Programmazione Logica e Rudimenti di Prolog

Advanced School in AI in Emilia Romagna

Roberta Calegari

roberta.calegari@unibo.it

Alma Mater Studiorum – Università di Bologna

24 July 2023

Next in Line. . .

Context: why?

→ AI systems to formalize, scale, and accelerate processes

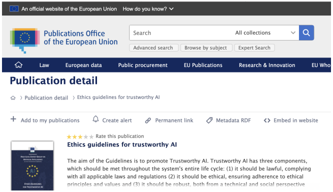
→ *trust* these systems

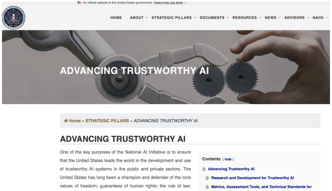
Europe Strategy

- Ethics Guidelines for Trustworthy AI (EG-TAI) [European Commission, 2019]
- First AI regulation (the "AI Act", 2021) [Act, 2021]
  - *ensuring* that AI systems, introduced on the *EU market* are trustworthy
  - creating *legal certainty* to facilitate investments and innovation in AI
- TAI is the basis for the development, deployment and use of AI in Europe

⇒ **close the AI "trust gap"**

Explainable AI: why?





EG-TAI: TAI Requirements

Main pillars

- lawfulness
- ethics
- robustness

Seven specific requirements – dimensions to be audited – of an AI system:

1. human agency and oversight
2. technical robustness and safety
3. privacy and data governance
4. transparency (traceability, explainability)
5. diversity, non-discrimination and *fairness*
6. societal and environmental well-being
7. accountability

Why logic & logic programming?

> *"What is or can be the added value of logic programming for implementing machine ethics and explainable AI?"*

Three main features of LP:

(i) being a declarative paradigm

(ii) working as a tool for knowledge representation

(iii) allowing for different forms of reasoning and inference

Why logic programming?

LP features

Provability

- correctness, completeness, well-founded extension

Explainability

- formal methods for argumentation-, justification-, and counterfactual often based on LP [Saptawijaya and Pereira, 2019]

Expressivity and situatedness

- different nuances → extensions [Dyckhoff et al., 1996]
- explicit assumptions and exceptions [Borning et al., 1989]
- capture the specificities of the context [Calegari et al., 2018b]

Hybridity

- integration of diversity [Calegari et al., 2018a]

Next in Line. . .

Origins I

Early history [Apt, 2005]

- automatic deduction of theorems
- *first-order logic* (FOL) by Frege, Peano and Russell
- computation as deduction by Gödel and Herbrand
- *resolution* principle by Robinson [Robinson, 1965], along with *unification*

The key issue

- resolution by Robinson
  - allowed proof of FOL theorem made it possible to compute with logic
  - not yet to see logic as a full computational framework
- from computable logic to logic as a programming language something was still missing

Origins II

The procedural interpretation of Horn clauses

- by defining logic programs as collections of Horn clauses
- by restricting Robinson's principle accordingly
- Kowalski showed how a logical implication could be amenable of both a *declarative* and a *procedural* interpretation [Kowalski, 1974]
- thus providing the *foundations* for a logic programming language
- Prolog, by Colmerauer in Marseille, came along in 1973

> *There is no question that Prolog is essentially a theorem prover à la Robinson. Our contribution was to transform that theorem prover into a programming language.* [Colmerauer and Roussel, 1996]

Essentials I

Three fundamental features [Apt, 2005]

terms *Computing takes place over the domain of all terms defined over a "universal" alphabet.*

mgu *Values are assigned to variables by means of automatically-generated substitutions, called most general unifiers. These values may contain variables, called logical variables.*

backtracking *The control is provided by a single mechanism: automatic backtracking.*

Other Features I

Declarative programming

- according to Aristotle, declarative is a sentence that can be said either *true* or *false* [Rijk, 2002]

→ declarative programming means first of all programming through (true) sentences, which declare *what* to compute—the meaning

