

Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan
Störmer



[The art of Prologue: 1.4 - 2.3, 3 - 3.2]

Common instances

- C is a **common instance** of A and B
 - If C is an instance of A
 - And an instance of B is
- This means that there are θ_1 and θ_2 , so that $C = A\theta_1$ and $C = B\theta_2$
- Example
 - The targets A : `plus(0,3,Y)`. and B : `plus(0,X,X)`. have a common instance C : `plus(0, 3, 3)`.
 - $\theta_1 = \{Y=3\}$
 - $\theta_2 = \{X=3\}$

Common instances

- Meaning of a **non-basic term**
 - Given a non-basic query and a program of **universally quantified facts**
 - Find a basic fact that is a **common instance** of the query and the fact
 - Is there a common instance,
 - Then the answer is yes and the substitution that leads from the query to the common instance is the solution
 - There is no common authority,
 - Then the answer is no
- Answering an existence query with a universal fact:
 1. The common instance is derived from the fact by **instantiation**
 2. The query is derived from the common instance by **generalization**

Conjunctive queries

- Conjunctive queries string targets together
- Significance of the program: Have all objectives been met?
- General
 - $?- Q_1 , \dots , Q_n .$
 - The comma (,) stands for the conjunction, i.e. a logical AND
 - The comma should not be confused with the comma in the enumeration of arguments
 - Simple queries are a special case: Conjunction of a single target

Conjunctive queries

- If all targets are **basic terms**
 - Answer: Is each target of the query implied by the program?
 - Example
 - `?- father(abraham,isaac), male(lot).`
- Yes
- If **non-basic terms** occur as targets
 - Are there instances of the targets that are each implied by the program, whereby identical variables must always be substituted in the same way
 - `?- father(terach,X), father(X,Y).`
- Yes {X = abraham, Y = isaac}

Conjunctive queries

- Shared variables
 - Variables that occur in several targets of the conjunctive query
- The scope of a logical variable is the entire expression of conjunctive queries
- General:
 - Given an expression $?- p(X), q(X)$.
 - Meaning: "Is there an X such that both $?- p(X)$. and $?- q(X)$. are fulfilled?"

Shared variables

- Split variables restrict a query
- Example:
 - `?- father(haran,X), male(X).`
 - Solutions to the query `?- father(haran,X).` are restricted to male objects
 - Alternatively: Solutions to the query `?- male(X).` are restricted to objects whose father is haran

Shared variables

- Example:
 - ?- father(terach,X), father(X, Y).
 - Children of terach are restricted to objects that are themselves fathers
 - Alternatively: Determine the objects Y whose father is a child of terach
 - Solution: the grandchildren of terach (in Y)
 - {X = abraham, Y = isaac}, {X = haran, Y = lot}.

Conjunctive queries

- Given a **conjunctive query** and a program P
 - The query is a logical consequence of P ,
 - if all goals in the conjunction are a logical consequence of P , and
 - all shared variables in all targets are substituted by the same terms.
- **Meaning of conjunctive queries**
 - Given a query A_1, A_2, \dots, A_n and a program P
 - Find a substitution θ such that $A_1 \theta$ and $A_2 \theta$ and ... and $A_n \theta$ are basic instances of terms in P
 - Applying the same substitution in all targets ensures that variables are substituted consistently

Rules

- Define new relations based on existing ones

- General

- $A :- B_1, \dots, B_n.$
- A is the **rule header**
- B_1, \dots, B_n is the **standard body**

- Example

- $\text{son}(X,Y) :- \text{father}(Y,X), \text{male}(X).$
- $\text{daughter}(X,Y) :- \text{father}(Y,X), \text{female}(X).$
- $\text{grandfather}(X,Y) :- \text{father}(X,Z), \text{father}(Z,Y).$

Something is still missing.

Or Y is the mother of X ... We are still learning disjunctions.

