Computational Logic

A “Hands-on” Introduction to (Pure) Logic Programming

Note: slides with executable links. Follow the run example ⟼ links to execute the example code.

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

- *Examples of terms*: (using Prolog notation conventions)

Term	*Type*	*Main functor:*
`dad`	constant	`dad/0`
`time(min, sec)`	structure	`time/2`
`pair(Calvin, tiger(Hobbes))`	structure	`pair/2`
`Tee(Alf, rob)`	illegal	—
`A_good_time`	variable	—

- A variable is **free** if it has not been assigned a value yet.
- A term is **ground** if it contains no free variables.
- *Functors* can be defined as *prefix*, *postfix*, or *infix* *operators* (just syntax!):

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

We assume that some such operator definitions are always preloaded.

Syntax: Rules and Facts (Clauses)

- **Fact:** an expression of the form $p(t_1, t_2, \ldots, t_n).$ where the t_i are *terms*.
- **Rule:** an expression of the form:

$$\begin{array}{ll} p_0(t_1, t_2, \ldots, t_{n_0}) \texttt{ :- } & p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1), \\ & \ldots \\ & p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m). \end{array}$$

  - ◇ $p_0(...)$ 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**.
  - ◇ `:-` is called the **neck** of the rule.

Example:

```
meal(soup, beef, coffee).                  % <- A fact.
meal(First, Second, Third) :-              % <- A rule.
     appetizer(First),                     %
     main_dish(Second),                    %
     dessert(Third).                       %
```

- A fact is a rule with an empty (true) body (i.e., equiv to $p(t_1, t_2, \ldots, t_n)$:- $true$.).
- 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(tim).
  pet(X) :- animal(X), barks(X).           animal(spot).
  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: $?\text{-}p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_m}^n).$
  (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.`” can be seen as the rule “ `p :- true.` ”)

  Example: the fact `animal(spot).`
  can be read as “spot is an animal”.

- **Rules:**
  - ◇ Commas in rule bodies represent conjunction, and “`:-`” represents logical implication (backwards, i.e., if).
  - ◇ i.e., p `:-` $p_1, \cdots, p_m$. represents $p \leftarrow p_1 \wedge \cdots \wedge p_m$.

  Thus, a rule p `:-` $p_1, \cdots, p_m$. means “if p_1 and . . . and p_m are true, then p is true”

  Example: the rule `pet(X):- animal(X), barks(X).`
  can be read as “X is a pet if it is an animal and it barks”.

- Variables in facts and rules are universally quantified, $\forall$ (recall *clausal form*!).

“Declarative” Meaning of Predicates and Queries

- **Predicates**: clauses in the same predicate

  ```
  p :- p1, ..., pn
  p :- q1, ..., qm
  ...
  ```

  provide different *alternatives* (for p).

  Example: the rules

  ```
  pet(X) :- animal(X), barks(X).
  pet(X) :- animal(X), meows(X).
  ```

  express two *alternative* 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:

  ```
  ?- pet(spot).
  ```

  Asks: Is spot a pet?

  ```
  ?- pet(X).
  ```

  Asks: “Is there an X which is a pet?”

"Execution" and Semantics

- Example of a **logic program**: run example ⟼

```
pet(X) :-  animal(X), barks(X).
pet(X) :-  animal(X), meows(X).
animal(tim).        barks(spot).
animal(spot).       meows(tim).
animal(hobbes).     roars(hobbes).
```

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

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

  ◇ The **declarative semantics** specifies *what* should be computed (all possible answers).

  ⇒ Intuitively, we have two possible answers: `X = spot` and `X = tim`.

  ◇ The **operational semantics** specifies *how* answers are computed (which allows us to determine *how many steps* it will take).

Running Programs in a Logic Programming System

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

```
Ciao X.Y-...
?- use_module(pets).
yes
?- pet(spot).
yes
?- pet(X).
X = spot ? ;
X = tim ? ;
no
?-
```

See the part on Developing Programs with a Logic Programming System
for more details on the particular system used in the course (Ciao).

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* $\texttt{p :- p}_1\texttt{,...,p}_m\texttt{.}$ means:
  "to execute a call to $\texttt{p}$ you have to *call* $\texttt{p}_1$ and ...and $\texttt{p}_m$"
  - ◇ In principle, the order in which $\texttt{p}_1$, ..., $\texttt{p}_n$ are called does not matter, but, in practical systems it is fixed.
- If several clauses (definitions) $\texttt{p :- p}_1\texttt{, ..., p}_n$ means:
  $\texttt{p :- q}_1\texttt{, ..., q}_m$
  ...
  "to execute a call to $\texttt{p}$, call $\texttt{p}_1$ and ...and $\texttt{p}_n$, or, alternatively, $\texttt{q}_1$ and ...and $\texttt{q}_n$, or ..."
  - ◇ Unique to logic programming –it is like having several alternative procedure definitions.
  - ◇ Means that several possible paths may exist to a solution and they *should be explored*.
  - ◇ System usually stops when the first solution found, user can ask for more.
  - ◇ Again, in principle, the order in which these paths are explored does not matter (*if certain conditions are met*), but, for a given system, this is typically also fixed.

In the following we define a more precise operational semantics.

Unification: A fundamental Operation with Multiple Uses

- **Unification** (`=/2`) is an operation which finds the conditions needed for two terms to be equal:
  - ◇ E.g.: `f(a,g(X) = f(Y,g(b)` if `Y=a` and `X=b`.

  It has many uses:
  - ◇ It 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.
- It is a procedure to **solve equations on data structures**.
  - ◇ As usual, it returns a minimal solution to the equation (or the equation system).
  - ◇ As many equation solving procedures it is based on isolating variables and then *instantiating* them with their values.

Unification (more formally)

- **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** θ such that $\boxed{A\theta \equiv B\theta}$ (or, if not possible, *fail*).
  - Only variables can be given values!
  - Two structures can be made identical only by making their arguments identical.

E.g.:

A	$=$	B	θ	$A\theta$	$\equiv$	$B\theta$
f(X,g(t))		f(m(h),g(M))	X=m(h), M=t	f(m(h),g(t))		f(m(h),g(t))
dog		dog	$\emptyset$	dog		dog
dog		cat	*fail*			
X		a	X=a	a		a
X		Y	X=Y	Y		Y
f(X,g(t))		f(m(h),t(M))	*fail* (1)			
X		f(X)	*fail* (2)			

- (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**. (See, however, *cyclic terms* later.)

Unification

- Several solutions can exist, e.g.:

A	$=$ B	θ	$A\theta$ and $B\theta$
`f(X)`	`f(m(H))`	`X=m(a), H=a`	`f(m(a))`
”	”	`X=m(g(b)), H=g(b)`	`f(m(g(b)))`
...			

These are correct, but we want the solution that is *minimal* (that binds minimally the variables). In this case it is:

A	$=$ B	θ	$A\theta$ and $B\theta$
`f(X)`	`f(m(H))`	`X=m(H)`	`f(m(H))`

Note that the result, `f(m(H))`, is “more general” than `f(m(a))` or `f(m(g(b)))`.

- This minimal or *most general* solution always exists (unless unification fails), and is unique, modulo variable renaming.
- The *unification algorithm* finds this solution.

Unification Algorithm

Given a set of one or more equations: $A_1 = B_1$, $A_2 = B_2$, $\ldots$

- Initialize the solution θ to empty.
- Until no more equations left, do:
  - ◇ select an equation E,
  - ◇ *delete it*, and,

depending on the form equation E:	do:
`X=X`	ignore
`X=f(..., X, ...)`	fail (*occurs check*)
`X=`$term$	add X=$term$ to the solution and replace `X` by $term$ anywhere else
a=a	ignore
a=b	fail
`a=f(...)`	fail
`g(...)=f(...)`	fail
`f(...`m`...)=f(...`n`...)` ($m \neq n$)	fail
`f(`s_1`,...,`s_n`)=f(`t_1`,...,`t_n`)`	add to the system: s_1=t_1, ..., s_n=t_n

Unification Algorithm Examples

run example ⟼

- Unify: `p(X,f(b))` and `p(a,Y)`

`p(X,f(b))=p(a,Y)`	`X=a`
	`Y=f(b)`

- Unify: `p(X,f(Y))` and `p(a,g(b))`

`p(X,f(Y))=p(a,g(b))`	`X=a`	
	`f(Y)=g(b)`	*fail*

- Unify: `p(X,X)` and `p(f(Z),f(W))`

`p(X,X)=p(f(Z),f(W))`	`X=f(Z)`	`X=f(Z)`	`X=f(Z)`	`X=f(W)`
	`X=f(W)`	`f(Z)=f(W)`	`Z=W`	`Z=W`

