Computational Logic

Introduction to Logic Programming

Overview

1. Syntax: data
2. Manipulating data: Unification
3. Syntax: code
4. Semantics: meaning of programs
5. Executing logic programs

Syntax: Terms (Variables, Constants, and Structures)

- **Variables:** start with uppercase character (or "_"), may include "_" and digits:

  Examples: `X, Im4u, A_little_garden, _, _x, _22`

- **Constructor:** (or **functor**) lowercase first character, may include "_" and digits. Also, some special characters. Quoted, any character:

  Examples: `a, dog, a_big_cat, x22, 'Hungry man', [], *, >`
  `'Doesn''t matter'`

- **Structures:** a constructor (the structure name) followed by a fixed number of arguments between parentheses:

  Example: `date(monday, Month, 1994)`

  Arguments can in turn be variables, constants and structures.

- **Constants:** structures without arguments (only name) and also numbers (with the usual decimal, float, and sign notations).

  - ◇ Numbers: `0`, `999`, `-77`, `5.23`, `0.23e-5`, `0.23E-5`.

Syntax: Terms

- **Arity:** is the number of arguments of a structure. Constructors are represented as *name/arity* (e.g., `date/3`).
  - ◇ 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.

- *Examples*:

Term	*Type*	*Constructor*
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 does not contain free variables.

Manipulating Data Structures (Unification)

- **Unification** is the only mechanism available in logic programs for manipulating data structures. It is used to:
  - ◇ Pass parameters.
  - ◇ Return values.
  - ◇ Access parts of structures.
  - ◇ Give values to variables.
- Unification 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 substituting them by their values.

Unification

- **Unifying two terms A and B:** is asking if they can be made syntactically identical by giving (minimal) values to their variables.
  - I.e., find a solution θ to equation $\boxed{A = B}$ (or, if impossible, *fail*).
  - Only variables can be given values!
  - Two structures can be made identical only by making their arguments identical.

E.g.:

A	*B*	θ	$A\theta$	$B\theta$
`dog`	`dog`	$\emptyset$	`dog`	`dog`
`X`	`a`	$\{\texttt{X} = \texttt{a}\}$	`a`	`a`
`X`	`Y`	$\{\texttt{X} = \texttt{Y}\}$	`Y`	`Y`
`f(X, g(t))`	`f(m(h), g(M))`	$\{\texttt{X=m(h), M=t}\}$	`f(m(h), g(t))`	`f(m(h), g(t))`
`f(X, g(t))`	`f(m(h), t(M))`	Impossible (1)		
`f(X, X)`	`f(Y, l(Y))`	Impossible (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**.

Unification Algorithm

Let A and B be two terms:

1. $\theta = \emptyset$, $E = \{A = B\}$
2. while not $E = \emptyset$:
   - 2.1. delete an equation $T = S$ from E
   - 2.2. case T or S (or both) are (distinct) variables. Assuming T variable:
     - (occur check) if T occurs in the term $S \rightarrow$ halt with failure
     - substitute variable T by term S in all terms in θ
     - substitute variable T by term S in all terms in E
     - add $T = S$ to θ
   - 2.3. case T and S are non-variable terms:
     - if their names or arities are different $\rightarrow$ halt with failure
     - obtain the arguments $\{T_1, \ldots, T_n\}$ of T and $\{S_1, \ldots, S_n\}$ of S
     - add $\{T_1 = S_1, \ldots, T_n = S_n\}$ to E
3. halt with θ being the m.g.u of A and B

Unification Algorithm Examples (I)

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

θ	E	T	S
{}	{ p(X,X)=p(f(Z),f(W)) }	p(X,X)	p(f(Z),f(W))
{}	{ X=f(Z), X=f(W) }	X	f(Z)
{ X=f(Z) }	{ f(Z)=f(W) }	f(Z)	f(W)
{ X=f(Z) }	{ Z=W }	Z	W
{ X=f(W), Z=W }	{}		

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

θ	E	T	S
{}	{ p(X,f(Y))=p(Z,X) }	p(X,f(Y))	p(Z,X)
{}	{ X=Z, f(Y)=X }	X	Z
{ X=Z }	{ f(Y)=Z }	f(Y)	Z
{ X=f(Y), Z=f(Y) }	{}		

Unification Algorithm Examples (II)

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

θ	E	T	S
{}	{ p(X,f(Y))=p(a,g(b)) }	p(X,f(Y))	p(a,g(b))
{}	{ X=a, f(Y)=g(b) }	X	a
{ X=a }	{ f(Y)=g(b) }	f(Y)	g(b)
fail			

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

θ	E	T	S
{}	{ p(X,f(X))=p(Z,Z) }	p(X,f(X))	p(Z,Z)
{}	{ X=Z, f(X)=Z }	X	Z
{ X=Z }	{ f(Z)=Z }	f(Z)	Z
fail			

Syntax: Literals and Predicates (Procedures)

- **Literal:** a *predicate name* (like a *functor*) followed by a fixed number of arguments between parentheses:

  Example: `arrives(john,date(monday, Month, 1994))`

  - The arguments are *terms*.
  - The number of arguments is the *arity* of the predicate.
  - Full predicate names are denoted as *name/arity* (e.g., `arrives/2`).

