.

A “Hands-on” Introduction to Pure Logic Programming

*Computational Logic*
Syntax: Terms

- Examples of terms:
  - `term` type
  - `dad` constant
  - `time(min, sec)` structure
  - `pair(Calvin, tiger(Hobbes))` structure
  - `Tee(Alf, rob)` illegal

- A good time variable
- Illegal (arity 0)
- `meal` structure (arity 2)
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:
  
  \[
  \begin{align*}
  &\text{pet}(\text{spot}) \leftarrow . \\
  &\text{animal}(\text{spot}) \leftarrow . \\
  &\text{pet}(\text{X}) \leftarrow \text{animal}(\text{X}), \text{barks}(\text{X}). \\
  &\text{animal}(\text{barry}) \leftarrow . \\
  &\text{pet}(\text{X}) \leftarrow \text{animal}(\text{X}), \text{meows}(\text{X}). \\
  &\text{animal}(\text{hobbes}) \leftarrow .
  \end{align*}
  \]

  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:
    
    \[
    \text{some predicate} \leftarrow \ldots \leftarrow \ldots \\
    \]
    
    (i.e., a clause without a head).

    A query represents a question to the program.

  - **Rules:**
    
    - Commas in rule bodies represent conjunction, i.e., \( p \leftarrow p_1, \ldots, p_m \).
    
    - The fact that a rule \( p \leftarrow \) can be seen as the rule \( p \leftarrow \text{true} \).

    Thus, a rule \( p \leftarrow \ldots \leftarrow d \rightarrow \text{true} \) means \( p \rightarrow \ldots \rightarrow d \rightarrow \text{true} \), then \( p \) is true.

    - "Declarative" Meaning of Facts and Rules

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

      - **Facts:** things that are true.
        
        Example: the fact \( \text{animal}(\text{spot}) \leftarrow . \) 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 \( p \leftarrow p_1 \land \cdots \land p_m \).
        
        - "\:\leftarrow" represents as usual logical implication.

        Thus, a rule \( p \leftarrow p_1, \ldots, p_m \) means "if \( p_1, \ldots, p_m \) are true, then \( p \) is true."

      Example: the rule \( \text{pet}(\text{X}) \leftarrow \text{animal}(\text{X}), \text{barks}(\text{X}) \). 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 provide different alternatives.

\[ p \leftarrow p_1, \ldots, p_n \]
\[ q \leftarrow q_1, \ldots, q_m \]

provide different alternatives (for p(X)).

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.

Example:

\[ \text{barks}(\text{spot}). \]
\[ \text{meows}(\text{barry}). \]
\[ \text{roars}(\text{hobbes}). \]

are different.

Note (variable scope): the X vars. in the two classes above are different, despite the same name. Verbs are local to clauses (and are renamed any time a clause is repeated).

A query represents a question to the program.

An example of a logic program:

Execution and Semantics

• 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.

⇒ Intuitively, we have two possible answers: \( X = \text{spot} \) and \( X = \text{barry} \).

⋄ The declarative semantics specifies what should be computed.

⋄ The operational semantics specifies how answers are computed.

(All possible answers.)

Execution:

\[ \text{antwaan} \rightarrow \text{roars} \]
\[ \text{meows} \rightarrow \text{barks} \]
\[ \text{pet} \rightarrow \text{antwaan} \]
\[ \text{pet} \rightarrow \text{antwaan} \]

Example of a logic program:

Predicates: classes in the same predicate.

Declarative "Meaning of Predicates and Queries"
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']).```)
  for “user” files, i.e., files that are not modules.
  - Facts must end with `:-`. The neck (arrow) of rules must be `:-`.

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”):

```
? - use_module(pets).
yes
? - pet(spot).
yes
? - pet(X).
X = spot ;
   X = barry ;
   no
```
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 of literals $A$ and $B$ is asking if they can be made syntactically identical by giving (minimal) values to their variables.
| Only variables can be given values.

| Two structures can be made identical only by making their arguments identical.
| Only variables can be given values.

<table>
<thead>
<tr>
<th>$X = Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$θ$</td>
</tr>
<tr>
<td>$A$</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 a literal contains that variable, because it would create an infinite term. This is known as the occurs check.

- If it is also used to:
  - Return values.
  - Pass parameters.

- It is the mechanism used to resolve calls.