- Unify: `p(X,f(Y))` and `p(Z,X)`

`p(X,f(Y))=p(Z,X)`	`X=Z`	`X=Z`	`X=f(Y)`
	`f(Y)=X`	`f(Y)=Z`	`Z=f(Y)`

- Unify: `p(X,f(X))` and `p(Z,Z)`

`p(X,f(X))=p(Z,Z)`	`X=Z`	`X=Z`	
	`f(X)=Z` }	`f(Z)=Z`	*fail* ("Occurs check")

A Schematic Interpreter for Logic Programs (SLD–resolution)

Input: A logic program P, a query Q

Output: μ (answer substitution) if Q is provable from P, *failure* otherwise

1. Make a copy Q' of Q
2. Initialize the "resolvent" R to be $\{Q\}$
3. While R is nonempty do:

   3.1. Take **a** literal A in R

   3.2. Take **a** clause A' :- $B_1, \ldots, B_n$ (*renamed*) from P with A' same predicate symbol as A

   3.2.1. If there is a solution θ to $A = A'$ (*unification*)
   - Replace A in R by $B_1, \ldots, B_n$
   - Apply θ to R and Q

   3.2.2. Otherwise, take **another** clause and repeat

   3.3. If there are no more clauses, go back to **some other choice**

   3.4. If there are no pending choices left, output *failure*
4. (R empty) Output solution μ to $Q = Q'$
5. Explore **another** pending branch for more solutions (upon request)

A Schematic Interpreter for Logic Programs (Standard Prolog)

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

1. Make a copy Q' of Q
2. Initialize the "resolvent" R to be $\{Q\}$
3. While R is nonempty do:
   - 3.1. Take **the leftmost** literal A in R
   - 3.2. Take **the first** clause A' :- $B_1, \ldots, B_n$ (*renamed*) from P with A' same predicate symbol as A
     - 3.2.1. If there is a solution θ to $A = A'$ (*unification*)
       - Replace A in R by $B_1, \ldots, B_n$
       - Apply θ to R and Q
     - 3.2.2. Otherwise, take **the next** clause and repeat
   - 3.3. If there are no more clauses, go back to **most recent pending choice**
   - 3.4. If there are no pending choices left, output *failure*
4. (R empty) Output solution μ to $Q = Q'$
5. Explore **the most recent** pending branch for more solutions (upon request)

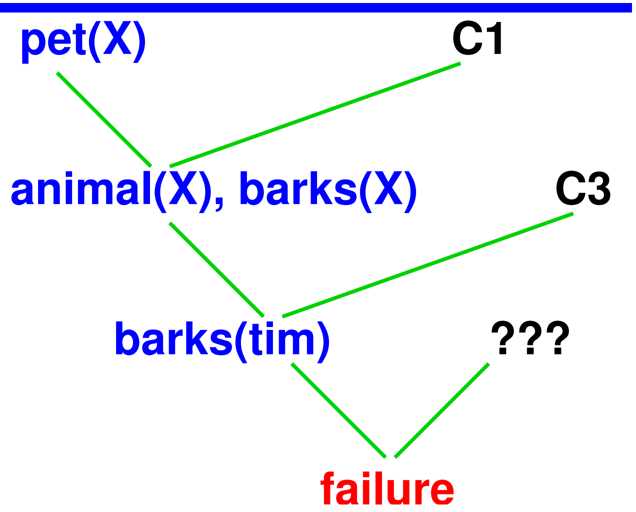
A Schematic Interpreter for Logic Programs (Contd.)

- Step 3.2 defines *alternative paths* to be explored to find answer(s); execution explores this tree (for example, breadth-first).
- Since step 3.2 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 3.2."
- Note that choosing a different clause (in step 3.2) can lead to finding solutions in a different order – Example (two valid executions):

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

- In fact, there is also some freedom in step 3.1, i.e., a system may also specify:
  - ◇ **Computation rule(s):** "how are literals selected in 3.1."

Running Programs: Alternative Execution Paths




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

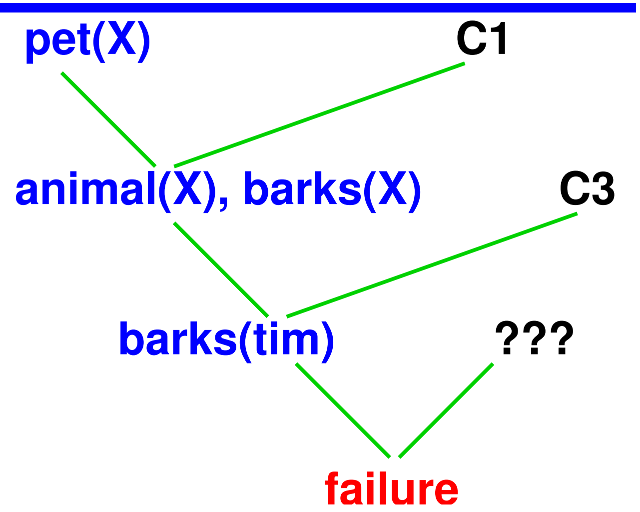
- `?- pet(X).` (top-down, left-to-right)

Q	R	**Clause**	θ
pet(X)	pet(X)	C_1*	{ $X=X_1$ }
pet(X_1)	animal(X_1), barks(X_1)	C_3*	{ X_1=tim }
pet(tim)	barks(tim)	???	*failure*

* means *choice-point*, i.e., other clauses applicable.

- But solutions exist in other paths!

Running Programs: Alternative Execution Paths




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

- `?- pet(X).` (top-down, left-to-right)

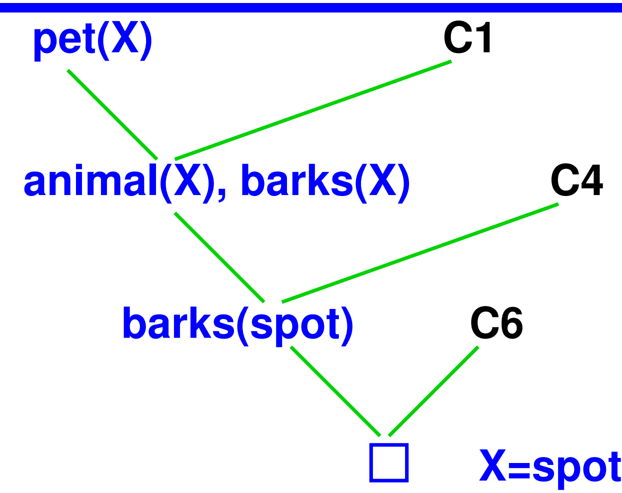
Q	R	**Clause**	θ
pet(X)	pet(X)	C_1*	{ X=X_1 }
pet(X_1)	animal(X_1), barks(X_1)	C_3*	{ X_1=tim }
pet(tim)	barks(tim)	???	*failure*

* means *choice-point*, i.e., other clauses applicable.

- But solutions exist in other paths!

→ Let's go back to our last choice point (C_3*) and try the next alternative...

Running Programs: Alternative Execution Paths




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

- `?- pet(X).` (top-down, left-to-right, different branch)

Q	R	**Clause**	θ
pet(X)	pet(X)	C_1*	{ X=X_1 }
pet(X_1)	animal(X_1), barks(X_1)	C_4*	{ X_1=spot }
pet(spot)	barks(spot)	C_6	{}
pet(spot)	—	—	—

- System response: `X = spot ?`
- 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_4*, or C_1*).

Comparison with Imperative and Functional Languages

- **Programs without search** (that do not perform “deep” backtracking):
  - Generally (if no disjunction etc. used) this means programs that:
    * Have only one clause per procedure, or
    * if several clauses, only one of them selected for every call to that predicate.

    Note that this is *dependent on call mode*, i.e., which variables are bound on a given call.
  - Because of the left-to-right rule, these programs *run in Prolog similarly to their imperative and (strict) functional counterparts.*
  - Imperative/functional programs can be directly expressed as such programs.
- **Programs *with search*** (perform “deep” backtracking):
  - These are programs that have at least one procedure that:
    * has multiple clauses, and
    * more than one of them is selected for some calls to that procedure.

    Again, this is *dependent on call mode*.
  - These programs *perform search* (backtracking-based, or other search rules).
  - They have no *direct* counterparts in imperative or functional programming.

Comparison with Imperative and Functional Languages (Contd.)

- Conventional languages and Prolog both implement *(forward) continuations*: the place to go after a procedure call *succeeds*. I.e., in:

```
p(X,Y):- q(X,Z), r(Z,Y).
q(X,Z) :- ...
```

  when the procedure call to `q/2` finishes (with "success"), execution continues in `p/2`, just after the call to `q/2`, i.e., at the call to `r/2` –the *forward continuation*.