- procedural programming is instead programming through *operational statements*, which determine *how* to compute—the method
- e.g., in object-oriented languages, classes and interfaces are defined declaratively, whereas methods are defined procedurally
- logic programming is amenable of either a declarative or an operational interpretation, and the two corresponding *semantics* match [Kowalski, 1974]

Other Features II

Declarative programming: features and issues [Apt, 2005]

- logic programs can be seen as executable specifications
  - the logic programmer is concerned on *what* to compute
  - *how* to compute (*control*) is delegated to the underlying (logic programming) *machinery*

! sometimes this could lead to *inefficiency*

- logic programming languages can be seen as formalisms for either executable code or *knowledge representation*

→ languages for artificial intelligence

Other Features III

Interactive programming

- the model behind the notion of *computation as deduction* natively supports the idea of writing a logic program, then interact with the logic machinery by means of multiple queries, or, by asking for multiple solutions
- logic languages intrinsically support the interactive style of programming and computing

! while this will be evident in the lab session, it should be already clear how such a feature could be useful in distributed systems, supporting novel notions such as LPaaS (Logic Programming as a Service) [Calegari et al., 2018b]

Basic Units of Computation I

Atomic actions [Apt, 2005]

- logic programming is a different paradigm for programming languages
- since it is ruled by different *principles* w.r.t. the other sorts of programming languages
  - atomic actions are *equations* between *terms*
  - executed by means of the *unification* process trying to solve them
  - unification assigns *values* to *variables*
  - values can be *arbitrary terms*—in fact, there is just one sort of variable, ranging over the set of all terms
- so, in order to understand logic programming as a computational paradigm, we first need to understand its basic units of computation

Basic Units of Computation II

Terms: Definition

- a *variable* is a *term*
- a *functor* (or, *function symbol*) with arity 0 is called a *constant*, and is a term
- if f is a functor of arity n, and $t_1, \ldots, t_n$ are n terms, then $f(t_1, \ldots, t_n)$ is a term

Basic Units of Computation III

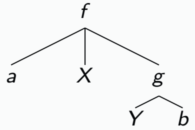
Terms: Examples

- let's say that X, Y are variables, a, b constants (or, functors of arity 0), f, g functors of arity $3, 2$ respectively. Then
  - a, b, X, and Y are proper terms
  - $f(a, b, a)$ and $g(X, Y)$ are proper terms
  - $f(a, X, g(Y, b))$ is a proper term
- variables and constant are *atomic terms*, terms built out of proper functors are *structured terms*. Then
  - a, b, X, and Y are atomic terms
  - $f(a, b, a)$, $g(X, Y)$ and $f(a, X, g(Y, b))$ are structured terms
  - ! in the structured term $f(a, X, g(Y, b))$, f is the *functor symbol* of *arity* 3, whereas $a, X, g(Y, b)$ are the three *subterms*

Basic Units of Computation IV

Terms: Remarks

- a *recursive* definition, leading to a recursive data structure—a tree
  - e.g., structured term $f(a, X, g(Y, b))$ maps onto tree



- fundamental in mathematical logic, terms are *essential in computer science*, too: e.g., they capture both arithmetic expressions and strings
- no specific alphabet is assumed—*universal alphabet* for all terms
- no meaning is a *a priori* attached to symbols, in particular to functors—e.g., $+$ is just a functor, not associated a priori with the plus sign of arithmetic

$\rightarrow$ no types

Terms: Semantics

Logic Formulae I

Basic question

- since logic programs compute over the truth values of sentences, how do we write sentences?
- we know how to denote the elements of the domain of discourse, not how to talk about them
- sentences, in logic, are typically called *propositions*