- Literals and terms are syntactically identical!
  But, they are distinguished by context:
  if `dog(name(barry), color(black))` is a literal
  then `name(barry)` and `color(black)` are terms

  if `color(dog(barry,black))` is a literal
  then `dog(barry,black)` is a term

- Literals are used to define procedures and procedure calls. Terms are data structures, so the arguments of literals.

Syntax: Operators

- *Functors* and *predicate names* can be defined as *prefix*, *postfix*, or *infix* $\boxed{\textit{operators}}$ (just syntax!).
- *Examples*:

`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, so that they can be always used.

Syntax: Clauses (Rules and Facts)

- **Rule:** an expression of the form:

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

  - ◇ $p_0(...)$ to $p_m(...)$ are *literals*.
  - ◇ $p_0(...)$ is called the **head** of the rule.
  - ◇ The p_i to the right of `:-` are called **goals** and form the **body** of the rule. They are also called **procedure calls**.
  - ◇ Usually, `:-` is called the **neck** of the rule.

- **Fact:** an expression of the form:

$$\boxed{p(t_1, t_2, \ldots, t_n).}$$

(i.e., a rule with empty body –no neck–).

Syntax: Clauses

Rules and facts are both called **clauses** (since they are clauses in first–order logic) and form the code of a logic program.

- *Example*:

```
meal(soup, beef, coffee).
meal(First, Second, Third) :-
      appetizer(First),
      main_dish(Second),
      dessert(Third).
```

- `:-` stands for $\leftarrow$, i.e., logical implication (but written "backwards"). Comma is conjunction.
  - Therefore, the above rule stands for:

$$\texttt{appetizer(First)} \wedge \texttt{main\_dish(Second)} \wedge \texttt{dessert(Third)} \rightarrow \texttt{meal(First, Second, Third)}$$

  - And thus, is a *Horn clause* of the form:

$$\neg\, \texttt{appetizer(First)} \vee \neg\, \texttt{main\_dish(Second)} \vee \neg\, \texttt{dessert(Third)} \vee \texttt{meal(First, Second, Third)}$$

Syntax: Predicates and Programs

- **Predicate** (or *procedure definition*): a set of clauses whose heads have the same name and arity (the **predicate name**).

  Examples:

  ```
  pet(barry).                            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.

- **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).

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

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 " $\texttt{p} \leftarrow \texttt{true}$ ")

  Example: the fact $\boxed{\texttt{animal(spot).}}$
  can be read as "spot is an animal".

- **Rules:** state implications that are true.

  - p :- $p_1, \cdots, p_m$. represents $p_1 \wedge \cdots \wedge p_m \rightarrow p$.
  - 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 $\boxed{\texttt{pet(X) :- animal(X), barks(X).}}$
  can be read as "X is a pet if it is an animal and it barks".

Declarative Meaning of Predicates and Programs

- **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 ways for `X` to be a pet.

- **Programs** are sets of logic formulae, i.e., a first–order theory: a set of statements assumed to be true. In fact, a set of Horn clauses.
  - The declarative meaning of a program is the set of all (ground) facts that can be logically deduced from it.

Queries

- **Query:** an expression of the form:

  $$\boxed{\texttt{?-}\ p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_m}^n).}$$

  (i.e., a clause without a head)
  (`?-` stands also for $\leftarrow$).

  - ◇ The p_i to the right of `?-` are called **goals** (*procedure calls*).
  - ◇ Sometimes, also the whole query is called a (complex) goal.

- A *query* is a clause to be deduced:

  Example: `?- pet(X).`
  can be seen as "`true` $\leftarrow$ `pet(X)`", i.e., "$\neg$ `pet(X)`"

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

  Examples:

  `?- pet(spot).`
  asks whether `spot` is a pet.

  `?- pet(X).`
  asks: "Is there an X which is a pet?"

Execution

- Example of a **logic program**:

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

animal(tim).               meows(tim).
animal(spot).              barks(spot).
animal(hobbes).            roars(hobbes).
```

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

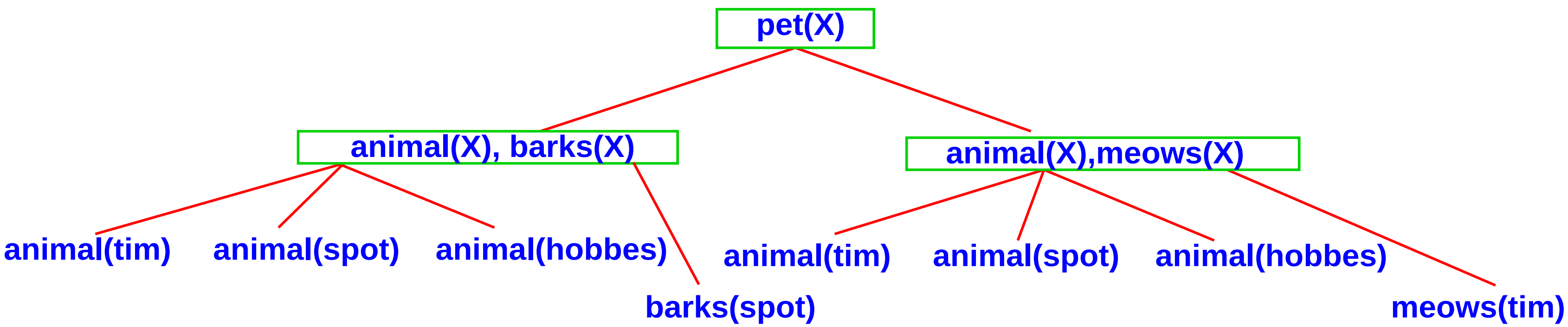
  Example: given the program above and the query `?- pet(X).`
  the system will try to find a “solution” for `X` which makes `pet(X)` true.