Rules: Procedural view

- In a query that uses a rule, the rule call is replaced by the rule body
- Example:
 - $\text{grandfather}(X,Y) \text{ :- father}(X,Z), \text{ father}(Z,Y).$
 - "To answer the question whether X is the grandfather of Y, answer the conjunctive query whether X is the father of Z and Z is the father of Y."
 - $\text{?- grandfather}(X, \text{isaac}).$



$\text{?- father}(X,Z), \text{ father}(Z, \text{isaac}).$

Rules: Declarative view

- When defining a rule, `:-` is used
 - Stands for \leftarrow or "**logical implication**"
- Example
 - `grandfather(X,Y) :- father(X,Z), father(Z,Y).`
 - "For all X, Y and Z, if X is the father of Z and Z is the father of Y, then X is the grandfather of Y"
- **Formal**: all variables in a rule are universally quantified (see example above)
- **Intuitively**: we can also say that variables that only occur in the body are existentially quantified
 - "For all X and Y, X is the grandfather of Y if a Z exists, so that X is the father of Z and Z is the father of Y."

Horn clause

- **"Horn clause"**
 - $A :- B_1, \dots, B_n.$
 - Common spelling for:
 - Rules ($n > 0$)
 - Facts ($n = 0$)
 - Queries (A is the result, $n > 0$)
 - Variables are universally quantified
 - Scope of variables is the entire Horn clause

Mode Ponens

- Law of the **universal Modus Ponens**

- Informal: "If a predicate P implies a predicate Q and P is true, then Q is also true."

- **Given**

- A rule R with $A :- B_1, \dots, B_n$
- The facts
 - B_1'
 - ...
 - B_n'
- Then: A' can be derived if
 - $A' :- B_1', \dots, B_n'$ is an instance of R

- General deduction rule, includes:

- Identity (query and fact)
- Instantiation (basic query, universally quantified facts)
 - "Search for a fact of which the query is an instance"

Logic programs

- A logic program is a finite **set of rules**
- An **existentially quantified goal (G)** is the logical consequence of a program (P), precisely if G can be derived by a finite number of applications of the Ponen mode

Logic programs

• Example

- $\text{son}(X,Y) \text{ :- father}(Y,X), \text{male}(X).$
- $\text{?- son}(S, \text{haran}).$

Universal mode Ponens
implies the query with
solution $\{S=\text{lot}\}$

Facts in our program

Substitution $\{S=\text{lot}\}$

- $\text{son}(\text{lot}, \text{haran}) \text{ :- father}(\text{haran}, \text{lot}), \text{male}(\text{lot}).$

Substitution $\{X=\text{lot}, Y=\text{haran}\}$

- $\text{son}(X,Y) \text{ :- father}(Y,X), \text{male}(X).$

Logic programs

- Several rules can be specified for a relation
 - Rules form alternatives (disjunction)
- The set of rules for the same relation (the same predicate) is called a "procedure"
- Example
 - $\text{son}(X,Y) \text{ :- father}(Y,X), \text{male}(X).$
 - $\text{son}(X,Y) \text{ :- mother}(Y,X), \text{male}(X).$

Auxiliary relations

- Simplification of relations through auxiliary relations
 - Avoidance of redundancy/simplification of enumerations
- Example
 - Cumbersome
 - `grandparent(X,Y) :- father(X,Z), father(Z,Y).`
 - `grandparent(X,Y) :- father(X,Z), mother(Z,Y).`
 - `grandparent(X,Y) :- mother(X,Z), father(Z,Y).`
 - `grandparent(X,Y) :- mother(X,Z), mother(Z,Y).`
 - Simpler
 - `parent(X, Y) :- father(X, Y).`
 - `parent(X,Y) :- mother(X,Y).`
 - `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`

Meaning of Prologue

- Description of the semantics of Prolog by abstract interpreter
 - Given a program and a query
 - Answer:
 - Yes: The program implies the query
 - No: The program does not imply the query
 - No response: Interpreter does not terminate if the query cannot be derived in a finite number of steps

Abstract interpreter

- **Basic idea**

- **Resolvent**: Target to be derived in the current step
 - In general: Conjunctive query
- **Reduction**: Calculation of the next resolvent by applying the Ponon mode
 - Reduction of a target G under a program P :
 - Replace G with the body of an instance of a rule in P
 - Whereby the head of the rule is identical to G
- **Trace**: Sequence of the resolvents calculated during the evaluation

- Here: **Restriction to basic queries**

Abstract interpreter

- Given a basic query G and a program P
- **Algorithm**
 - Initialize the resolvent to G
 - As long as the resolvent is not empty
 - Select a target A from the resolvent
 - Select a basic instance $A' :- B_1, \dots, B_n$ of a clause from P
 - So that A and A' are identical
 - If no such instance exists, end the algorithm with the answer "No"
 - Replace A in the resolvent with B_1, \dots, B_n
 - Resolvent empty: end the algorithm with the answer "Yes"

Abstract interpreter

- Program:

- father(abraham,isaac).
- father(haran,lot).
- father(haran,milcah).
- father(haran,yiscah).
- male(isaac).
- male(lot).
- female(milcah).
- female(yiscah).
- son(X, Y) :- father (Y, X), male (X) .
- daughter(X, Y) :- father (Y, X), female (X).