Logic Formulae II

Predicate and atoms

- predicates can be used to write propositions in logic programming
- if p is a *predicate symbol* of arity n, $t_1, \ldots, t_n$ are terms, then

$$p(t_1, \ldots, t_n)$$

  is an atom
- atoms represent *elementary propositions* in logic programming
- if A is an atom, then

  atoms A is a logic formula, stating that A is true

Logic Formulae III

Negation and literals

- negation makes it possible to deal with false propositions
- if A is an atom, then

  negation $\neg A$ (read: not A) is a logic formula, stating that A is false

  literals A, $\neg A$ are *literals*

Logic Formulae IV

Logical connectives

- literals can be combined through *logical connectives* to build articulated *logic formulae*
- if A, B are literals, then

conjunction $A \wedge B$ (read: A and B) is a logic formula, stating that both A and B are true

disjunction $A \vee B$ (read: A or B) is a logic formula, stating that either A or B are true

implication $A \rightarrow B$ (read: A implies B) is a logic formula, stating that if A is true then B is true

equivalence $A \leftrightarrow B$ (read: A is equivalent to B) is a logic formula, stating that A is true if and only if B is true

Logic Programs I

Logic clause

- a logic clause is a (finite) disjunction of literals [Console et al., 1997]
- if $A_1, \ldots, A_n, B_1 \ldots, B_m$ are atoms, containing variables $X_1, \ldots, X_k$, then

$$\forall X_1, \ldots, X_k (A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m)$$

  is a logic clause, which is *logically equivalent* to

$$\forall X_1, \ldots, X_k ((A_1 \vee \ldots \vee A_n) \leftarrow (B_1 \wedge \ldots \wedge B_m))$$

  usually written simply as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

- a clausal normal form (CNF) is a *conjunction of clauses*

Logic Programs II

Definite clauses

- a definite clause, has just one positive literal ($n = 1$)

$$A \leftarrow B_1, \ldots, B_m$$

- a unitary clause, is a definite clause with no negative literal ($m = 0, n = 1$)

$$A \leftarrow$$

- a definite goal is a definite clause with no positive literal ($n = 0$)

$$\leftarrow B_1, \ldots, B_m$$

Horn clauses

- a Horn clause is either a definite clause or a definite goal ($n = 1$ or $n = 0$)

Logic Programs III

Logic program

- in a logic program
  - a definite clause is called a rule
  - a unitary clause is a fact
  - a definite goal is just a goal
- a logic program is a CNF of Horn clauses
  - so, it is a conjunction of rules and facts (and goals)

. . . a logic program is a conjunction of Horn clauses. . . *waitbutwhy*???

Goals & Proofs I

Resolution principle

- Robinson's resolution principle works for general clauses [Robinson, 1965]
  - given a CNF H and a formula F, it shows that it is possible to compute (by contradiction) whether H logically entails F
  - however, it does not provide a proof strategy for a full-fledged logic programming language
- Kowalski showed that this could be obtained by *restricting* logic programs to CNF of *Horn clauses*, and re-casting Robinson's principle accordingly [Kowalski, 1974]
  - given a CNF H and a formula F, it shows that it is possible to compute (by contradiction) whether H logically entails F
  - so-called SLD-resolution principle [Nilsson and Maluszynski, 1995]

Goals & Proofs II

Declarative vs. procedural interpretation

- a definite clause $A \leftarrow B_1, \ldots, B_m$ is amenable of either a *declarative* or a *procedural interpretation*

  declarative interpretation A is true if $B_1, \ldots, B_m$ are true

  procedural interpretation to prove A, prove $B_1, \ldots, B_m$

- the two interpretations coincide [Kowalski, 1974]

! logic programming languages such as Prolog are the only ones for which this property holds [Metakides and Nerode, 1996]

Goals & Proofs III

Proving goals

- Robinson's principle proceed by contradiction, trying to prove a formula F false against CNF H, succeeding if this fails
  - technically, proving that $H \cup \neg F$ is not satisfiable
- proving an atom G in logic programming amounts at proving $\neg G$ against logic program P
  - technically, proving goal $\leftarrow G$ on P
- computation in logic programming proceeds by *proving goals*

! resolution leads to *backward chaining*—from goal back to axioms

Goals & Proofs IV

SLD resolution informally

- to prove a goal G w.r.t. program P, the resolution principle for logic programming proceeds according to the procedural interpretation
- so, first we look for one clause $A \leftarrow B_1, \ldots, B_n$ in P whose head A unifies with G
- if the most general unifier of G and A is θ ($mgu(G, A) = \theta$), then the proof of G succeeds if we can further prove $B_1\theta, \ldots, B_n\theta$—where $B_i\theta$ represents the application of the *mgu* θ to B_i