- Unification uses
Unification

- Often several solutions exist, e.g.:

\[
A \theta = T_1, B \theta = T_2
\]

\[
f(X, g(T)) = f(m(H), g(M))
\]

\[
\begin{align*}
X &= m(a), & H &= a, & M &= b, & T &= b \\
f(m(a), g(b)) &= f(m(H), g(f(A)))
\end{align*}
\]

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

\[
A \theta = T_1, B \theta = T_2
\]

\[
f(X, g(T)) = f(m(H), g(M))
\]

\[
\begin{align*}
X &= m(H), & T &= M, & f(m(H), g(M))
\end{align*}
\]

- Always a unique (modulo variable renaming) most general solution exists
- 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 \(E = \emptyset\):
   1. Delete an equation \(T = S\) from \(E\)
   2. Case \(T\) or \(S\) (or both) are non-variable terms:
      * If their names or arities are different
        → halt with failure
      * Obtain the arguments \(\{T_1, \ldots, T_n\}\) of \(T\) and \(\{S_1, \ldots, S_n\}\) of \(S\).
        Add \(\{T_1 = S_1, \ldots, T_n = S_n\}\) to \(E\)
        Add \(T = S\) to \(\theta\)
      3. Case \(T\) and \(S\) are non-variable terms:
         * If their arguments are different
           → halt with failure
         * Add \(\{T_1 = S_1, \ldots, T_n = S_n\}\) to \(E\)
         4. While not \(E = \emptyset\)
            * Add \(T = S\) to \(\theta\)
            * Add \(\theta = \emptyset\) to \(E\)

3. Halt with \(\theta\) being the m.g.u of \(A\) and \(B\).

---

The unification algorithm finds this solution.
- Unless unification fails.
- Always a unique (modulo variable renaming) most general solution exists.
- This is the one that we are interested in.
- These are correct, but a simpler (“more general”) solution exists:

\[
((\forall y, y)z) \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z}
\]

- Open several solutions exist, e.g.:

\[
((\forall y, y)z) \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z} \frac{z = \lambda x y}{x = \lambda z}
\]

-
Unification Algorithm Examples (I)

• Unify:
  \[ A = p(X, X) \] and \[ B = p(f(Z), f(W)) \]

\[ \begin{align*}
  & S \\
  & T \\
  & \theta
\end{align*} \]

\[ (Z'Z)^d = B \text{ and } ((\chi)\chi)^d = V : \text{Unify} \]

• Unify:
  \[ A = p(X, f(Y)) \] and \[ B = p(Z, X) \]

\[ \begin{align*}
  & S \\
  & T \\
  & \theta
\end{align*} \]

Unification Algorithm Examples (I)
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 sols.

Step 2.2 defines alternative paths to be explored to find answer(s); execution explores this tree (for example, breadth-first).

Searching rule(s): how are clauses/branches selected in 2.2?

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).

Example (two valid executions):

Finding solutions in a different order.

Since choosing a different clause (in step 2.2) can lead to different solutions.

If the search rule is not specified, execution is nondeterministic.

Example rule(s): how are clauses/branches selected in 2.2?

Since step 2.2 is left open, a given logic programming system must specify how it

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

Clause

\[ \text{pet}(P) \]

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X).\)  

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X).\)  

\(\text{animal}(\text{spot}) \leftarrow.\)  

\(\text{animal}(\text{barry}) \leftarrow.\)  

\(\text{animal}(\text{hobbes}) \leftarrow.\)  

\(\text{barks}(\text{spot}) \leftarrow.\)  

\(\text{meows}(\text{barry}) \leftarrow.\)  

\(\text{roars}(\text{hobbes}) \leftarrow.\)  

We take C\(\text{\text{1}}\) instead of C\(\text{\text{5}}\) to find a solution.

\(\text{pet}(P).\)  

\(\text{pet}(P)\)  

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X).\)  

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X).\)  

\(\text{animal}(\text{spot}) \leftarrow.\)  

\(\text{animal}(\text{barry}) \leftarrow.\)  

\(\text{animal}(\text{hobbes}) \leftarrow.\)  

\(\text{barks}(\text{spot}) \leftarrow.\)  

\(\text{meows}(\text{barry}) \leftarrow.\)  

\(\text{roars}(\text{hobbes}) \leftarrow.\)  

We take C\(\text{\text{1}}\) instead of C\(\text{\text{5}}\) to find a solution.

\(\text{pet}(P).\)  

\(\text{pet}(P)\)  

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X).\)  

\(\text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X).\)  

\(\text{animal}(\text{spot}) \leftarrow.\)  

\(\text{animal}(\text{barry}) \leftarrow.\)  

\(\text{animal}(\text{hobbes}) \leftarrow.\)  

\(\text{barks}(\text{spot}) \leftarrow.\)  

\(\text{meows}(\text{barry}) \leftarrow.\)  

\(\text{roars}(\text{hobbes}) \leftarrow.\)  

We take C\(\text{\text{3}}\).