- This can be done in several ways:
  - ◇ View the program as a set of formulae and apply deduction.
  - ◇ View the program as a set of clauses and apply SLD-resolution.
  - ◇ View the program as a set of procedure definitions and execute the procedure calls corresponding to the queries.

The Search Tree

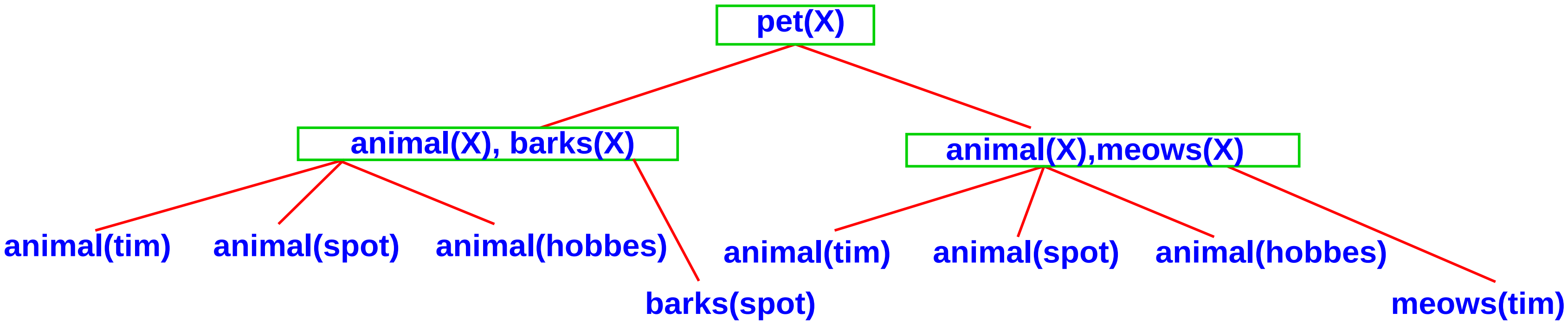
- A query + a logic program together specify a *search tree*.

  Example: query `?- pet(X)` with the previous program generates this search tree (the boxes represent the "and" parts [except leaves]):



- Different query $\rightarrow$ different tree.
- A particular execution strategy defines how the search tree will be explored during execution.
- Note: execution always finishes in the leaves (the facts).

Exploring the Search Tree



- Explore the tree top–down $\rightarrow$ "call"
- Explore the tree bottom–up $\rightarrow$ "deduce"
- Explore goals in boxes left–to–right or right–to–left
- Explore branches left–to–right or right–to–left
- Explore goals in boxes all at the same time
- Explore branches all at the same time
- ...

Running Programs: Interaction with the System

- Practical systems implement a particular strategy (all Prolog systems implement the same one).
- The strategy is meant to explore the whole tree, but returns solutions one by one:

  Example: (`?-` is the system prompt)

```
?- pet(X).
X = spot ?
yes
?-
```

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

- Prolog systems also allow to create executables that start with a given predefined query (which is usually `main`/0 and/or `main`/n).
- Some systems allow to introduce queries in the text of the program, starting with `:-` (remember: a rule without head). These are executed upon loading the file (or starting the executable).

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 `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) means:

  $\texttt{p :- p}_1\texttt{, ..., p}_n$

  $\texttt{p :- q}_1\texttt{, ..., q}_m$

  $\vdots$

  "to execute a call to `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.

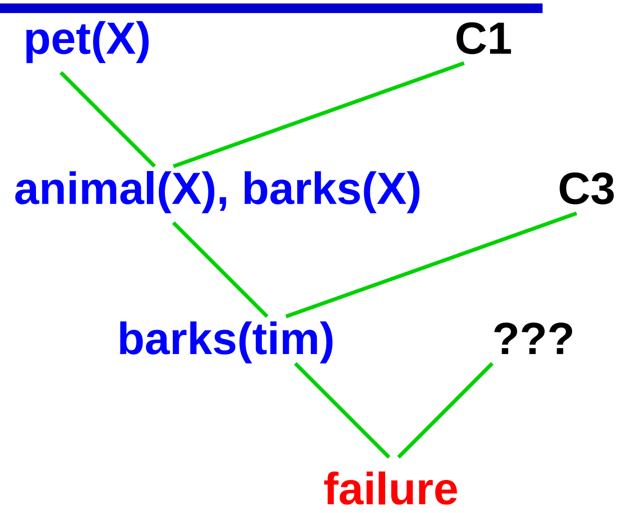
A (Schematic) Interpreter for Logic Programs (Prolog)

Let a logic program P and a query Q,

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 as A
      3.2.1. If there is a solution θ to $A = A'$ (*unification*) continue
      3.2.2. Otherwise, take next clause and repeat
      3.2.3. If there are no more clauses, explore the last pending branch
      3.2.4. If there are no pending branches, output *failure*
   3.3. Replace A in R by $B_1, \ldots, B_n$
   3.4. Apply θ to R and Q
4. Output solution μ to $Q = Q'$
5. Explore last pending branch for more solutions (upon request)