! the application of θ to clause $A \leftarrow B_1, \ldots, B_n$ specialises the clause to the specific atom we need to proof—that is, our current goal

! resolution proceed recursively with the proof of *subgoals* $B_1\theta, \ldots, B_n\theta$

→ in general, the computational state of the SLD resolution include a (possibly empty) conjunction of atom (goals) $G_1, \ldots, G_n$ to be proven—the *current goal* of the proof

Goals & Proofs V

SLD Resolution: how it ends—*if* it does

- when the current goal is empty, the proof (called *SLD derivation*) ends as a *successful* one—SLD refutation
- when the current goal is not empty, a selection rule $\mathbb{R}$ is used to select the subgoal to prove (one if the execution is sequential)
- if the selected goal matches no head of the clauses in the program, the proof *fails*
- if the current goal never gets emptied, but there is always a clause whose head matches the selected subgoal, the SLD derivation *does not terminate*

Goals & Proofs VI

SLD resolution: inference rule

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, , B_n}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, , B_n, A_{i+1}, \ldots, A_m)\theta}$$

- $A_1, \ldots, A_m$ are atomic formulas
  - $\leftarrow A_1, \ldots, A_m$ is the list / set / conjunction of the subgoals to prove
- $B_0 \leftarrow B_1, \ldots, , B_n$ is a definite clause in program P ($n \geq 0$)
  - suitably *renamed* (that is, with new and uniques variable names) to avoid name clashes
- there is an A_i unifying with B_0 such that $mgu(A_i, B_0) = \theta$

Goals & Proofs VII

Non-determinism of SLD resolution

- **or** more than one clause could unify (through its head) with our current goal: we could choose either one of them for the resolution step
- **and** more than one goal could be subject to proof at the same time (as for $B_1\theta, \ldots, B_n\theta$): we could proceed by choosing either one of them—through a selection rule

- the choice do not affect correctness of the resolution, so we could choose *non-deterministically*

! how to exploit either *or-nondeterminism* or *and-nondeterminism*, or both, determines how the automatic resolution process explores the proof tree

! also, different computational models (sequential, parallel, concurrent) could be exploited to explore the proof tree—e.g., more clauses with a unifying head could be used for goal proof at the same time, either parallel or concurrently

An Example I

A simple logic program

$parent(joey, luca)$
$parent(joey, simone)$
$parent(lino, joey)$
$parent(mirella, joey)$

$grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$

An Example II

Declarative interpretation

- four *facts* are expressed by means of predicate $parent/2$
  - four propositions that are considered true with no need of proof—our axioms
  - a possible interpretation is that, e.g., *joey* is a parent of *luca*—just one of the many, even though the most intuitive for English speakers
- one *rule* is expressed by means of predicate $grandparent/2$
  - since it is the short form for

    $$\forall X, Y, Z, grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$$

    it means that formula $grandparent(X, Z)$ holds if both $parent(X, Y)$ and $parent(Y, Z)$ are true, whatever the values of X, Y, Z
  - so, it can be used to prove the truth of, e.g., formula $grandparent(lino, luca)$ since both $parent(joey, luca)$ and $parent(lino, joey)$ are true since they are facts in the logic program
  - independently of the possible interpretations

An Example III

Procedural interpretation

- two procedure are defined: *parent*/2 and *grandparent*/2
- two (procedure) calls can be executed correspondingly—goals of the form
  - $\leftarrow parent(?, ?)$
  - $\leftarrow grandparent(?, ?)$

  with any sort of term in the place of the ?
  - for instance, $\leftarrow grandparent(lino, luca)$
- to compute *parent*/2 we can use the four facts, non-deterministically
- to compute *grandparent*/2 we can use the rule, first matching the rule head, then proceeding by calling the two subprocedures, via the two *subgoals* of the form *parent*/2
  - for instance, to compute $\leftarrow grandparent(lino, luca)$ we will compute subgoals $\leftarrow parent(lino, Y)$ and $\leftarrow parent(Y, luca)$