System response:  

\(\text{pet}(\text{puppy}) \leftarrow.\)  

\(\text{pet}(\text{puppy})\)  

\(\text{dog}(\text{puppy}) \leftarrow.\)  

\(\text{meows}(\text{puppy}) \leftarrow.\)  

\(\text{barks}(\text{puppy}) \leftarrow.\)  

Go and execute a different branch (i.e., a different choice in C\(\text{\text{3}}\) or C\(\text{\text{5}}\)).

If we type \(\text{;} \) after the \(?\) prompt (i.e., we ask for another solution) the system can explore another branch (i.e., a different choice in C\(\text{\text{3}}\) or C\(\text{\text{5}}\)).

System response:  

\(\text{pet}(\text{puppy}).\)  

\(\text{pet}(\text{puppy})\)  

\(\text{dog}(\text{puppy}) \leftarrow.\)  

\(\text{meows}(\text{puppy}) \leftarrow.\)  

\(\text{barks}(\text{puppy}) \leftarrow.\)  

There is a choice-
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):

- All solutions are at finite depth in the tree.
- Different query → different tree.

The search and computation rules explain how the search tree will be explored.

How can we achieve completeness (guarantee that all solutions will be found)?

- How can we achieve completeness (guarantee that all solutions will be found)?

Characterization of The Search Tree

- Failures can be at finite depth, or in some cases, be an infinite branch.
- All solutions are at finite depth in the tree.

- Failures can be at finite depth, or in some cases, be an infinite branch.
- All solutions are at finite depth in the tree.

The boxes represent the "and" parts (except leaves):

A query + a logic program together specify a search tree.
Depth-First Search

- Used in all the following examples (via Ciao’s dt package).
- But costly in terms of time and memory.
- Will find all solutions before falling through an infinite branch.

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
\[\text{animal}(A), \text{named}(A, \text{Name}).\]
with:
\[\text{animal}(\text{dog(barry)}) \leftarrow . \quad \text{named}(\text{dog(Name)}, \text{Name}) \leftarrow .\]

Also, unification is used to pass parameters in procedure calls and to return values upon procedure exit.

\[\text{pet}(P) \leftarrow \text{animal}(X), \text{barks}(X), \text{grandfather}(L, M) \leftarrow \text{father}(L, N), \text{father}(N, M).\]
\[\text{pet}(\text{spot}) \leftarrow \text{barks}(\text{spot}).\]

In fact, argument positions are not fixed a priori to be input or output. Example: Consider query
\[\text{father}(\text{john}, \text{peter}).\]
\[\text{father}(\text{john}, \text{mary}).\]

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

\[\text{pet}(\text{X}) \leftarrow \text{add}(\text{s(0)}, \text{Y}, \text{Z}).\]
\[\text{pet}(\text{s(0)}), \text{pet}(\text{s(s(0))}).\]

SQL (such as:)

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

\[\text{father}(\text{john}, \text{peter}).\]
\[\text{father}(\text{john}, \text{mary}).\]
\[\text{father}(\text{peter}, \text{michael}).\]
\[\text{mother}(\text{mary}, \text{david}).\]

Given such database a logic programming system can answer questions.

\[\text{father}(\text{john}, \text{peter}).\] Answer: Yes
\[\text{father}(\text{john}, \text{mary}).\] Answer: No
\[\text{father}(\text{john}, \text{X}).\] Answer: \{X = \text{peter}\}
\[\text{father}(\text{john}, \text{X}).\] Answer: \{X = \text{mary}\}
\[\text{grandfather}(\text{X}, \text{michael}).\] Answer: \{X = \text{john}\}
\[\text{grandfather}(\text{X}, \text{Y}).\] Answer: \{X = \text{john}, Y = \text{michael}\}
\[\text{grandfather}(\text{X}, \text{X}).\] Answer: No

Rules for \(\text{grandmother}(\text{X}, \text{Y})\)

\[\text{mother}(\text{X}, \text{Y}) \rightarrow \text{father}(\text{X}, \text{Z}), \text{mother}(\text{Z}, \text{Y}).\]

Also, unification is used to access data and give values to variables.
Database Programming (Contd.)