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)

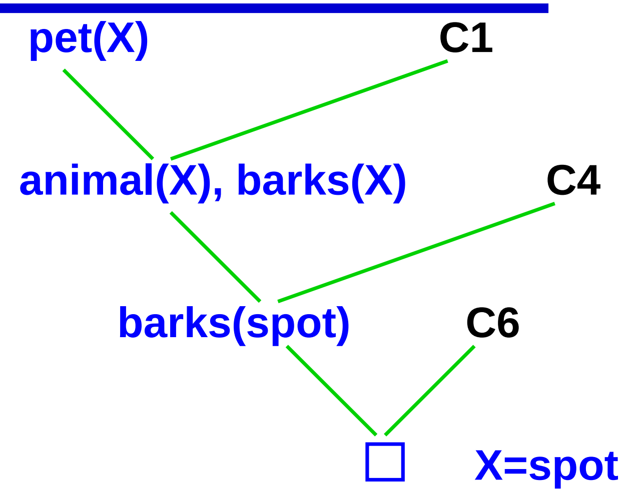
Q	R	**Clause**	θ
pet(X)	<u>pet(X)</u>	C_1*	{ X=X_1 }
pet(X_1)	<u>animal(X_1)</u>, barks(X_1)	C_3*	{ X_1=tim }
pet(tim)	<u>barks(tim)</u>	???	*failure*

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

- But solutions exist in other paths!

Running Programs: Different Branches

```
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)	—	—	—

Backtracking (Prolog)

- **Backtracking** is the way in which Prolog execution strategy explores different branches of the search tree.
- It is a kind of "*backwards execution*".
- (Schematic) Algorithm:

  "Explore the last pending branch" means:

  1. Take the last literal successfully executed
  2. Take the clause against which it was executed
  3. Take the unifier of the literal and the clause head
  4. Undo the unifications
  5. Go to 3.2.2 (forwards execution again)

- **Shallow backtracking:** the clause selection performed in 3.2.2.
- **Deep backtracking:** the application of the above procedure (undo the execution of the previous goal(s)).

Running Programs: Complete Execution (All Solutions)

C_1: `pet(X) :- animal(X), barks(X).`

C_2: `pet(X) :- animal(X), meows(X).`

C_3: `animal(tim).`

C_4: `animal(spot).`

C_5: `animal(hobbes).`

C_6: `barks(spot).`

C_7: `meows(tim).`

C_8: `roars(hobbes).`

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

Q	R	**Clause**	θ	Choice-points		
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*			
	deep backtracking				*	
pet(X_1)	animal(X_1), barks(X_1)	C_4*	{ X_1=spot }		*	
pet(spot)	barks(spot)	C_6	{}			
pet(spot)	—	—	—			
;	triggers backtracking				*	
continues...						

Running Programs: Complete Execution (All Solutions)