An Example IV

Possible goals

- $grandparent(lino, luca)$ succeeds—one refutation, no *computed substitution*
- $grandparent(lino, joey)$ fails—no refutations
- $grandparent(lino, X)$ succeeds twice—two refutations, two different computed substitutions
  - $X/luca$
  - $X/simone$
- $grandparent(X, simone)$ succeeds twice—two refutations, two different computed substitutions
  - $X/lino$
  - $X/mirella$
- $grandparent(X, Y)$ succeeds four times—four refutations, four different computed substitutions
  - $X/lino$, $Y/luca$
  - $X/lino$, $Y/simone$
  - $X/mirella$, $Y/luca$
  - $X/mirella$, $Y/simone$

An Example: Remarks

Remarks

From the example we get some early hints about some benefits of logic programming

- multiple uses of the single program
  - the simple program above can be used to test the family relations between known people, or, to compute them
  - mostly, input / output parameters needs not to be defined a priori
- knowledge-based programming
  - arbitrarily complex relations expressed as FOL facts represent the core of a logic program
  - knowledge representation is straightforward in the logic programming formalism—with FOL
- language for rule-based systems
  - classical AI, such as expert systems [Buchanan and Shortliffe, 1984]

SLD Resolution Principle – Example I

A theory (in implication form)

- $parent(abraham, isaac).$
- $parent(isaac, jacob).$
- $parent(sarah, isaac).$
- $parent(jacob, joseph).$
- $parent(jacob, dan).$
- $parent(jacob, dinah).$

- $male(abraham).$
- $male(isaac).$
- $male(jacob).$
- $male(joseph).$
- $male(dan).$

- $son(X, Y) \Leftarrow parent(Y, X) \wedge male(X).$
- $\Leftarrow son(S, jacob).$

SLD Resolution Principle – Example I

The same theory (in disjunctive form)

- $parent(abraham, isaac).$
- $parent(isaac, jacob).$
- $parent(sarah, isaac).$
- $parent(jacob, joseph).$
- $parent(jacob, dan).$
- $parent(jacob, dinah).$
- $male(abraham).$
- $male(isaac).$
- $male(jacob).$
- $male(joseph).$
- $male(dan).$

- $son(X, Y) \lor \neg parent(Y, X) \lor \neg male(X).$
- $\neg son(S, jacob).$

SLD Resolution Principle – Example II

```
parent(abraham, isaac).  %p1
parent(isaac, jacob).    %p2
parent(sarah, isaac).    %p3
parent(jacob, joseph).   %p4
parent(jacob, dan).      %p5
parent(jacob, dinah).    %p6

male(abraham).           %m1
male(isaac).             %m2
male(jacob).             %m3
male(joseph).            %m4
male(dan).               %m5

son(X,Y) :- parent(Y,X), %s1
            male(X).
```

```
?- son(S,jacob).
```

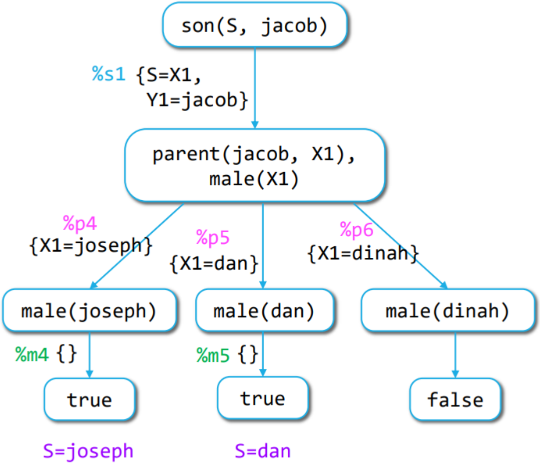


Figure: Proof tree exploration subtended by the query $\Leftarrow son(S, jacob)$.