• Another example:

\[
\begin{align*}
\text{resistor}(\text{power}, n_1) & \leftarrow. \\
\text{transistor}(n_2, \text{ground}, n_1) & \leftarrow. \\
\text{transistor}(n_3, n_4, n_2) & \leftarrow. \\
\text{transistor}(n_5, \text{ground}, n_4) & \leftarrow. \\
\text{inverter}(\text{Input}, \text{Output}) & \leftarrow. \\
\text{nand}_{\text{gate}}(\text{Input}_1, \text{Input}_2, \text{Output}) & \leftarrow. \\
\text{and}_{\text{gate}}(\text{Input}_1, \text{Input}_2, \text{Output}) & \leftarrow.
\end{align*}
\]

Query and set (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:

Structured version:

\[
\begin{align*}
\text{course}(\text{complog}, \text{Wednesday}, 18:30, 20:30, \text{M.\ 'Hermenegildo'}, \text{new}, 5102) & \leftarrow.
\end{align*}
\]

Note: "x = y" is equivalent to "\( \lambda(x).x = y \)"

- course = \text{course}(\text{complog}, \text{Day}, \text{StartH}, \text{StartM}, \text{FinishH}, \text{FinishM}, \text{C}, \text{D}, \text{E}, \text{F}).

Structured data and data abstraction (and the '=' Predicate):

\[
\begin{align*}
\text{course}(\text{complog}, \text{time}, \text{ lecturer}, \text{ location}) & \leftarrow. \\
\text{time} = \text{t(\text{wed}, 18:30, 20:30)}, \text{ lecturer} = \text{lect(\text{M.\ 'Hermenegildo})}, \text{ location} = \text{loc(new, 5102)}.
\end{align*}
\]

Structured version:

\[
\begin{align*}
\text{course}(\text{complog}, \text{time}, \text{ lecturer}, \text{ location}) & \leftarrow. \\
\text{time} = \text{t(\text{wed}, 18:30, 20:30)}, \text{ lecturer} = \text{lect(\text{M.\ 'Hermenegildo})}, \text{ location} = \text{loc(new, 5102)}.
\end{align*}
\]

Note: "\( X = Y \)" is equivalent to "\( '='(X,Y) \)

where the predicate \( '=' \) is defined as the fact "\( '='(X,X) \leftarrow. \)" – Plain unification!

- Equivalent to:

\[
\begin{align*}
\text{course}(\text{complog}, \text{time}, \text{ lecturer}, \text{ location}) & \leftarrow. \\
\text{time} = \text{t(\text{wed}, 18:30, 20:30)}, \text{ lecturer} = \text{lect(\text{M.\ 'Hermenegildo})}, \text{ location} = \text{loc(new, 5102)}.
\end{align*}
\]
Structured Data and Data Abstraction (and The Anonymous Variable)

• Given:
  \[
  \text{course}(\text{complog}, \text{Time}, \text{Lecturer}, \text{Location}) \leftarrow \\
  \text{Time} = t(\text{wed},18:30,20:30), \\
  \text{Lecturer} = \text{lect}(\text{M.},\text{Hermenegildo}), \\
  \text{Location} = \text{loc}(\text{new},5102).
  \]

  \[
  \text{When is the Computational Logic course?} \\ \text{course}(\text{complog}, \text{Time}, \text{A}, \text{B}).
  \]

  The query has solution:

  \[
  \{ \text{Time}=t(\text{wed},18:30,20:30), \text{A}=\text{lect}(\text{M.},\text{Hermenegildo}), \text{B}=\text{loc}(\text{new},5102) \}.
  \]

• Using the anonymous variable (“”):

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

  The query has solution:

  \[
  \{ \text{Time}=t(\text{wed},18:30,20:30) \}.
  \]

The circuit example revisited:

\[
\begin{align*}
\text{resistor}(r1,\text{power},n1) & \leftarrow. \\
\text{transistor}(t1,n2,\text{ground},n1) & \leftarrow. \\
\text{resistor}(r2,\text{power},n2) & \leftarrow. \\
\text{transistor}(t2,n3,n4,n2) & \leftarrow. \\
\text{transistor}(t3,n5,\text{ground},n4) & \leftarrow. \\
\text{inverter}(\text{inv}(T,R),\text{Input},\text{Output}) & \leftarrow. \\
\text{transistor}(T1,\text{Input},X,\text{Output}), \\
\text{resistor}(R,\text{power},\text{Output}). \\
\text{nand_gate}(\text{nand}(T1,T2,R),\text{Input1},\text{Input2},\text{Output}) & \leftarrow. \\
\text{transistor}(T1,\text{Input1},X,\text{Output}), \\
\text{transistor}(T2,\text{Input2},\text{ground},X), \\
\text{resistor}(R,\text{power},\text{Output}). \\
\text{and_gate}(\text{and}(N,I),\text{Input1},\text{Input2},\text{Output}) & \leftarrow. \\
\text{nand_gate}(N,\text{Input1},\text{Input2},X), \\
\text{inverter}(I,X,\text{Output}).
\end{align*}
\]