- In Prolog, *when there are procedures with multiple definitions*, there is also a *backward continuation*: the place to go to if there is a *failure*. I.e., in:

```
p(X,Y):- q(X,Z), r(Z,Y).
p(X,Y):- ...
```

```
q(X,Z) :- ...
```

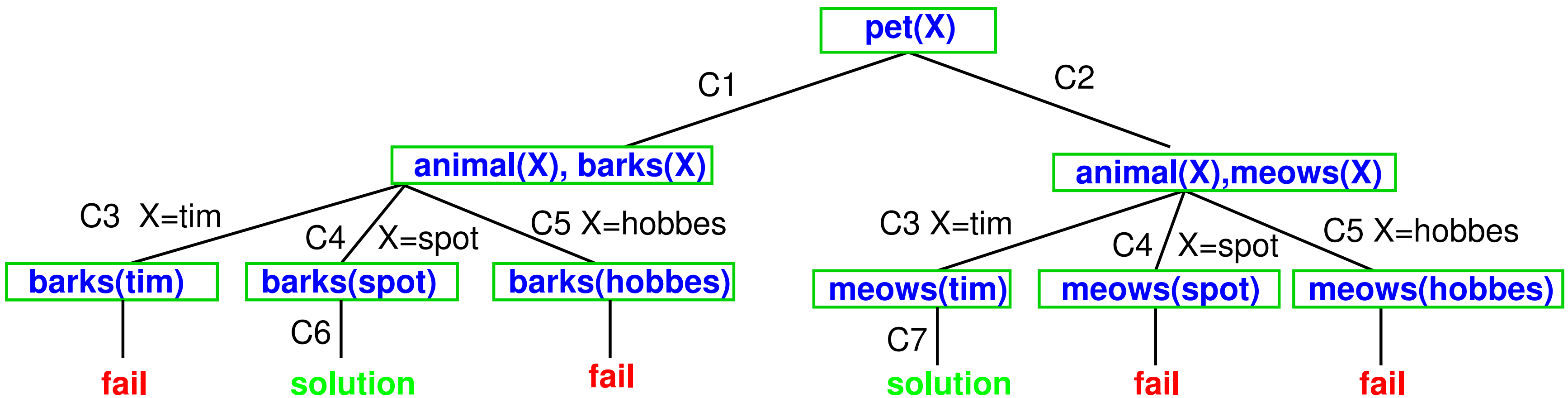
  if `q/2` succeeds, it is as above, but if it fails, execution continues at (backtracks to) the *previous alternative*: the second clause of `p/2` –the *backward continuation*x.

- We say that `p/2` has a **choice point**.

- Again, the debugger is very useful to observe how execution proceeds.

The Search Tree

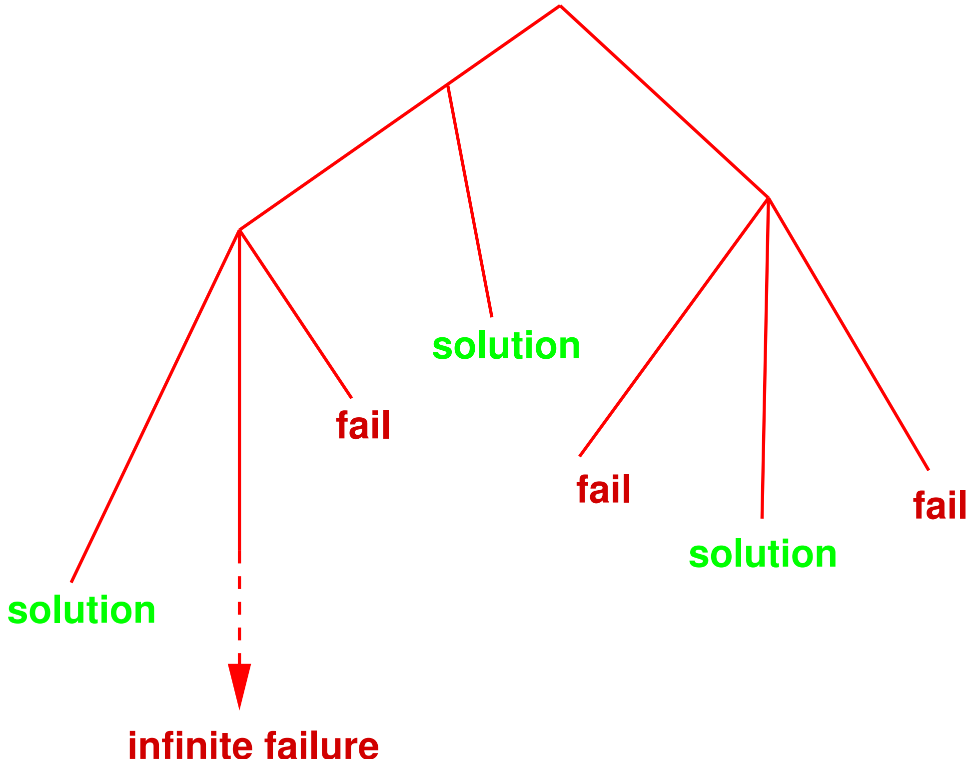
- A query + a logic program together determine a *search tree*.
  Example: previous program and `?- pet(X).`:
  (Boxes are the resolvents R; we skip variable renamings, i.e., $X = X_1$, for brevity.)




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

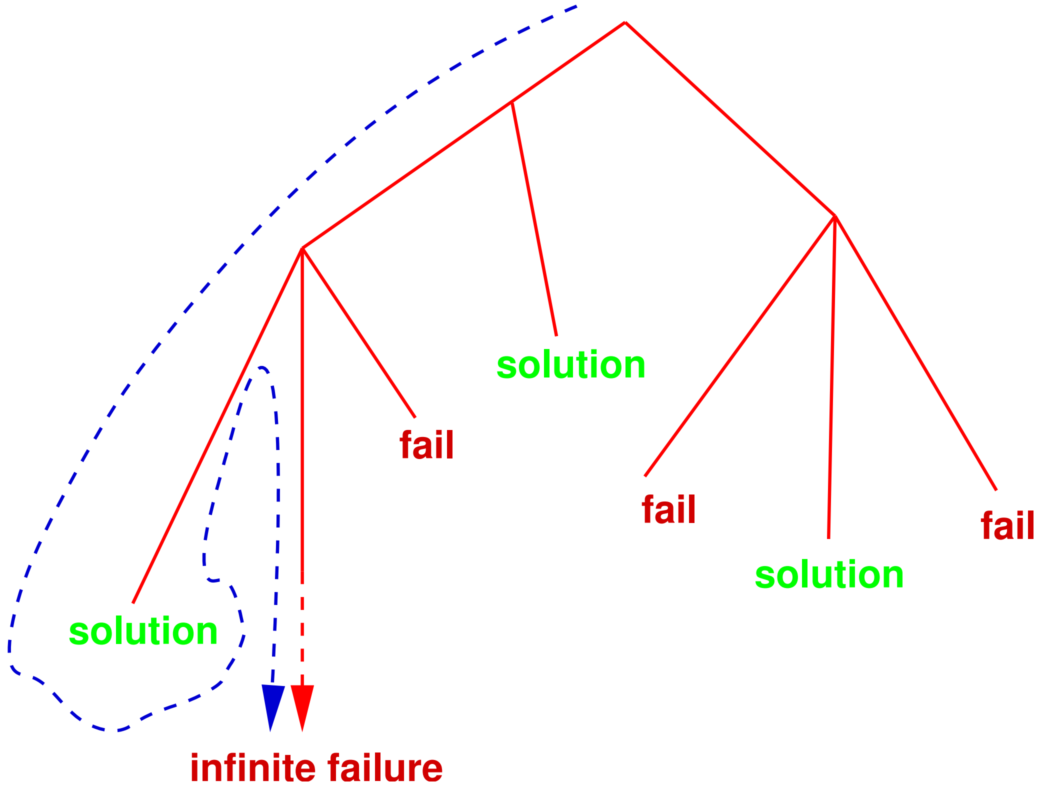
- Different execution strategies explore the tree in different ways
  (determined y the *search rule* and the *computation rule*).
- How can we achieve completeness (guarantee that all solutions will be found)?