```
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).
```

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

Q	R	**Clause**	θ	Choice-points		
pet(X_1)	animal(X_1), barks(X_1)	C_5	{ X_1=hobbes }			
pet(hobbes)	barks(hobbes)	???	*failure*			
	deep backtracking					*
pet(X)	pet(X)	C_2	{ X=X_2 }			
pet(X_2)	animal(X_2), meows(X_2)	C_3*	{ X_2=tim }		*	
pet(tim)	meows(tim)	C_7	{}			
pet(tim)	—	—	—			
;	triggers backtracking				*	
continues...						

Running Programs: Complete Execution (All Solutions)

C_1: `pet(X) :- animal(X), barks(X).`
C_2: `pet(X) :- animal(X), meows(X).`
C_3: `animal(tim).`
C_4: `animal(spot).`
C_5: `animal(hobbes).`

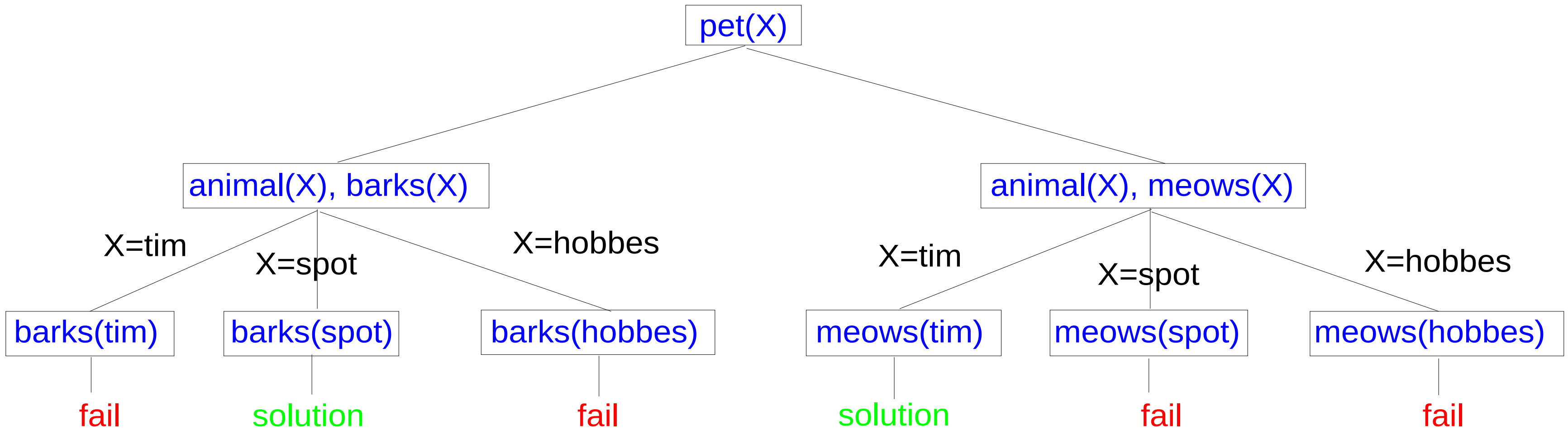
C_6: `barks(spot).`
C_7: `meows(tim).`
C_8: `roars(hobbes).`

- $\boxed{\texttt{?- pet(X).}}$ (continued)

Q	R	**Clause**	θ	Choice-points		
pet(X_2)	$\underline{\text{animal}(X_2)}$, meows($X_2$)	C_4*	{ X_2=spot }		*	
pet(spot)	$\underline{\text{meows(spot)}}$	???	*failure*			
	deep backtracking				*	
pet(X_2)	$\underline{\text{animal}(X_2)}$, meows($X_2$)	C_5	{ X_2=hobbes }			
pet(hobbes)	$\underline{\text{meows(hobbes)}}$	???	*failure*			
	deep backtracking					
failure						

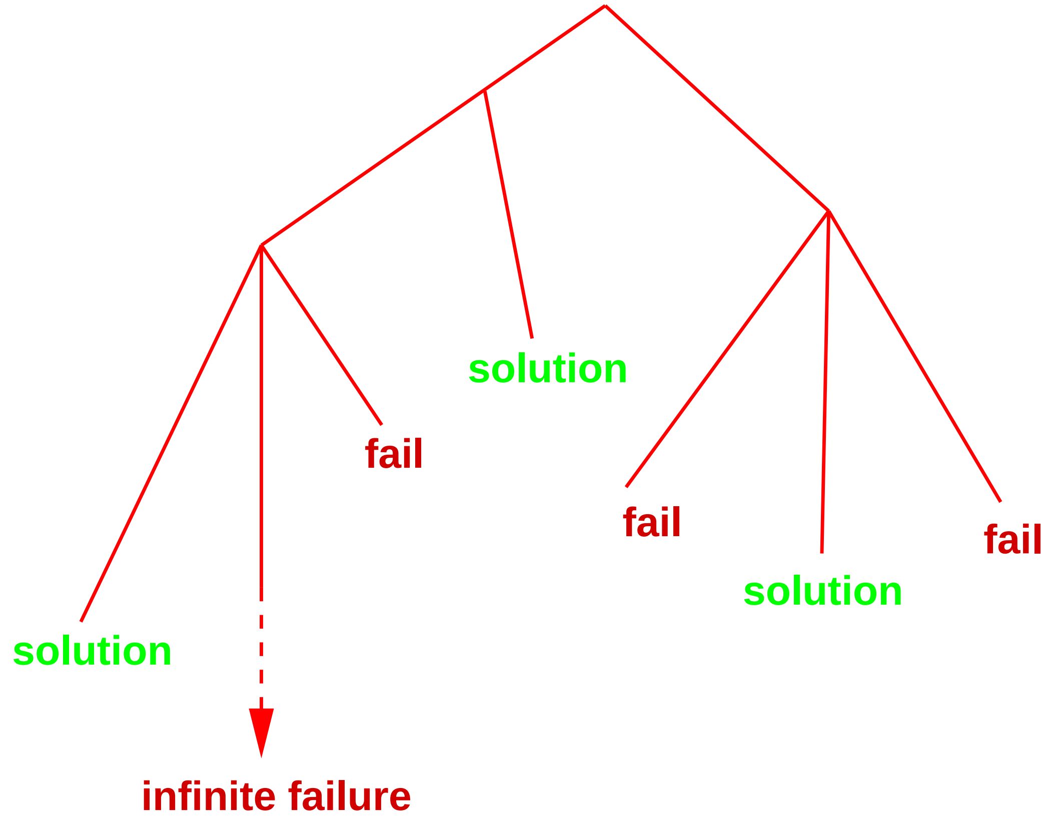
The Search Tree Revisited

- Different execution strategies explore the tree in a different way.
- A strategy is complete if it guarantees that it will find all existing solutions.
- Prolog does it top-down, left-to-right (i.e., depth-first).



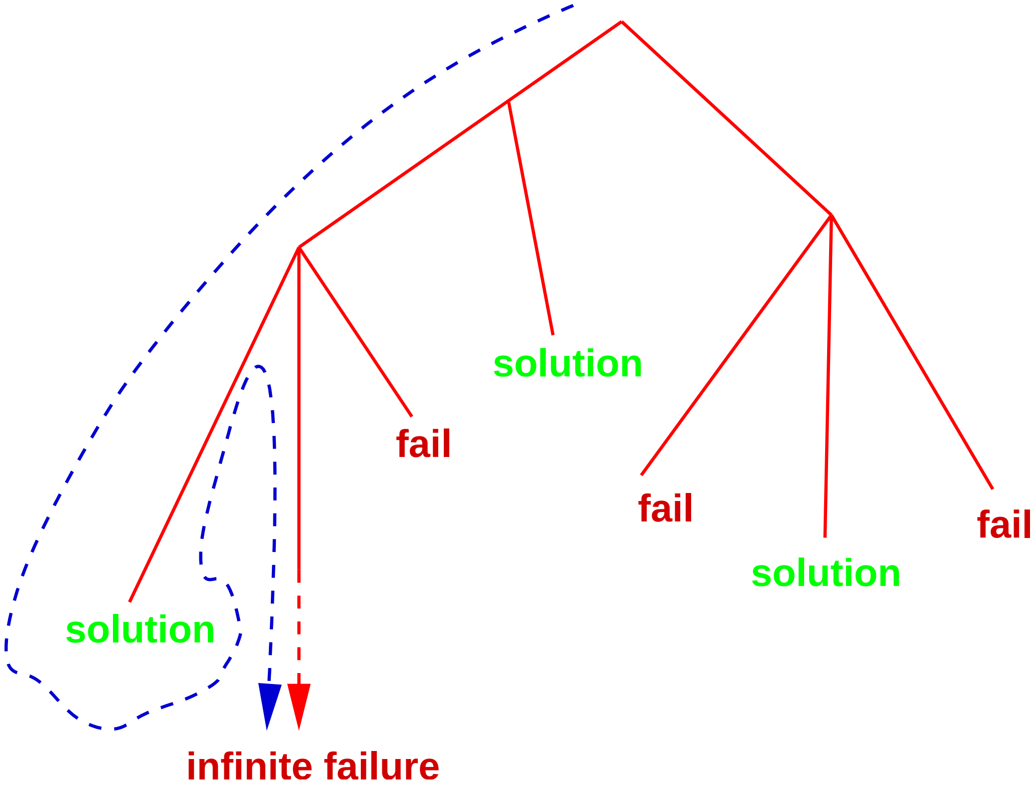