- G: ?- son(lot,haran).

- Interpretation:

- Initialization
 - Resolvent: son(lot,haran)
- Resolvent is not empty

Abstract interpreter

• Program:

- father(abraham,isaac).
- father(haran,lot).
- father(haran,milcah).
- father(haran,yiscah).
- male(isaac).
- male(lot).
- female(milcah).
- female(yiscah).
- son(X, Y) :- father (Y, X), male (X) .
- daughter(X, Y) :- father (Y, X), female (X).

• G: ?- son(lot,haran).

• Interpretation:

- Resolvent: son(lot,haran)
- Select target from the resolvent
 - son(lot,haran)
- Select basic instance of a rule P
 - son(lot,haran) :- father(haran,lot), male(lot)
- Replace target of the resolvent with control body
 - New resolvents: father(haran,lot), male(lot)
- Resolvent is not empty

Only goal

"Guessing" a suitable rule

Abstract interpreter

- Program:

- father(abraham,isaac).
- father(haran,lot).
- father(haran,milcah).
- father(haran,yiscah).
- male(isaac).
- male(lot).
- female(milcah).
- female(yiscah).
- son(X, Y) :- father (Y, X), male (X) .
- daughter(X, Y) :- father (Y, X), female (X).

- G: ?- son(lot,haran).

- Interpretation:

- Resolvente: father(haran,lot), male(lot)
- Select target from the resolvent
 - father(haran,lot)
- Select basic instance of a rule P
 - father(haran,lot)
- Replace target of the resolvent with control body
 - New resolvent: male(lot)
- Resolvent is not empty

Fact: Control body is empty

Abstract interpreter

- Program:

- father(abraham,isaac).
- father(haran,lot).
- father(haran,milcah).
- father(haran,yiscah).
- male(isaac).
- male(lot).
- female(milcah).
- female(yiscah).
- son(X, Y) :- father (Y, X), male (X) .
- daughter(X, Y) :- father (Y, X), female (X).

- G: ?- son(lot,haran).

- Interpretation:

- Resolvent: male(lot)
- Select target from the resolvent
 - male(lot)
- Select basic instance of a rule P
 - male(lot)
- Replace target of the resolvent with control body
 - New resolvents:
- Resolvent is empty


Result: Yes

Abstract interpreter

- Elections
 - Selection of the target from the resolvent is arbitrary
 - Selecting the rule is difficult
 - In general, all rules and all basic facts must be tried out
 - Several rules can be considered
- Real Prolog implementations do not have to guess the selection of the rule
 - Is outside the scope of the event



Programming in Prolog



```
1((\Phi^2x^2 - y^2)(\Phi^2y^2 - z^2)(\Phi^2z^2 - x^2) - 1  
perspective=central;  
spec_p=150.0;  
radius=10.0;  
sextic=rotate(  
    sextic,-0.1,xAxis);
```

Compare

- Comparing terms in queries
 - Variables are substituted by terms
 - Without comparison: Try out all possible terms
 - With comparison: Restriction of the tested terms
- Example
 - `brother(Brother, Sib) :- parent(Parent, Brother), parent(Parent, Sib), male(Brother).`
 - `?- brother(isaac, isaac).` results in Yes
 - But this does not correspond to the intuition behind the relation brother
 - Restriction Brother and Sib must be different objects
 - `brother(Brother, Sib) :- parent(Parent, Brother), parent(Parent, Sib), male(Brother), Brother \= Sib.`

Restriction by comparison

Comments on the description of relations

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

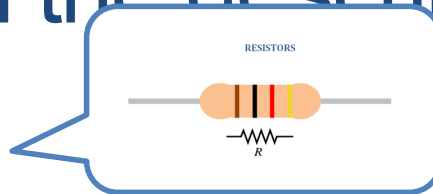
```
/*inverter (input, output) :-  
    Output is the inversion of Input.*/  
inverter(Input,Output) :- transistor(Input,ground,Output),  
resistor(power,Output) .
```

```
/*nand_gate(Input1,Input2,Output) :-  
    Output is the logical nand of Input 1 and Input2.*/  
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),  
transistor(Input2,ground,X),resistor(power,Output) .
```