Characterization of The Search Tree



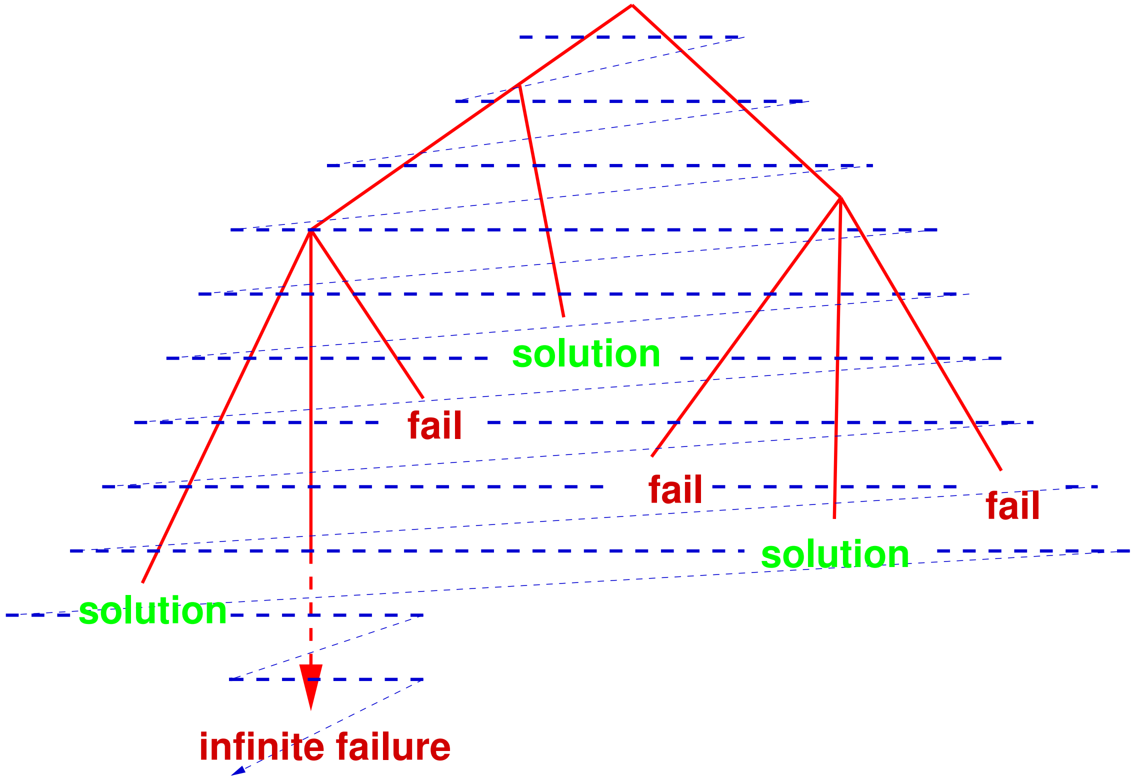
- All solutions are at *finite depth* in the tree.
- Failures can be at finite depth or, in some cases, be an infinite branch.

Depth-First Search (Backtracking)



- 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.
- It is the standard search rule in Prolog.

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 `bfall` package).

Selecting breadth-first or depth-first search

- In the Ciao system we can select the search rule using the *packages* mechanism.

- Files should start with the following line:

  ◇ To execute in *breadth-first* mode:

```
:- module(_,_,[sr/bfall]).
```

  ◇ To execute in *depth-first* mode:

```
:- module(_,_,[]).
```

See the part on Developing Programs with a Logic Programming System
for more details on the particular system used in the course (Ciao).

Control of Search in Depth-First Search (Backtracking)

Conventional programs (no search) execute conventionally.

Programs **with search**: programmer has at least three ways of *controlling search*:

1 The *ordering of literals* in the body of a clause:

- Profound effect on the *size of the computation* (at the limit, on termination).

  Compare executing `?- p(X), q(X,Y).` with executing `?- q(X,Y), p(X).` in:

```
p(4).          q(1, a) :- lots_of_computing...
p(5).          q(2, b) :- lots_of_computing...
               q(4, c) :- lots_of_computing...
               q(4, d) :- lots_of_computing...
```

run example ⟼

`p(X), q(X,Y)` is more efficient: execution of `p/2` *reduces the choices* of `q/2`.

- Note that optimal order depends on the instantiation of variables:
  E.g., for `q(X,d), p(X)`, this order is better than `p(X), q(X,d)`.

Control of Search in Depth-First Search (Backtracking) (Contd.)

2 The *ordering of clauses* in a predicate:

- Affects the *order* in which solutions are generated.

  E.g., in the previous example we get:
  `X=4,Y=c` as the first solution and `X=4,Y=d` as the second.

  If we reorder `q/2`:

```
p(4).          q(4, d) :- lots_of_computing...
p(5).          q(4, c) :- lots_of_computing...
               q(2, b) :- lots_of_computing...
               q(1, a) :- lots_of_computing...
```

run example ⟼

  we get `X=4,Y=d` first and then `X=4,Y=c`.

- If can also affect the *size* of the computation and *termination*.

3 The *pruning operators* (e.g., "cut"), which cut choices dynamically –see later.

Role of Unification in Execution

- As mentioned before, unification used to *access data* and *give values to variables*.

  Example: Consider query

  ```
  ?- animal(A), named(A,Name).
  ```

  with:

  ```
  animal(dog(tim)).
  ```

  ```
  named(dog(Name),Name).
  ```

  The call to `animal(A)` succeeds with `A=dog(tim)`.
  In order to access the name we call `named(A,Name)` which binds `Name=tim`.
  We could have also done simply: `?- animal(A), A=dog(Name).`

- Also, unification is used to *pass parameters* in procedure calls and to *return values* upon procedure exit. Here a value `spot` is returned in `P`:

Q	R	**Clause**	θ
pet(P)	pet(P)	C_1*	{ $P=X_1$ }
pet(X_1)	animal(X_1), barks(X_1)	C_3*	{ X_1=spot }
pet(spot)	barks(spot)	C_6	{}
pet(spot)	—	—	—

Answer: `P=spot`

“Modes”

- In fact, argument positions are not fixed a priory to be input or output.

  Example: Consider query `?- pet(spot).` vs. `?- pet(X).` run example ⟼

  or in the Peano arithmetic example from the introduction: run example ⟼

```
     ?- plus( s(0),  s(s(0)), Z).          % Adds
vs.  ?- plus( s(0),  Y,       s(s(s(0)))). % Subtracts
```

- Thus, procedures can be used in different **modes**
  s.t. different sets of arguments are input or output in each mode.

- We sometimes use `+` and `-` to refer to, respectively, and argument being an input or an output, e.g.:

  `plus(+X, +Y, -Z)` means we call `plus` with

  ◇ `X` instantiated, and

  ◇ `Y` instantiated,

  and we expect `Z` to be bound if `plus/3` succeeds.

Computational Logic

Pure Logic Programming Examples

(Non-Recursive)

Pure Logic Programs (Overview)

- Programs that only make use of unification
  (i.e., what we have described so far).
- They are fully "logical:"
  the set of computed answers "coincides" with the set of logical consequences.
  - *Computed answers*: the answers for all queries that terminate successfully.
- Allow programming declaratively:
  describe the problem, make queries, obtain correct answers
  $\rightarrow$ specifications as programs
- They have full computational power (Turing completeness).

(Recall the initial slides for the course.)

Database Programming

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

```
father_of(john,peter).
father_of(john,mary).
father_of(peter,michael).

mother_of(mary, david).
```

```
grandfather_of(L,M) :- father_of(L,N),
                       father_of(N,M).
grandfather_of(X,Y) :- father_of(X,Z),
                       mother_of(Z,Y).
```

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

```
?- father_of(john, peter).
yes
?- father_of(john, david).
no
?- father_of(john, X).
X = peter ;
X = mary
```

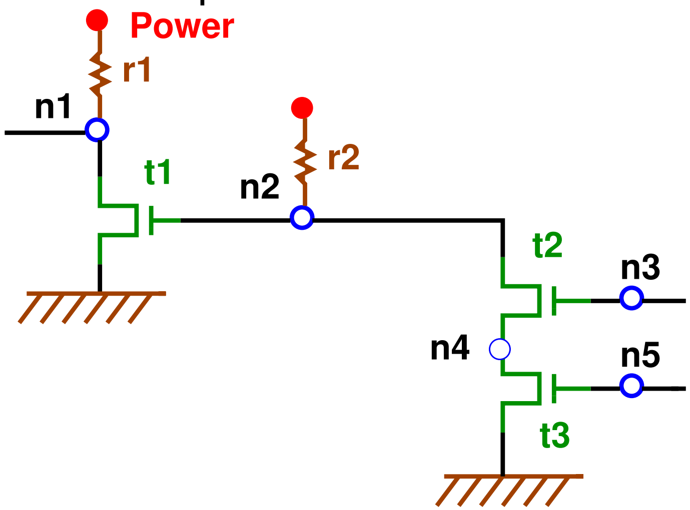
```
?- grandfather_of(X, michael).
X = john
?- grandfather_of(X, Y).
X = john, Y = michael ;
X = john, Y = david
?- grandfather_of(X, X).
no
```

- Try to write the rules for `grandmother_of(X,Y)`.
- Also for `parent/2`, `ancestor/2`, `related/2` (have a common ancestor).

Database Programming (Contd.)