• The query \(-\text{and}\) gate \(G\), \text{In1}, \text{In2}, \text{Out}\) has solution:

  \[
  \{ G=\text{and}(\text{nand}(t2,t3,r2),\text{inv}(t1,r1)), \text{In1}=n3, \text{In2}=n5, \text{Out}=n1 \}.
  \]

Structured Data and Data Abstraction (Contd.)

When is the Computational Logic course?

\[
\begin{align*}
\text{course}(\text{complog}, \text{Time}, \text{Lecturer}, \text{Location}) & \leftarrow. \\
\text{Time} = t(\text{wed},18:30,20:30), \\
\text{Lecturer} = \text{lect}(\text{M.},\text{Hermenegildo}), \\
\text{Location} = \text{loc}(\text{new},5102).
\end{align*}
\]

The query has solution:

\[
\{ \text{Time}=t(\text{wed},18:30,20:30), \text{A}=\text{lect}(\text{M.},\text{Hermenegildo}), \text{B}=\text{loc}(\text{new},5102) \}.
\]

• Given:

  \[
  \text{course}(\text{complog}, \text{Time}, \text{Lecturer}, \text{Location}) \leftarrow. \\
  \text{Lecturer} = \text{lect}(\text{M.},\text{Hermenegildo}), \\
  \text{Location} = \text{loc}(\text{new},5102), \\
  \text{Time} = t(\text{wed},18:30,20:30).
  \]

  The query has solution:

  \[
  \{ \text{course}(\text{complog}, \text{Time}, \text{Lecturer}, \text{Location}), \text{A}=\text{lect}(\text{M.},\text{Hermenegildo}), \text{B}=\text{loc}(\text{new},5102) \}.
  \]
Logic Programs and the Relational DB Model

Traditional → Codd's Relational Model

File → Relation Table
Record → Tuple Row
Field → Attribute 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>

Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Town</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>London</td>
<td>15</td>
</tr>
<tr>
<td>Brown</td>
<td>York</td>
<td>5</td>
</tr>
<tr>
<td>Jones</td>
<td>Paris</td>
<td>21</td>
</tr>
<tr>
<td>Smith</td>
<td>Brussels</td>
<td>15</td>
</tr>
<tr>
<td>Smith</td>
<td>Santander</td>
<td>5</td>
</tr>
</tbody>
</table>

Lived-in

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

Example:

- person(Brown,20,Male) <-.
- person(Jones,21,Female) <-.
- person(Smith,36,Male) <-.

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

• Example:

<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>

Logic Programs and the Relational DB Model (Contd.)
• The operations of the relational model are easily implemented as rules.

- Derived operations – some can be expressed more directly in LP:
  - Intersection:
    \[
    r \cap s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n).
    \]
  - Join:
    \[
    r \bowtie s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n).
    \]
  - Intersection:
    \[
    r \cap s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n).
    \]

- Duplicates an issue: see "setof" later in Prolog.

- See later for definition of \( \leq \) / (relation).
Deductive Databases

- The subject of “deductive databases” uses these ideas to develop logic-based databases.
  - Often syntactic restrictions (a subset of definite programs) used.
  - Variations of a “bottom-up” execution strategy used: Use the \( \top \) operator explained in the theory part to complete the model, restrict to the query.
  - A “related”, “cousin”, “same generation”, etc.

Recursive Programming

- Example: 

```
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: 