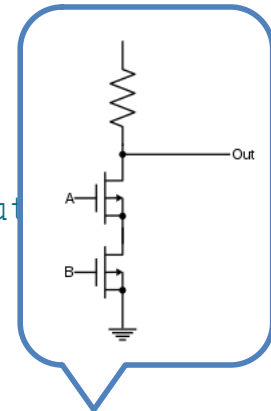
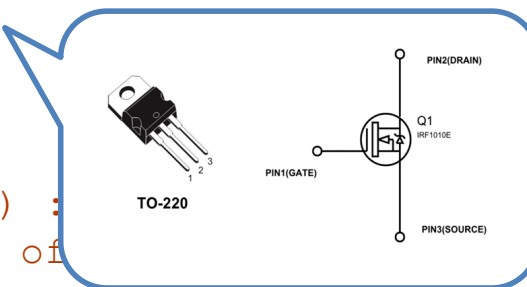
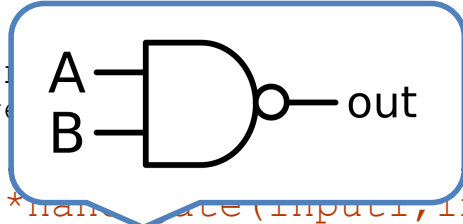
```
/*and_gate (Input1,Input2,Output) :-  
    Output is the logical and of Input1 and Input2.*/  
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),  
inverter(X,Output) .
```

Comments on the description of relations

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



```
/*inverter (input, output) :-
inverter(input,output) :- transistor(input,ground,output).
```



```
/*nand_gate(input1,input2,output) :-
Output is the logical nand of Input 1 and Input2.*/
nand_gate(input1, input2, output) :- transistor(input1,X,output),
transistor(input2,ground,X),resistor(power,output).
```

```
/*and_gate (input1,input2,output) :-
Output is the logical and of Input1 and Input2.*/
and_gate(input1, input2, output) :- nand_gate(input1,input2,X),
inverter(X,output).
```

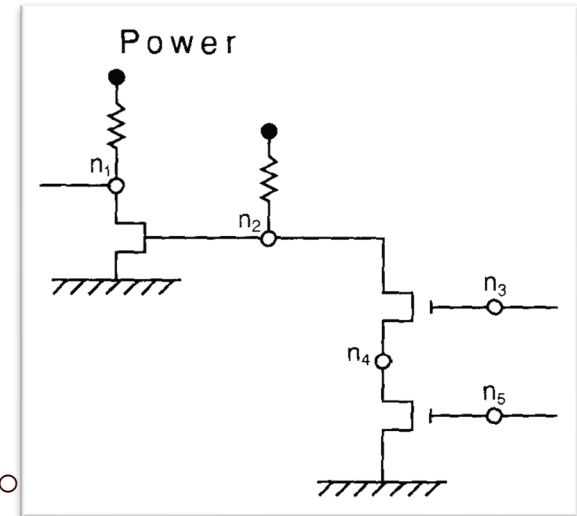
Comments on the description of relations

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

```
/*inverter (input, output) :-
   Output is the inversion of Input.*/
inverter(Input,Output) :- transistor(Input,ground,X),
resistor(power,X) .
```

```
/*nand_gate(Input1,Input2,Output) :-
   Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,X) .
```

```
/*and_gate (Input1,Input2,Output) :-
   Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output) .
```



Comments on the description of relations

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

```
/*inverter (input, output) :-  
    Output is the inversion of Input.*/
```

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

```
/*nand_gate(Input1,Input2,Output) :-
```

```
    Output is the logical nand of Input 1 and Input2.*/  
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),  
transistor(Input2,ground,X),resistor(power,Output) .
```

```
/*and_gate (Input1,Input2,Output) :-
```

```
    Output is the logical and of Input1 and Input2.*/  
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),  
inverter(X,Output) .
```

For each procedure:
Scheme of the
predicate followed by
textual description

Comments on the description of relations

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

```
/*inverter (input, output) :-
   Output is the inversion of Input.*/
inverter(Input,Output) :- transistor
resistor(power,Output).
```

```
/*nand_gate(Input1,Input2,Output) :-
   Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).
```

```
/*and_gate (Input1,Input2,Output) :-
   Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

?- and_gate(In1,In2,Out).

Yes.

In1 = n3, In2 = n5, Out = n1
Solution: and_gate(n3,n5,n1)

Data structures

- Fact database only implicitly represents data structure
- We want to create a meaningful description of the data
- Convention:
 - Adding a first argument to all destinations in the fact database
 - Basic facts: Identifier
 - Non-basic facts: compound term

Data structures

```

/*resistor(R,Node1,Node2) :-
   R is a resistor between Node1 and
   Node2.*/
resistor(r1,power,n1).
resistor(r2,power,n2).