- Another example:

run 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. Consider:

```
course(complog, wed, 18, 30, 20, 30, 'M.', 'Hermenegildo', new, 5102).
course(ai,      thu, 15, 00, 17, 00, 'J.', 'Smith',        old, 3102).
course(os,      wed, 18, 30, 20, 30, 'L.', 'Hubbard',      new, 6201).
 ...
```

- When is the Computational Logic course?

```
?- course(complog, Day, StartH, StartM, FinishH, FinishM, A, B, C, D).
```

- Structured version:

```
     course(complog, Time, Lecturer, Location) :-
         Time = t(wed,18:30,20:30),
         Lecturer = lect('M.','Hermenegildo'),
         Location = loc(new,5102).
```

Note: `X=Y` is equivalent to `=(X,Y)`
where predicate `=/2` is defined as the fact `=(X,X).` – i.e., unification.

- The `course` clause can also be written simply as:

```
course(complog,t(wed,18:30,20:30),lect('M.','Hermenegildo'),loc(new,5102)).
```

Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

```
course(complog,Time,Lecturer, Location) :-
        Time = t(wed,18:30,20:30),
        Lecturer = lect('M.','Hermenegildo'),
        Location = loc(new,5102).
```

- When is the Computational Logic course?

```
?- course(complog, Time, A, B).
```

  has solution:

```
Time=t(wed,18:30,20:30), A=lect('M.','Hermenegildo'), B=loc(new,5102)
```

- Using the *anonymous variable* ("_"):

```
?- course(complog,Time, _, _).
```

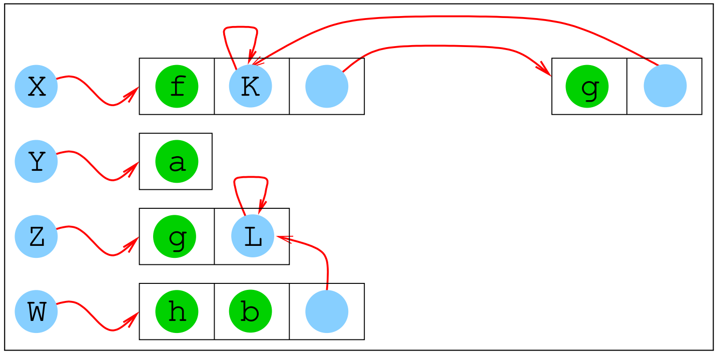
  has solution:

```
Time=t(wed,18:30,20:30)
```

Terms as Data Structures with Pointers

- `main` below is a procedure, that:
  - ◇ creates some data structures, with *pointers* and *aliasing*.
  - ◇ *calls* other *procedures*, *passing* to them *pointers* to these structures.

```
main :-
     X=f(K,g(K)),
     Y=a,
     Z=g(L),
     W=h(b,L),
% Heap memory at this point --->
     p(X,Y),
     q(Y,Z),
     r(W).
```



- Terms are data structures with pointers.
- Logical variables are *declarative* pointers.
  - ◇ Declarative: they can only be assigned once.

Structured Data and Data Abstraction (Contd.)

- The circuit example revisited: run example ⟼

```
resistor(r1,power,n1).        transistor(t1,n2,ground,n1).
resistor(r2,power,n2).        transistor(t2,n3,n4,n2).
                              transistor(t3,n5,ground,n4).
inverter(inv(T,R),Input,Output) :-
  transistor(T,Input,ground,Output),
  resistor(R,power,Output).

nand_gate(nand(T1,T2,R),Input1,Input2,Output) :-
  transistor(T1,Input1,X,Output),
  transistor(T2,Input2,ground,X),
  resistor(R,power,Output).

and_gate(and(N,I),Input1,Input2,Output) :-
   nand_gate(N,Input1,Input2,X), inverter(I,X,Output).
```

- The query `?- and_gate(G,In1,In2,Out).`

  has solution: `G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1`

Logic Programs and the Relational DB Model

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

"Person"

Name	**Age**	**Sex**
Brown	20	M
Jones	21	F
Smith	36	M

```
person(brown,20,male).
person(jones,21,female).
person(smith,36,male).
```

"Lived in"

Name	**Town**	**Years**
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

```
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 argnames package can be used to give names to arguments:

```
:- use_package(argnames).
:- argnames person(name,age,sex).
:- argnames lived_in(name,town,years).
```

run example ⟼

Logic Programs and the Relational DB Model (Contd.)

- The operations of the relational model are easily implemented as rules.
  - *Union*: $\texttt{r\_union\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n).$
    $\texttt{r\_union\_s}(X_1,\ldots,X_n) \leftarrow \texttt{s}(X_1,\ldots,X_n).$
  - *Cartesian Product*:
    $\texttt{r\_X\_s}(X_1,\ldots,X_m,X_{m+1},\ldots,X_{m+n}) \leftarrow \texttt{r}(X_1,\ldots,X_m),\texttt{s}(X_{m+1},\ldots,X_{m+n}).$
  - *Projection*: $\texttt{r13}(X_1,X_3) \leftarrow \texttt{r}(X_1,X_2,X_3).$
  - *Selection*: $\texttt{r\_selected}(X_1,X_2,X_3) \leftarrow \texttt{r}(X_1,X_2,X_3),\leq(X_2,X_3).$
    ($\leq/2$ can be, e.g., Peano, Prolog built-in, constraints...)
  - *Set Difference*: $\texttt{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n),\ \textit{not}\ \texttt{s}(X_1,\ldots,X_n).$
    $\texttt{r\_diff\_s}(X_1,\ldots,X_n) \leftarrow \texttt{s}(X_1,\ldots,X_n),\ \textit{not}\ \texttt{r}(X_1,\ldots,X_n).$
    (we postpone the discussion on *negation* until later.)
- Derived operations – some can be expressed more directly in LP:
  - Intersection: $\texttt{r\_meet\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,\ldots,X_n),\ \texttt{s}(X_1,\ldots,X_n).$
  - Join: $\texttt{r\_joinX2\_s}(X_1,\ldots,X_n) \leftarrow \texttt{r}(X_1,X_2,X_3,\ldots,X_n),\ \texttt{s}(X_1',X_2,X_3',\ldots,X_n').$
- 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.
  - ◇ Powerful notions of negation supported: S-models
    $\rightarrow$ **Answer Set Programming** (ASP)
    $\rightarrow$ powerful knowledge representation and reasoning systems.

Computational Logic

Pure Logic Programming Examples

(Recursion, Data Types)

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).
...
```

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).
```

run example ⟼

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).
```

run example ⟼

- Exercise (from previous part): define “related”, “cousin”, “same generation”, etc.

Types

- *Type:* a (possibly infinite) set of terms.

Types

- *Type:* a (possibly infinite) set of terms.
- *Type definition:* A program defining a type.

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:

```
weekday('Monday').
weekday('Tuesday').    ...
```

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:

```
weekday('Monday').
weekday('Tuesday').    ...
```

- *Example:* Date (weekday * day in the month):
  - ◇ Set of terms to represent: `date('Monday',23)`, `date('Tuesday',24)`, ...
  - ◇ Type definition: run example ⟼

```
is_date(date(W,D)) :- weekday(W), day_of_month(D).
day_of_month(1).
day_of_month(2).
...
day_of_month(31).
```

Recursive Programming: Recursive Types

- *Recursive types*: defined by recursive logic programs.

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).
- Types are *runnable* and can be used to check or produce values:
  - ◇ `?- nat(X)` ⇒ `X=0; X=s(0); X=s(s(0));` ...
- 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), s(s(0)), ...`
  - ◇ Type definition:

```
integer( X) :- nat(X).
integer(-X) :- nat(X).
```

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).
- Types are *runnable* and can be used to check or produce values:
  - ◇ `?- nat(X)` ⇒ `X=0; X=s(0); X=s(s(0)); ...`
- 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), s(s(0)), ...`
  - ◇ Type definition:

```
integer( X) :- nat(X).
integer(-X) :- nat(X).
```

  - ◇ 'Duplication' (of 0) is not a problem. But, can we eliminate it?

Recursive Programming: Arithmetic

- Defining the natural order ($\leq$) of natural numbers: run example $\longmapsto$

```
less_or_equal(0,X) :- nat(X).
less_or_equal(s(X),s(Y)) :- less_or_equal(X,Y).
```

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

Recursive Programming: Arithmetic

- Defining the natural order ($\leq$) of natural numbers: run example $\longmapsto$

```
less_or_equal(0,X) :- nat(X).
less_or_equal(s(X),s(Y)) :- less_or_equal(X,Y).
```

  ◇ Multiple uses (modes):
  `less_or_equal(s(0),s(s(0))), less_or_equal(X,0),` ...

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

- Addition:

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

  ◇ Multiple uses (modes): `plus(s(s(0)),s(0),Z), plus(s(s(0)),Y,s(0))`

  ◇ Multiple solutions: `plus(X,Y,s(s(s(0))))`, etc.