```
pet(X) :- animal(X), barks(X).        animal(tim).       barks(spot).
pet(X) :- animal(X), meows(X).        animal(spot).
                                      animal(hobbes).    meows(tim).
```

Characterization of the Search Tree



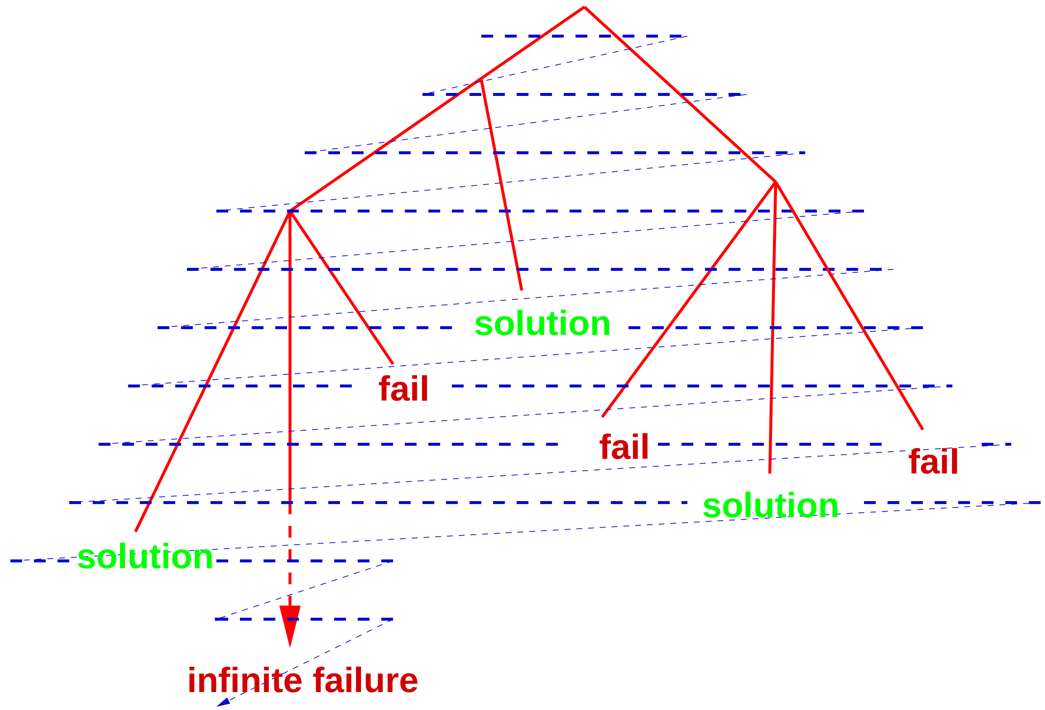
- All solutions are at *finite depth* in the tree.
- Failures can be at finite depth or, in some cases, be an infinite branch.

Depth-First Search



- Incomplete: may fall through an infinite branch before finding all solutions.
- But very efficient: it can be implemented with a call stack, very similar to a traditional programming language.

Breadth-First Search



- Will find all solutions before falling through an infinite branch.
- But costly in terms of time and memory.
- Used in some of our examples (via Ciao's `bf` package).

The Execution Mechanism of Prolog

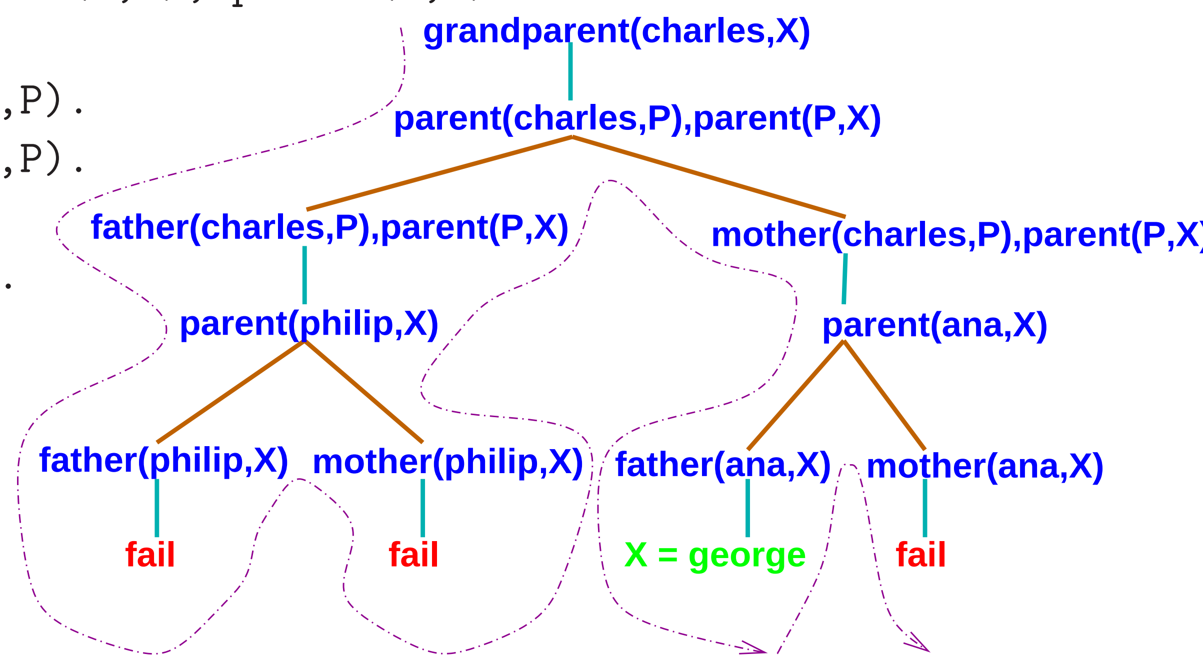
- Always execute literals in the body of clauses *left-to-right*.
- At a *choice point*, take *first unifying clause* (i.e., the leftmost unexplored branch).
- On failure, backtrack to the *next unexplored clause of last choice point*.