```prolog
ancestor(?, ?) :- parent(?, ?).
ancestor(?, ?) :- parent(?, ?), ancestor(?, ?).
ancestor(?, ?) :- parent(?, ?), ancestor(?, ?), ancestor(?, ?).
```

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

- Type: a (possibly infinite) set of terms.
- Type definition: A program defining a type.
- Example:
  - Weekday:
    - Set of terms to represent: Monday, Tuesday, Wednesday, ...
    - Type definition:
      ```prolog
      is weekday('Monday') <-.
      is weekday('Tuesday') <-.
      ...
      ```
  - Date (weekday * day in the month):
    - Set of terms to represent: date('Monday',23), date('Tuesday',24), ...
    - Type definition:
      ```prolog
      is date(date(W,D)) <- is weekday(W), is day of month(D).
      is day of month(1) <-.
      is day of month(2) <-.
      ...
      ```

Recursive Types: Recursive Programming

- 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:
      ```prolog
      nat(0) <-.
      nat(s(X)) <- nat(X).
      ```
- We can reason about complexity for a given class of queries (with a single body literal).
- 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:
      ```prolog
      integer( X) <- nat(X).
      integer(-X) <- nat(X).
      ```

Recursive Types: Recursive Programming

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

Types
Recursive Programming: Arithmetic

Defining the natural order (≤) of natural numbers:

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

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

Multiple uses:
\[ \text{less} \text{or} \text{equal}(\text{s}(\text{s}(0)), \text{s}(\text{s}(0))), \text{less} \text{or} \text{equal}(X, 0), \ldots \]

Multiple solutions:
\[ \text{less} \text{or} \text{equal}(X, \text{s}(0)), \text{less} \text{or} \text{equal}(\text{s}(\text{s}(0)), Y), \text{etc.} \]

Addition:
\[ \text{plus}(0, X, X) \leftarrow \text{nat}(X). \]

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

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

Multiple solutions:
\[ \text{plus}(X, Y, \text{s}(\text{s}(\text{s}(0)))) \]

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

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

The meaning of \text{plus} is the same if both definitions are combined.
\[ \text{plus}(X, Y, Z) \leftarrow (Z \text{ is } \text{s}(W) \text{ for some } W). \]

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:
\[ \text{times}(X, Y, Z) \]
\[ \text{exp}(N, X, Y) \]
\[ \text{factorial}(N, F) \]

\[ \text{minimum}(N_1, N_2, \text{Min}) \]

...
Recursive Programming: Arithmetic

• Definition of mod(X,Y,Z)
  "Z is the remainder from dividing X by Y" (∃Q s.t. X = Y*Q + Z and Z < Y):
  mod(X,Y,Z) <- less(Z, Y), times(Y,Q,W), plus(W,Z,X).

• Another possible definition:
  mod(X,Y,X) <- less(X, Y).
  mod(X,Y,Z) <- plus(X1,Y,X), 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:
  ackermann(0,N) = N+1
  ackermann(M,0) = ackermann(M-1,1)
  ackermann(M,N) = ackermann(M-1,ackermann(M,N-1))

• In Peano arithmetic:
  (n+1)(m+1) = n(m+1) + m + 1
  ackermann(0,N) = s(N)
  ackermann(s(M),0) = ackermann(M,s(0))
  ackermann(s(M),s(N)) = ackermann(M,ackermann(s(M),N))

• Can be defined as:
  ackermann(N,M,T) <- ackermann(N,M-1,ackermann(N,M,T-1)),
  ackermann(N,0,T) <- ackermann(N-1,s(T)),
  ackermann(s(N),s(M),T) <- ackermann(s(N),M,ackermann(s(N),s(M),T-1)),
  ackermann(s(N),M,0) <- ackermann(N,M-1,s(0)),
  ackermann(0,M,s(N)) <- ackermann(0,s(M),N).

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

• Syntactic support available (see, e.g., the Ciao functions package).

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

  The Ackermann function:

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

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

  \[ \text{ackermann}(N, M, T) \leftarrow \text{ackermann}(N, M-1, \text{ackermann}(N, M, T-1)) \],
  \[ \text{ackermann}(N, 0, T) \leftarrow \text{ackermann}(N-1, s(T)) \],
  \[ \text{ackermann}(s(N), s(M), T) \leftarrow \text{ackermann}(s(N), M, \text{ackermann}(s(N), s(M), T-1)) \],
  \[ \text{ackermann}(s(N), M, 0) \leftarrow \text{ackermann}(N, M-1, s(0)) \],
  \[ \text{ackermann}(0, M, s(N)) \leftarrow \text{ackermann}(0, s(M), N) \].

  \text{Recursive Programming: Arithmetic}
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 \(\cdot\) (which is overloaded).