About the Proof Tree Exploration I

- SL(D) is a non-deterministic algorithm
  - ie at any given step, several choices may be taken
    - aka different paths may be explored
- No prescription concerning which literals should be simplified first
  - aka which rule to try first when multiple ones could apply?
- Possible ways to explore the proof tree:
  - backward chaining (a.k.a. *goal-directed*) — start from a goal and try to solve any sub-goal implying it, recursively
  - forward chaining — start from theory and try to infer anything that can be inferred from it

About the Proof Tree Exploration II

- Possible search strategies to explore the proof tree:

  depth first — explore most recent goals first

  breadth first — explore most recent goals last

  ⋮

- Relevant *properties* a given search strategy should have:

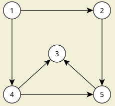
  soundness — any solution found by the strategy is correct

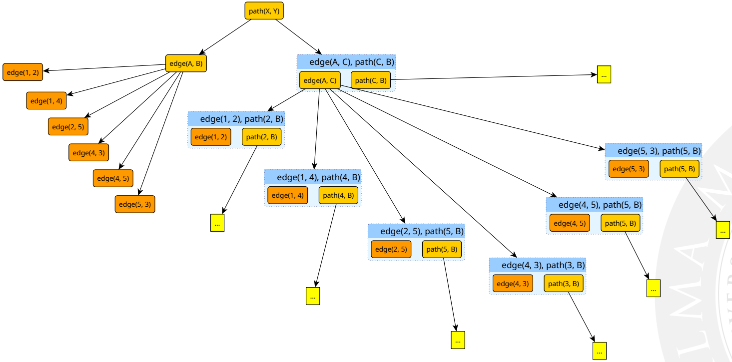
  completeness — the strategy enumerates all correct solution

Proof Tree Exploration – Example

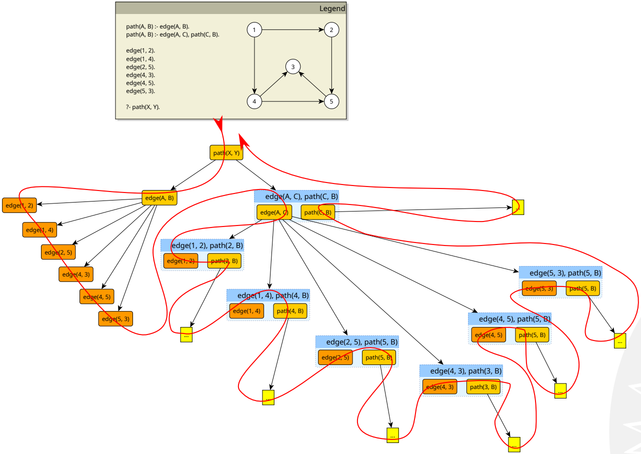
Legend

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

edge(1, 2).
edge(1, 4).
edge(2, 5).
edge(4, 3).
edge(4, 5).
edge(5, 3).