/*transistor(T,Gate,Source,Drain) :-
   T is a transistor whose gate is
   Gate,
   source is Source, and drain is
   Drain.*/
transistor(t1,n2,ground,n1).
transistor(t2,n3,n4,n2).
transistor(t3,n5,ground,n4).

/*inverter(I,Input, Output) :-
   I is an inverter that inverts Input
   to Output.*/
inverter(inv(T,R),Input,Output) :-
   transistor(T,Input,ground,Output),
   resistor(R,power,Output).

```

```

/*nand_gate(Nand,Input1,Input2,Output
)
   :- Nand is a gate forming the
   logical
   and, Output, of Input1 and
   Input2.*/
nand_gate(nand(T1,T2,R),Input1,Input2
,Output) :-
   transistor(T1,Input1,X,Output),
   transistor(T2,Input2,ground,X),
   resistor(R,power,Output).

/*and_gate (And,Input1 ,Input2,
Output) :-
   And is a gate forming the logical
   and,
   Output, of Input1 and Input2.*/
and_gate(and(N,I),Input1,Input2,
output) :-
   nand_gate(N,Input1,Input2,X),
   inverter(I,X,Output).

```

Data structures

```
/*resistor(R,Node1,Node2,Power,Input1,Input2,Output)
   R is a resistor between Node1 and Node2, with Power
   Node2.*/
```

First argument: Variable, a meaningful description of the target

```
resistor(r1,power,n1).
resistor(r2,power,n2).
```

```
/*transistor(T,Gate,Source,Drain,Input1,Input2,X,Output)
   T is a transistor whose gate is Gate,
   source is Source, and drain is Drain.*/
```

Basic fact: Identifier

```
transistor(t1,n2,ground,n1).
transistor(t2,n3,n4,n2).
transistor(t3,n5,ground,n4).
```

and (T1,T2,R), Input1, Input2

```
transistor(T1,Input1,X,Output),
transistor(T2,Input2,ground,X),
resistor(R,power,Output).
```

```
/*and_gate (And,Input1 ,Input2,
Output) :-
```

```
/*inverter(I,Input, Output)
   I is an inverter that takes Input
   to Output.*/
```

Non-basic fact: Structure is specified in the rule header as a compound term

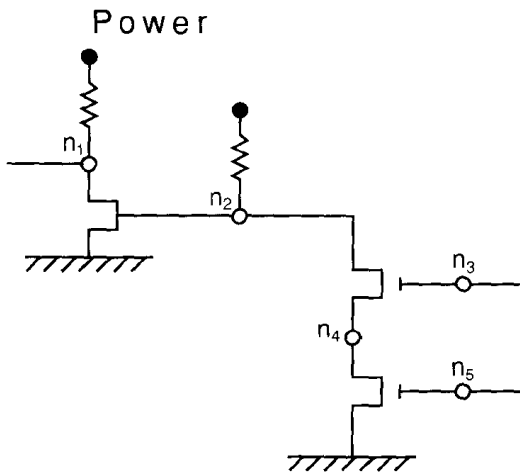
```
inverter(inv(T,R),Input,Output).
transistor(T,Input,X,Output).
resistor(R,power,Output).
```

Non-basic fact: Subterms are bound in control bodies

the logical
Input2.*/
1, Input2,

2, X),

Data structures



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

Data abstraction

- Facts can be represented by nested relations
 - Breakdown into meaningful partial facts
 - Better abstraction of data representation
- Example
 - Instead of
 - `course(complexity, monday, 9, 11, david, harel, feinberg, a).`
 - With data abstraction
 - `course(complexity, time(monday, 9, 11), lecturer(david, harel), location(feinberg, a)).`

Time, lecturer and
location also as separate
facts

Data abstraction

```
lecturer(Lecturer, Course) :-  
    course(Course, Time, Lecturer, Location) .  
  
duration(Course, Length) :-  
    course(Course, time(Day, Start, Finish), Lecturer, Location) ,  
    plus(Start, Length, Finish) .  
  
teaches(Lecturer, Day) :-  
    course(Course, time(Day, Start, Finish), Lecturer, Location) .  
  