```
grandparent(C,G):- parent(C,P), parent(P,G).

parent(C,P):- father(C,P).
parent(C,P):- mother(C,P).

father(charles,philip).
father(ana,george).

mother(charles,ana).
```



- Check how Prolog explores this tree by running the **debugger**!

Comparison with Conventional Languages

- 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 call to `q/2` finishes (with "success"), execution continues in the next procedure call (literal) in `p/2`, i.e., 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).
  q(X,Z) :- ...
  q(X,Z) :- ...
  ```

  if the call to `q/2` succeeds, it is as above, but if it fails at any point, execution continues ("backtracks") at the second clause of `q/2` (the *backward continuation*).

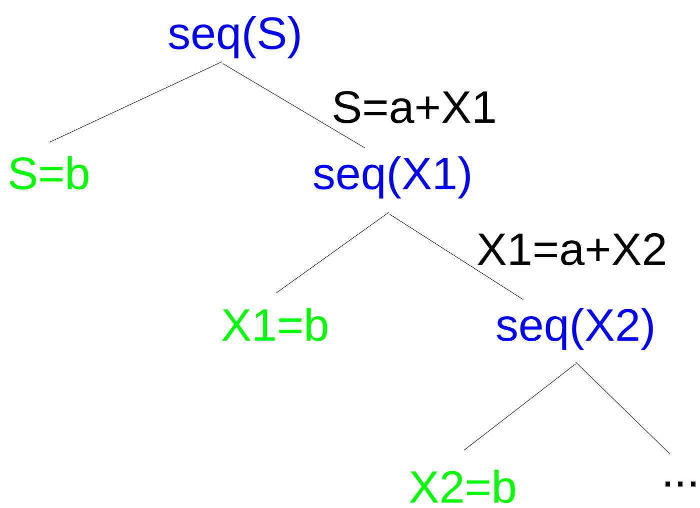
- Again, the debugger (see later) can be useful to observe execution.

Ordering of Clauses and Goals

- Since the execution strategy of Prolog is fixed, the ordering in which the programmer writes clauses and goals is important.
- Ordering of clauses determines the order in which alternative paths are explored. Thus:
  - ◇ The order in which solutions are found.
  - ◇ The order in which failure occurs (and backtracking triggered).
  - ◇ The order in which infinite failure occurs (and the program flounders).
- Ordering of goals determines the order in which unification is performed. Thus:
  - ◇ The selection of clauses during execution. That is:
    the order in which alternative paths are explored.
- The order in which failure occurs affects the size of the computation (efficiency).
- The order in which infinite failure occurs affects completeness (termination).