Recursive Programming: Arithmetic

- Another possible definition of addition:

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

- The meaning of `plus` is the same, even if both definitions are combined.

Recursive Programming: Arithmetic

- Another possible definition of addition:

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

- The meaning of `plus` is the same, even if both definitions are combined.
- Not recommended: several proof trees for the same query $\rightarrow$ not efficient, not concise. We look for minimal axiomatizations.

Recursive Programming: Arithmetic

- Another possible definition of addition:

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

- The meaning of `plus` is the same, even if both definitions are combined.
- Not recommended: several proof trees for the same query $\rightarrow$ not efficient, not concise. We look for minimal axiomatizations.
- The art of logic programming: finding compact and computationally efficient formulations!

Recursive Programming: Arithmetic

- Another possible definition of addition:

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

- The meaning of `plus` is the same, even 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: `times(X,Y,Z)` (Z = X*Y), `exp(N,X,Y)` (Y = X^N), `factorial(N,F)` (F = N!), `minimum(N1,N2,Min)`, ...

Recursive Programming: Arithmetic

- Definition of `mod(X,Y,Z)`
  "`Z` is the remainder from dividing `X` by `Y`"

  $\exists Q s.t.\ X = Y * Q + Z \wedge Z < Y$

  $\Rightarrow$

```
mod(X,Y,Z) :- less(Z, Y), times(Y,Q,W), plus(W,Z,X).

less(0,s(X)) :- nat(X).
less(s(X),s(Y)) :- less(X,Y).
```

run example $\longmapsto$

Recursive Programming: Arithmetic

- Definition of `mod(X,Y,Z)`
  "Z is the remainder from dividing X by Y"
  $\exists Q s.t.\ X = Y * Q + Z \land Z < Y$
  $\Rightarrow$

```
mod(X,Y,Z) :- less(Z, Y), times(Y,Q,W), plus(W,Z,X).

less(0,s(X)) :- nat(X).
less(s(X),s(Y)) :- less(X,Y).
```

run example $\longmapsto$

- Another possible definition:

```
mod(X,Y,X) :- less(X, Y).
mod(X,Y,Z) :- plus(X1,Y,X), mod(X1,Y,Z).
```

Recursive Programming: Arithmetic

- Definition of `mod(X,Y,Z)`
  "Z is the remainder from dividing X by Y"
  $\exists Q s.t.\ X = Y * Q + Z \wedge Z < Y$
  $\Rightarrow$

```
mod(X,Y,Z) :- less(Z, Y), times(Y,Q,W), plus(W,Z,X).

less(0,s(X)) :- nat(X).
less(s(X),s(Y)) :- less(X,Y).
```

run example $\longmapsto$

- Another possible definition:

```
mod(X,Y,X) :- less(X, Y).
mod(X,Y,Z) :- plus(X1,Y,X), mod(X1,Y,Z).
```

- Much more efficient than the previous 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))
```

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:

```
ackermann(0,N)          = s(N)
ackermann(s(M1),0)      = ackermann(M1,s(0))
ackermann(s(M1),s(N1)) = ackermann(M1,ackermann(s(M1),N1))
```

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:

```
ackermann(0,N)             = s(N)
ackermann(s(M1),0)         = ackermann(M1,s(0))
ackermann(s(M1),s(N1))     = ackermann(M1,ackermann(s(M1),N1))
```

- Can be defined by a logic programming as follows: run example ⟼

```
ackermann(0,N,s(N)).
ackermann(s(M1),0,Val)        :-  ackermann(M1,s(0),Val).
ackermann(s(M1),s(N1),Val)    :-  ackermann(s(M1),N1,Val1),
                                  ackermann(M1,Val1,Val).
```

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:

```
ackermann(0,N)            = s(N)
ackermann(s(M1),0)        = ackermann(M1,s(0))
ackermann(s(M1),s(N1)) = ackermann(M1,ackermann(s(M1),N1))
```

- Can be defined by a logic programming as follows: run example ⟼

```
ackermann(0,N,s(N)).
ackermann(s(M1),0,Val)       :-  ackermann(M1,s(0),Val).
ackermann(s(M1),s(N1),Val) :-  ackermann(s(M1),N1,Val1),
                                 ackermann(M1,Val1,Val).
```

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

Functional Syntax: Packages and Directives (I)

- `:- use_package(fsyntax).` Provides:
  - `~` "*eval*", which makes the last argument implicit. This allows writing, e.g.

```
p(X,Y) :- q(X,Z), r(Z,Y).
```

as

```
p(X,Y) :- r(~q(X),Y).
```

or

```
p(X,~r(~q(X))).
```

Functional Syntax: Packages and Directives (I)

- `:- use_package(fsyntax).` Provides:
  - `~` "*eval*", which makes the last argument implicit. This allows writing, e.g.

```
p(X,Y) :- q(X,Z), r(Z,Y).
```

    as

```
p(X,Y) :- r(~q(X),Y).
```

    or

```
p(X,~r(~q(X))).
```

  - `:=` for definitions: which allows writing, e.g.

```
p(X,Y) :- q(X,Z), r(Z,Y).
```

    as

```
p(X) := Y :- r(~q(X),Y).
```

    or

```
p(X) := ~r(~q(X)).
```

Functional Syntax: Packages and Directives (I)

- `:- use_package(fsyntax).` Provides:
  - `~` "*eval*", which makes the last argument implicit. This allows writing, e.g.

    ```
    p(X,Y) :- q(X,Z), r(Z,Y).
    ```

    as

    ```
    p(X,Y) :- r(~q(X),Y).
    ```

    or

    ```
    p(X,~r(~q(X))).
    ```

  - `:=` for definitions: which allows writing, e.g.

    ```
    p(X,Y) :- q(X,Z), r(Z,Y).
    ```

    as

    ```
    p(X) := Y :- r(~q(X),Y).
    ```

    or

    ```
    p(X) := ~r(~q(X)).
    ```

  - `|` for *or*

  - etc.

Functional Syntax: Packages and Directives (II)

- Thus, for example, this clause:

```
ackermann(s(M),s(N),Val) :-
          ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

  can be rewritten as:

```
ackermann(s M, s N) := ~ackermann(M, ~ackermann(s M, N) ).
```

Functional Syntax: Packages and Directives (II)

- Thus, for example, this clause:

```
ackermann(s(M),s(N),Val) :-
          ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

  can be rewritten as:

```
ackermann(s M, s N) := ~ackermann(M, ~ackermann(s M, N) ).
```

- To evaluate automatically functors that are defined as functions (so there is no need to use `~` for them):

```
:- fun_eval ackermann/2.
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

Functional Syntax: Packages and Directives (II)

- Thus, for example, this clause:

```
ackermann(s(M),s(N),Val) :-
         ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

  can be rewritten as:

```
ackermann(s M, s N) := ~ackermann(M, ~ackermann(s M, N) ).
```

- To evaluate automatically functors that are defined as functions (so there is no need to use `~` for them):

```
:- fun_eval ackermann/2.
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

- To enable this for *all* functions defined in a given file:

```
:- fun_eval defined(true).
```

Functional Syntax: Packages and Directives (II)

- Thus, for example, this clause:

```
ackermann(s(M),s(N),Val) :-
         ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

can be rewritten as:

```
ackermann(s M, s N) := ~ackermann(M, ~ackermann(s M, N) ).
```

- To evaluate automatically functors that are defined as functions (so there is no need to use `~` for them):

```
:- fun_eval ackermann/2.
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

- To enable this for *all* functions defined in a given file:

```
:- fun_eval defined(true).
```

- To evaluate arithmetic functors automatically (no need for `~` for them):

```
:- fun_eval arith(true).
add_one(X,X+1).
```

Functional Syntax: Packages and Directives (II)

- Thus, for example, this clause:

```
ackermann(s(M),s(N),Val) :-
         ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

can be rewritten as:

```
ackermann(s M, s N) := ~ackermann(M, ~ackermann(s M, N) ).
```

- To evaluate automatically functors that are defined as functions (so there is no need to use `~` for them):

```
:- fun_eval ackermann/2.
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

- To enable this for *all* functions defined in a given file:

```
:- fun_eval defined(true).
```

- To evaluate arithmetic functors automatically (no need for `~` for them):

```
:- fun_eval arith(true).
add_one(X,X+1).
```

- The `functional` package includes `fsyntax` + both `fun_eval`'s above:

```
:- use_package(functional).
```

Recursive Programming: Arithmetic/Functions (Functional Syntax)

- The Ackermann function (Peano) in Ciao's functional Syntax and defining s as a prefix operator:

run example ⟼

```
:- use_package(functional).
:- op(500,fy,s).

ackermann(  0,   N) := s N.
ackermann(s M,   0) := ackermann(M, s 0).
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

Recursive Programming: Arithmetic/Functions (Functional Syntax)

- The Ackermann function (Peano) in Ciao's functional Syntax and defining s as a prefix operator: run example ⟼