Syntactic sugar: the term \(\cdot(X,Y)\) is denoted by \([X | Y]\)

Formal object: Cons pair syntax

Element syntax

\[(\cdot(a,\cdot(b,\cdot(c,\cdot))))\]
Recursive Programming: Lists (Contd.)

• X is a member of the list Y:
  - member(a,[a]) <-.
  - member(b,[b]) <-.
  - etc.
  ⇒ member(X,[X]) <-.
  - member(a,[a,c]) <-. member(b,[b,d]) <-.
  - etc.
  ⇒ member(X,[X,Y]) <-.
  - member(a,[a,c,d]) <-. member(b,[b,d,l]) <-.
  - etc.
  ⇒ member(X,[X,Y,Z]) <-.
  - ⇒ member(X,[Y,X]) <- list(Y).
  - member(a,[c,a])
  - member(b,[d,b]).
  - etc.
  ⇒ member(X,[Y,Z,X]) <-.
  - ⇒ member(X,[Y|Z]) <- member(X,Z).

• Resulting definition:
  - member(X,[X|Y]) <- list(Y).
  - member(X,[|T]) <- member(X,T).

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

Resulting definition:

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

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

⋄ Rest of cases (first step):
\[ \text{append}([a], [b], [a, b]) \]
\[ \text{append}([a], [b, c], [a, b, c]) \]

⇒
\[ \text{append}([X], Ys, [X|Ys]) \leftarrow \text{list}(Ys). \]
\[ \text{append}([a, b], [c], [a, b, c]) \]
\[ \text{append}([a, b], [c, d], [a, b, c, d]) \]

⇒
\[ \text{append}([X, Z], Ys, [X, Z|Ys]) \leftarrow \text{list}(Ys). \]

This is still infinite → we need to generalize more.

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

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

• Uses of append:

⋄ Concatenate two given lists:
\[ \text{append}([a, b], [c], Z) \]

⋄ Find differences between lists:
\[ \text{append}(X, [c], [a, b, c]) \]

⋄ Split a list:
\[ \text{append}(X, Y, [a, b, c]) \]

Second generalization:

This is still infinite → we need to generalize more.

• Real of cases (first step):

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

• Base case:
\[ \text{append}([], Ys, Ys) \leftarrow \text{list}(Ys). \]
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:
  - reverse([X|Xs],Ys) <- reverse(Xs,Zs), append(Zs,[X],Ys).
  - How can we stop?
  - reverse([],[]) <-.

- As defined, reverse(Xs,Ys) is very inefficient. Another possible definition:
  - reverse(Xs,Ys) <- reverse(Xs,[],Ys).
  - reverse([],Ys,Ys) <-.
  - reverse([X|Xs],Acc,Ys) <- reverse(Xs,[X|Acc],Ys).

Recursive Programming: Binary Trees

- Represented by a ternary functor:
  - tree(Element,Left,Right).
  - Empty tree represented by void:
  - binary_tree(void) <-.
  - binary_tree(tree(Element,Left,Right)) <- binary_tree(Left), binary_tree(Right).

- Defining tree_member(Element,Tree):
  - binary_tree(tree(X,Left,Right)) <- binary_tree(Left), binary_tree(Right).
  - tree_member(X,tree(X,Left,Right)) <-.
  - tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Left).
  - tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Right).

Find the differences in terms of efficiency between the two definitions.

- reverse([X|Xs],Ys) <- reverse(Xs,X[|Ys],X[|Ys]).
  - reverse([],Ys) <-.
  - reverse([X|Xs],Ys) <- reverse(Xs,Ys). reverse(Xs,Ys) <- reverse(Xs,[]).
  - As defined, reverse(Xs,Ys) is very inefficient. Another possible definition:
  - reverse([],Ys) <-.
  - How can we stop?
  - reversed(0,Xs) <- reversed(Xs,0).
  - reversed(0,Xs) <- reversed(Xs,0).
  - reversed([X|Xs],Ys) <- reversed(Xs,Ys).
  - reversed([X|Xs],Ys) <- reversed(Xs,Ys).

Recursive Programming: Lists (Contd.)
Recursive Programming: Binary Trees

• Defining preorder(Tree,Order):
  
  \[
  \text{pre-order}(\text{void},[]) \leftarrow
  \]
  
  \[
  \text{pre-order}(\text{tree}(X,\text{Left},\text{Right}),\text{Order}) \leftarrow
  \]
  
  \[
  \text{pre-order}(	ext{Left},\text{OrderLeft}),
  \]
  
  \[
  \text{pre-order}(	ext{Right},\text{OrderRight}),
  \]
  
  \[
  \text{append}([X\mid\text{OrderLeft}],\text{OrderRight},\text{Order}).
  \]

• Define inorder(Tree,Order), postorder(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:
  
  \[
  \text{type} \text{tree} = \text{^treerec};
  \text{treerec} = \text{record}
  \]
  
  \[
  \text{data} : \text{integer};
  \]
  
  \[
  \text{left} : \text{tree};
  \]
  
  \[
  \text{right} : \text{tree};
  \]
  
  \[
  \text{end};
  \]
  
  \[
  \text{var} t : \text{tree};
  \]
  
  \[
  ...\]

Recursive Programming: Binary Trees
Polymorphism

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