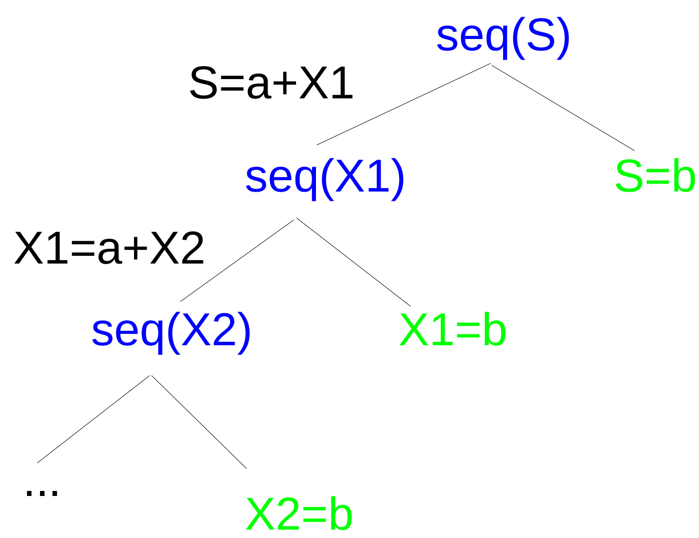
Ordering of Clauses

```
seq(b).
seq(a+X):- seq(X).
```



- An infinite computation which yields all solutions

```
seq(a+X):- seq(X).
seq(b).
```

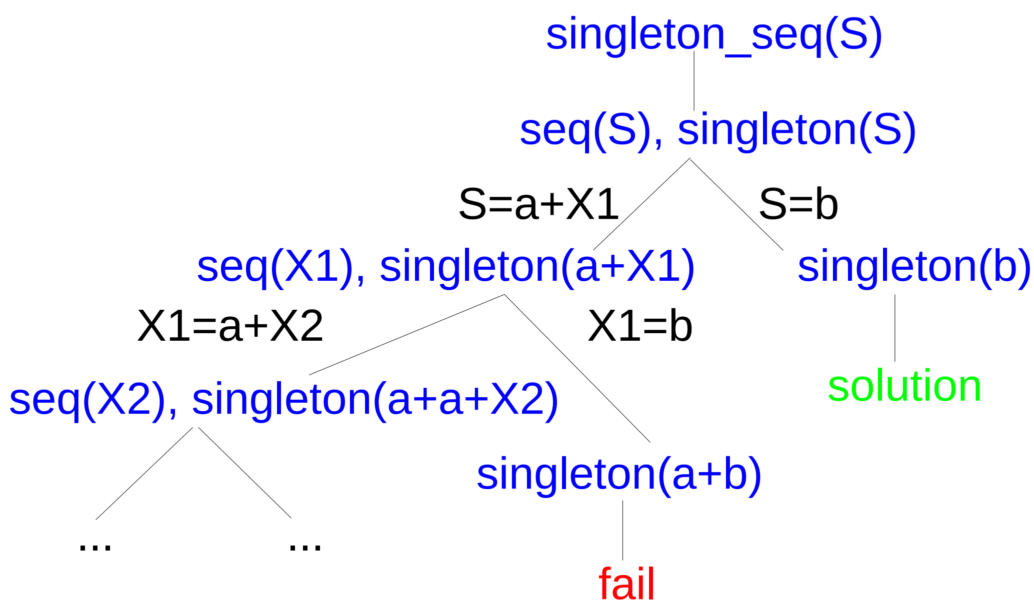


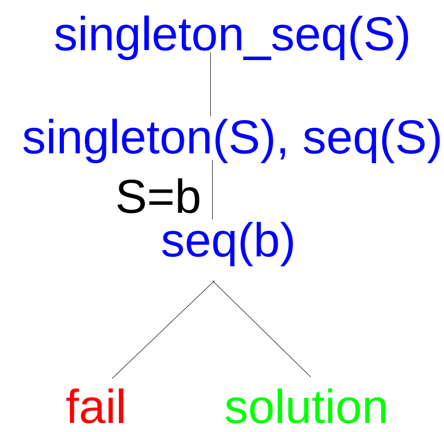
- An infinite computation with no solutions (infinite failure)

Ordering of Goals

```
seq(a+X):- seq(X).                    singleton(b).
seq(b).

singleton_seq(X):- seq(X),            singleton_seq(X):- singleton(X),
                   singleton(X).                         seq(X).
```





- A finite failure plus all solutions (1)

Execution Strategies

- **Search rule(s):** how are clauses/branches selected in the search tree (step 3.2 of the resolution algorithm).
- **Computation rule(s):** how are goals selected in the boxes of the search tree (step 3.1 of the resolution algorithm).

- Prolog execution strategy:
  - ◇ Computation rule: left–to–right (as written)
  - ◇ Search rule: top–down (as written)

Summary

- A logic program declares known information in the form of rules (implications) and facts.
- Executing a logic program is deducing new information.
- A logic program can be executed in any way which is equivalent to deducing the query from the program.
- Different execution strategies have different consequences on the computation of programs.
- Prolog is a logic programming language which uses a particular strategy (and goes beyond logic because of its predefined predicates).

Exercise

- Write a predicate jefe/2 which lists who is boss of whom (a list of facts). It reads:

  jefe(X,Y) iff X is direct boss of Y.

- Write a predicate curritos/2 which lists pairs of people who have the same direct boss (should not be a list of facts). It reads:

  curritos(X,Y) iff X and Y have a common direct boss.

- Write a predicate jefazo/2 (no facts) which reads:

  jefazo(X,Y) iff X is above Y in the chain of “who is boss of whom”.