```
:- use_package(functional).
:- op(500,fy,s).

ackermann(  0,    N) := s N.
ackermann(s M,    0) := ackermann(M, s 0).
ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).
```

- Functional syntax is convenient also e.g. for defining types:

```
nat(0).
nat(s(X)) :- nat(X).
```

Using special := notation for the "return" (last) argument:

```
nat := 0.
nat := s(X) :- nat(X).
```

Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the ˜ notation (“evaluate and replace with result”):

```
nat := 0.
nat := s(˜nat).
```

Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the ~ notation ("evaluate and replace with result"):

```
nat := 0.
nat := s(~nat).
```

"~" not needed with `functional` package if inside its own definition:

```
nat := 0.
nat := s(nat).
```

Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the ~ notation ("evaluate and replace with result"):

```
nat := 0.
nat := s(~nat).
```

"~" not needed with `functional` package if inside its own definition:

```
nat := 0.
nat := s(nat).
```

Using an `:- op(500,fy,s).` declaration to define `s` as a *prefix operator*:

```
nat := 0.
nat := s nat.
```

Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the ~ notation ("evaluate and replace with result"):

```
nat := 0.
nat := s(~nat).
```

"~" not needed with `functional` package if inside its own definition:

```
nat := 0.
nat := s(nat).
```

Using an `:- op(500,fy,s).` declaration to define `s` as a *prefix operator*:

```
nat := 0.
nat := s nat.
```

Using "|" (disjunction):

```
nat := 0 | s nat.
```

Recursive Programming: Arithmetic/Functions (Funct. Syntax, Contd.)

Moving body call to head using the ~ notation (“evaluate and replace with result”):

```
nat := 0.
nat := s(~nat).
```

“~” not needed with `functional` package if inside its own definition:

```
nat := 0.
nat := s(nat).
```

Using an `:- op(500,fy,s).` declaration to define `s` as a *prefix operator*:

```
nat := 0.
nat := s nat.
```

Using “|” (disjunction):

```
nat := 0 | s nat.
```

Which is exactly equivalent to:

```
nat(0).
nat(s(X)) :- nat(X).
```

Recursive Programming: Lists

- Binary structure: first argument is *element*, second argument is *rest* of the list.
- We need:
  - ◇ A constant symbol: we use the *constant* [ ] ($\rightarrow$ denotes the empty list).
  - ◇ A functor of arity 2: traditionally the dot "." (which is overloaded).

Recursive Programming: Lists

- Binary structure: first argument is *element*, second argument is *rest* of the list.
- We need:
  - ◇ A constant symbol: we use the *constant* [ ] (→ denotes the empty list).
  - ◇ 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*).

Formal object	"Cons pair" syntax	"Element" syntax
`.(a,[])`	`[a|[]]`	`[a]`
`.(a,.(b,[]))`	`[a|[b|[]]]`	`[a,b]`
`.(a,.(b,.(c,[])))`	`[a|[b|[c|[]]]]`	`[a,b,c]`
`.(a,X)`	`[a|X]`	`[a|X]`
`.(a,.(b,X))`	`[a|[b|X]]`	`[a,b|X]`

- Note that:
  `[a,b]` and `[a|X]` unify with {`X = [b]`}
  `[a]` and `[a,b|X]` do not unify
  `[a]` and `[a|X]` unify with {`X = []`}
  `[]` and `[X]` do not unify

Recursive Programming: Lists (Contd.)

- Type definition (no syntactic sugar): run example ⟼

```
list([]).
list(.(X,Y)) :- list(Y).
```

- Type definition, with some syntactic sugar (`[ ]` notation):

```
list([]).
list([X|Y]) :- list(Y).
```

- Type definition, using also `functional` package:

```
list := [] | [_|list].
```

- "Exploring" the type:

```
?- list(L).
L = [] ? ;
L = [_] ? ;
L = [_,_] ? ;
L = [_,_,_] ?
...
```

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,[X|Y]) :- list(Y).**

```
member(a,[c,a]),    member(b,[d,b]).    etc. ⇒ member(X,[Y,X]).
member(a,[c,d,a]).  member(b,[s,t,b]).  etc. ⇒ member(X,[Y,Z,X]).
```

⇒ **member(X,[Y|Z]) :- member(X,Z).**

- Resulting definition: run example ⟼

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

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

Recursive Programming: Lists (Contd.)

- Combining lists and naturals: run example ⟼
  - Computing the length of a list:

```
len([],0).
len([H|T],s(LT)) :- len(T,LT)
```

  - Adding all elements of a list:

```
sumlist([],0).
sumlist([H|T],S) :- sumlist(T,ST), plus(H,ST,S).
```

  - The type of lists of natural numbers:

```
natlist([]).
natlist([H|T]) :- nat(H), natlist(T).
```

    or:

```
natlist := [] | [~nat|natlist].
```

Recursive Programming: One Way to Understand It

- One way to visualize recursion is as creating 'copies' of procedures:

```
?- sumlist([s(0), s(s(0)), s(0)], R).
        \
         sumlist([],0).       % fails
         sumlist([H|T],S) :-  % H=s(0), T=[s(s(0)), s(0)]
             sumlist(T,ST),   % T=[s(s(0)), s(0)]
             |      \
             |       sumlist([],0).       % fails
             |       sumlist([H1|T1],S1) :-  % H1=s(s(0), T1=[s(0)]
             |          sumlist(T1,ST1),     % T1=[s(0)]
             |          |       \
             |          |        sumlist([],0).          % fails
             |          |        sumlist([H2|T2],S2) :-  % H2=s(0), T2=[]
             |          |           sumlist(T2,ST2),     % T2=[]
             |          |           |      \
             |          |           |       sumlist([],0).      % ST2=0
             |          |           |      /
             |          |           % H2=s(0), ST2=0
             |          |           plus(H2,ST2,S2). % S2=s(0)
             |          |          /
             |          % H1=s(s(0)), ST1=s(0)
             |          plus(H1,ST1,S1). % S1=s(s(s(0)))
             |        /
             % H=s(0), ST=s(s(s(0)
             plus(H,ST,S). % S=s(s(s(s(0))))
           /
           % R=s(s(s(s(0))))
```

Recursive Programming: Lists (Contd.)

- Exercises:
  - ◇ 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)`, ...

Recursive Programming: Lists (Contd.)

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

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

Recursive Programming: Lists (Contd.)

- Second generalization:

```
append([X],Ys,[X|Ys]) :- list(Ys).
append([X,Z],Ys,[X,Z|Ys]) :- list(Ys).
append([X,Z,W],Ys,[X,Z,W|Ys]) :- list(Ys).
```

$\Rightarrow$ **append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).**

- So, we have: run example $\longmapsto$

```
append([],Ys,Ys) :- list(Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

- Another way of reasoning: thinking inductively.
  - ◇ The base case is: `append([],Ys,Ys):-list(Ys).`
  - ◇ If we assume that `append(Xs,Ys,Zs)` works for some iteration, then, in the next one, the following should hold: `append([X|Xs],Ys,[X|Zs])`.

Recursive Programming: Lists (Contd.)

- Uses of append:
  - Concatenate two given lists:

```
?- append([a,b,c],[d,e],L).
L = [a,b,c,d,e] ?
```

  - Find differences between lists:

```
?- append(D,[d,e],[a,b,c,d,e]).
D = [a,b,c] ?
```

  - Split a list:

```
?- append(A,B,[a,b,c,d,e]).
A = [],
B = [a,b,c,d,e] ? ;
A = [a],
B = [b,c,d,e] ? ;
A = [a,b],
B = [c,d,e] ? ;
A = [a,b,c],
B = [d,e] ?
...
```

Recursive Programming: Lists (Contd.)

- `reverse(Xs,Ys)`: Ys is the list obtained by reversing the elements in the list Xs

  Each element X of `[X|Xs]` should end up at the end of the reversed version of Xs:

```
reverse([X|Xs],Ys) :-
   reverse(Xs,Zs),
   append(Zs,[X],Ys).
```

  Inductively: if we assume Xs is already reversed as Zs, if Xs has one more element at the beginning, it goes at the end of Zs.