?- lecturer(L, complexity) .
```

I don't need to know
what qualities a
lecturer has.

Yes.
L = lecturer(david, harel)

Recursive rules

- Logic rules are also self-similar
- Example

```
grandparent (Ancestor, Descendant) :-
```

```
parent (Ancestor, Person), parent (Person, Descendant) .
```

```
greatgrandparent (Ancestor, Descendant) :-
```

```
parent (Ancestor, Person), grandparent (Person, Descendant) .
```

```
greatgreatgrandparent (Ancestor, Descendant) :-
```

```
parent (Ancestor, Person), greatgrandparent (Person, Descendant) .
```


Recursive rules

- Means to avoid this redundancy: Recursive definition
 - `ancestor(Ancestor, Descendant) :-
parent(Ancestor, Person), ancestor(Person, Descendant).`
- Non-recursive rule required for recursion termination
- Rules represent logical facts!
 - A fact `ancestor(X,X).` would lead to a recursion termination, but would also state that a person is their own ancestor.

```
/*ancestor (Ancestor, Descendant) :-  
  Ancestor is an ancestor of Descendant.*/  
ancestor(Ancestor, Descendant) :-  
    parent(Ancestor, Descendant).  
ancestor(Ancestor, Descendant) :-  
    parent(Ancestor, Person), ancestor(Person, Descendant).
```

Recursive programming: Arithmetic

- The number 0 is a natural number
- The successor of a natural number is a natural number

```
/*natural_number (X) :-  
    X is a natural number.*/  
natural_number(0).  
natural_number(s(X)) :- natural_number(X).
```

- Natural numbers can be represented as
 - 0, s(0), s(s(0)), etc.
 - Abbreviation $s^n(0)$ for n applications of the rule s(X) to 0

Arithmetic

```
/* X '<=' Y :-  
    X and Y are natural numbers,  
    such that X is less than or equal to Y.*/  
'<=' (0,X) :- natural_number(X).  
'<=' (s(X),s(Y)) - '<=' (X,Y).  
  
/* plus(X,Y,Z) :-  
    X, Y, and Z are natural numbers,  
    such that Z is the sum of X and Y.*/  
plus(0,X,X) :- natural_number(X).  
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Lists

- Lists as a recursive data structure
 - The first argument is an element of the list
 - The second argument is the rest of the list
 - Constant symbol for canceling the list: `[]`
- Spellings
 - cons cell
 - `[a | []]`
 - `[a | [b | []]]`
 - `[a | [b | [c | []]]]`
 - `[a | X]`
 - `[a | [b | X]]`
 - Element syntax
 - `[a]`
 - `[a, b]`
 - `[a, b, c]`
 - `[a | X]`
 - `[a, b | X]`

Definition of list

```
/* list(Xs) :- Xs is a list. */  
list([]).  
list([X | Xs]) :- list(Xs).
```

Procedures via lists

```
/*member (element,list) :-  
    Element is an element of the list List.*/  
member (X, [X|Xs]) .  
member (X, [Y|Ys]) :- member (X, Ys) .
```

- The member procedure can be used in various ways:
 - Does the list [a,b,c] contain a b?
 - ?- **member** (b, [a, b, c]) .
 - Let X be an element from the list [a,b,c]
 - ?- **member** (X, [a, b, c]) .
 - Let X be a list containing the element b
 - ?- **member** (b, X) .

Procedures via lists

```
/* prefix (Prefix, List) :-  
   Prefix is a prefix of List.*/  
prefix([], Ys).  
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).
```

```
/* suffix (Suffix, List) :-  
   Suffix is a suffix of List.*/  
suffix(Xs, Xs).  
suffix(Xs, [Y|Ys]) :- suffix(Xs, Ys).
```

```
/* append (Xs, Ys, Zs) :-  
   Zs is the result of concatenating the  
   lists Xs and Ys.*/  
append([], Ys, Ys).  
append([X | Xs], Ys, [X | Zs]) :-  
    append(Xs, Ys, Zs).
```