```prolog
lt_member(X, [X|Y]) <- list(Y).
lt_member(X, [_|T]) <- lt_member(X, T).
lt_member(X, tree(X, L, R)) <- binary_tree(L), binary_tree(R).
lt_member(X, tree(Y, L, R)) <- lt_member(X, L).
lt_member(X, tree(Y, L, R)) <- lt_member(X, R).
```

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

- Also, try (somewhat surprising):

```prolog
?- lt_member(X, [b,a,c]).
X = b ; X = a ; X = c

?- lt_member(X, tree(b, tree(a, void, void), tree(c, void, void))).
X = b ; X = a ; X = c
```

- Also, try (somewhat surprising):

```prolog
?- lt_member(M, T).
```

### Recognizing Polynomials

- 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 in `X`:

```prolog
polynomial(X, X) <-.
polynomial(Term, X) <- pconstant(Term),
polynomial(Term1+Term2, X) <- polynomial(Term1, X), polynomial(Term2, X),
polynomial(Term1-Term2, X) <- polynomial(Term1, X), polynomial(Term2, X),
polynomial(Term1*Term2, X) <- polynomial(Term1, X), polynomial(Term2, X),
polynomial(Term1/Term2, X) <- polynomial(Term1, X), pconstant(Term2),
polynomial(Term1^N, X) <- polynomial(Term1, X), nat(N).
```

Recursive Programming: Manipulating Symbolic Expressions

- A term `X` is a polynomial if its member can be used simultaneously with its non-member:

```prolog
?- member(X, tree(X, L, R)).
```

- Also, try (somewhat surprising):

```prolog
X = a ; X = b ; X = c
```

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

```prolog
?- member(X, tree(X, L, R)).
```

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

```prolog
?- member(X, tree(X, L, R)).
```

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

```prolog
?- member(X, tree(X, L, R)).
```
Recursive Programming: Manipulating Symbolic Expressions (Contd.)

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

- deriv(X, X, s(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).

- ...

- A simplification step can be added.

- Programs can be added to simplify the expressions.

- A non-deterministic finite automaton (NFA):

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

  - Where q0 is both the initial and final state.

  - Recursively generating the sequence of characters accepted by the following:

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

    - Programs:

      - initial(q0) <-
      - delta(q0, a, q1) <-
      - delta(q1, b, q0) <-
      - final(q0) <-
      - delta(q1, b, q1) <-

      - accept(S) <- initial(Q), accept_from(S, Q).

      - accept_from([], Q) <- final(Q).

      - accept_from([X | Xs], Q) <- delta(Q, X, NewQ), accept_from(Xs, NewQ).

- Symbolic differentiation: deriv(Expression, X, DifferentiatedExpression (Contd.))
Recursive Programming: Automata (Graphs) (Contd.)

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

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

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

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

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

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

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

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

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

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

• 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.

○ A nondecreasing stack-finite automaton (NDSFA):

\[
\text{accept} \rightarrow \text{Accept}(q, [], S, \text{Accept}(S, q, [])).
\]

\[
\text{delta} \rightarrow \text{delta}(q, S, \text{delta}(S, q, [])).
\]

\[
\text{final} \rightarrow \text{final}(q, []).\]

Recursive Programming: Automata (Graphs) (Contd.)
We will call the main predicate \( hanoi \) moves \( (N, \text{Moves}) \) where

\[
N \text{ is the number of disks and } \text{Moves} \text{ the corresponding list of "moves".}
\]

Each move \( \text{move}(A, B) \) represents that the top disk in \( A \) should be moved to \( B \).

Example:

Each move \( \text{move}(A, B) \) represents that the top disk in \( A \) should be moved to \( B \).

\( N \) is the number of disks and \( \text{Moves} \) the corresponding list of "moves".

We will call the main predicate \( hanoi \) moves \( (N, \text{Moves}) \).
Learning to Compose Recursive Programs

• To some extent it is a simple question of practice.

• Solve the pieces
• Break it down into subproblems
• State the general problem

Global top-down design approach

• Sometimes it helps to look at well-written examples and use the same “schemas”.

• 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.

• The way most people do it.

Best approach: practice.