  How can we stop (i.e., what is our base case): run example ⟼

```
reverse([],[]).
```

- As defined, reverse(Xs,Ys) is very inefficient. Another possible definition: (uses an *accumulating parameter*)

```
reverse(Xs,Ys) :- reverse(Xs,[],Ys).

reverse([],Ys,Ys).
reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
```

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

Recursive Programming: Binary Trees

- Represented by a ternary functor `tree(Element,Left,Right)`.
- Empty tree represented by `void`.
- Definition: run example ⟼

```
binary_tree(void).
binary_tree(tree(_Element,Left,Right)) :-
    binary_tree(Left),
    binary_tree(Right).
```

- Defining `tree_member(Element,Tree)`:

```
tree_member(X,tree(X,Left,Right)) :-
    binary_tree(Left),
    binary_tree(Right).
tree_member(X,tree(_,Left,Right)) :- tree_member(X,Left).
tree_member(X,tree(_,Left,Right)) :- tree_member(X,Right).
```

Recursive Programming: Binary Trees

- Defining `pre_order(Tree,Elements)`:
  `Elements` is a list containing the elements of `Tree` traversed in *preorder*.

```
pre_order(void,[]).
pre_order(tree(X,Left,Right),Elements) :-
      pre_order(Left,ElementsLeft),
      pre_order(Right,ElementsRight),
      append([X|ElementsLeft],ElementsRight,Elements).
```

run example ⟼

- Exercise – define:
  - ◇ `in_order(Tree,Elements)`
  - ◇ `post_order(Tree,Elements)`

Polymorphism

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

run example ⟼

```
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!

- `:- 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 (somewat surprising): `:- lt_member(M,T).`

Recursive Programming: Manipulating Symbolic Expressions

- Recognizing (and generating!) 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

run example ⟼

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

Recursive Programming: Manipulating Symb. Expressions (Contd.)

- Symbolic differentiation: `deriv(Expression, X, Derivative)` run example ⟼

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

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

- A simplification step can be added.

Recursive Programming: Graphs

- A common approach: make use of another data structure, e.g., lists:
  - Graphs as lists of edges.
- Alternative: make use of Prolog's program database:
  - Declare the graph using facts in the program.

```
edge(a,b).      edge(c,a).
edge(b,c).      edge(d,a).
```

- Paths in a graph: `path(X,Y)` iff there is a path in the graph from node `X` to node `Y`.

```
path(A,B) :- edge(A,B).
path(A,B) :- edge(A,X), path(X,B).
```

- Circuit: a closed path. `circuit` iff there is a path in the graph from a node to itself.

```
circuit :- path(A,A).
```

Recursive Programming: Graphs (Exercises)

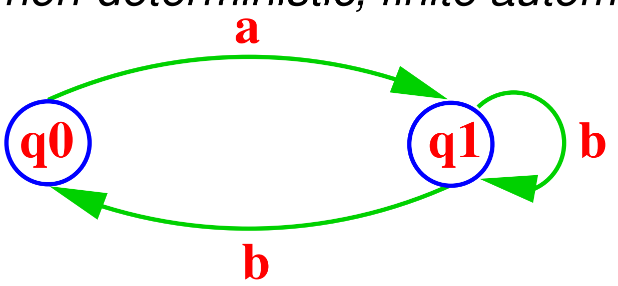
- Modify `circuit`/0 so that it provides the circuit. (You have to modify also `path`/2)
- Propose a solution for handling several graphs in our representation.
- Propose a suitable representation of graphs as data structures.
- Define the previous predicates for your representation.

- Consider unconnected graphs (there is a subset of nodes not connected in any way to the rest) versus connected graphs.
- Consider directed versus undirected graphs.

- Try `path(a,d)`. Solve the problem.

Recursive Programming: Automata (Graphs)

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



where **q0** is both the *initial* and the *final* state.

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

run example ⟼

```
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).
```

Recursive Programming: Automata (Graphs) (Contd.)

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

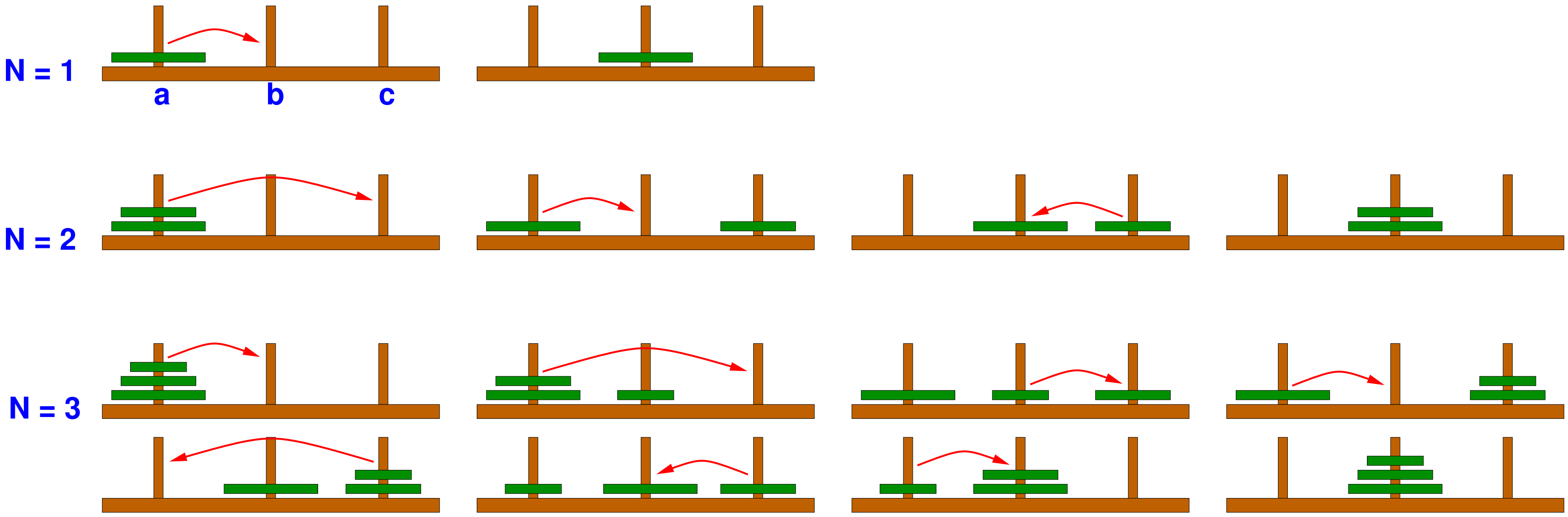
run example ⟼

```
accept(S) :- initial(Q), accept_from(S,Q,[]).

accept_from([],Q,[])     :- final(Q).
accept_from([X|Xs],Q,S) :- delta(Q,X,S,NewQ,NewS),
                           accept_from(Xs,NewQ,NewS).

initial(q0).
final(q1).

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

- What sequence does it recognize?

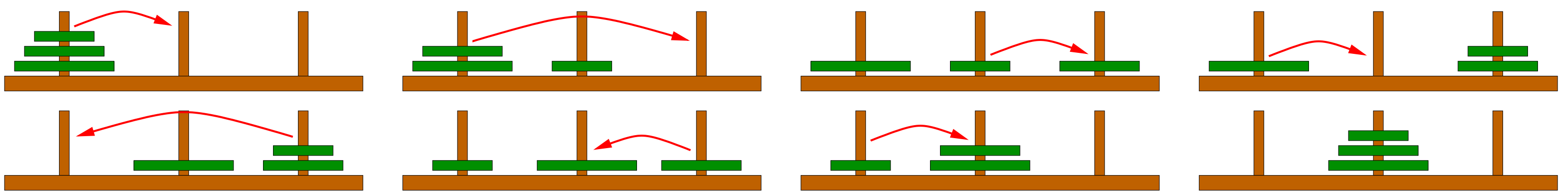
Recursive Programming: Towers of Hanoi

- *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.



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:* The moves for three disks

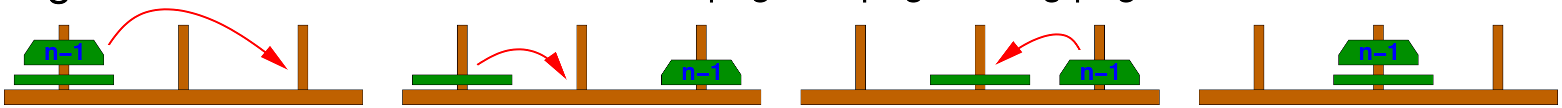


are represented by:

```
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: To move N disks from peg A to peg B using peg C we need to:



move N-1 disks to peg C using peg B, move the bottom disk to peg B, and then move the N-1 disks from peg C to peg B using peg A.

- 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`."

```
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:

```
hanoi_moves(N,Moves) :-
        hanoi(N,a,b,c,Moves).
```

run example ⟼

Learning to Compose Recursive Programs

- To some extent it is a simple question of practice.
- By generalization (as in the previous examples): elegant, but sometimes difficult? (Not the way most people do it.)
- Think inductively: 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 then also if alternative uses make sense.
- Sometimes it helps to look at well-written examples and use the same "schemas."
- Using a global top-down design approach can help (in general, not just for recursive programs):
  - State the general problem.
  - Break it down into subproblems.
  - Solve the pieces.
- Again, the best approach: practice, practice, practice.