?- path(X, Y).
```

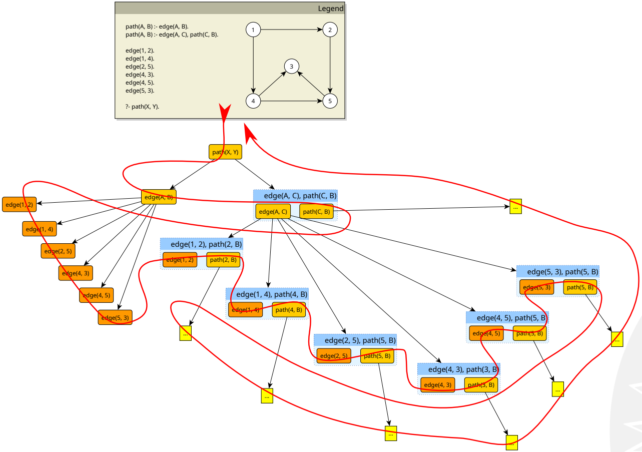




Proof Tree Exploration – Example (depth-first)



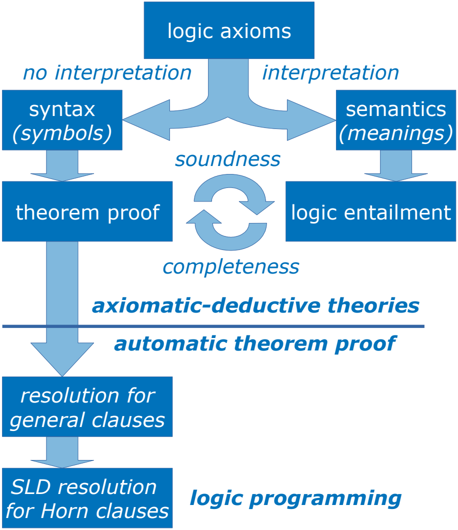
Proof Tree Exploration – Example (breadth-first)



Prolog's Proof Tree Exploration Strategy

- Goal-directed, depth-first, sequential exploration strategy
  - may get stuck in recursive definitions
- Goal-directed → procedural interpretation of Prolog
- Depth-first ≈ left-most goal first, top-most rule first
- Backtracking → sequential exploration
  - *concurrent* implementations may get rid of backtracking
- Support for side-effects *during* resolution
  - eg edits to the knowledge base (a.k.a. assertions and retractions)
  - eg manipulation of exploration procedure (e.g. cut)
  - eg I/O facilities via streams (a.k.a. sources and sinks)

Logic & Logic Programming: Overall Picture



Next in Line...

Prolog examples: facts, atom & predicate I

Imagine we want to encode characters of Homer's Iliad and Odyssey.
→ This translates into Prolog *facts* ended with a period

```
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
character(agamemnon, iliad).
character(patroclus, iliad).
character(hector, iliad).
character(andromache, iliad).
character(rhesus, iliad).
character(ulysses, iliad).
character(menelaus, iliad).
character(helen, iliad).
```

```
character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
character(laertes, odyssey).
character(nestor, odyssey).
character(menelaus, odyssey).
character(helen, odyssey).
character(hermione, odyssey).
```

Prolog examples: facts, atom & predicate II

Fact

Facts are statements that describe object properties or relations between objects.

Knowledge

Such a collection of facts, and later, of rules, makes up a database. It transcribes the knowledge of a particular situation into a logical format. Adding more facts to the database, we express other properties

Prolog examples: facts, atom & predicate III

such as the gender of characters:

```
% Male characters
male(priam).
male(achilles).
male(agamemnon).
male(patroclus).
male(hector).
male(rhesus).
male(ulysses).
male(menelaus).
male(telemachus).
male(laertes).
male(nestor).
```

```
% Female characters
female(hecuba).
female(andromache).
female(helen).
female(penelope).
```

Prolog examples: facts, atom & predicate IV

or relationships between characters such as parentage:

```
% Fathers
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
```

```
% Mothers
mother(hecuba, hector).
mother(penelope, telemachus).
mother(helen, hermione).
```

Prolog examples: facts, atom & predicate V

Finally, if we want to describe kings of some cities and their parties, this would be done as:

```
king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
king(nestor, pylos, achaean).
king(agamemnon, argos, achaean).
king(priam, troy, trojan).
king(rhesus, thrace, trojan).
```

Prolog examples: facts, atom & predicate VI

General form of a Prolog fact

$$relation(object_1, object_2, ..., object_n).$$

Symbols or names representing objects → `ulysses` or `penelope`: *atoms*

Atoms

- strings of letters, digits, underscores begin with a lowercase letter
- can also be a string beginning with an uppercase letter or including white spaces, but it must be enclosed between quotes → `‘Ulysses’` or `‘Pallas Athena’` are legal atoms

Prolog examples: facts, atom & predicate VII

Predicate

- name of the symbolic relation is the *predicate*,
- the objects $object_1, object_2, ..., object_n$ involved in the relation are the *arguments*
- the number n of the arguments is the *arity*

  Traditionally, a Prolog predicate is indicated by its name and arity

  `predicate/arity`

  → e.g., `character/2`, `king/3`

Prolog examples: terms I

Terms

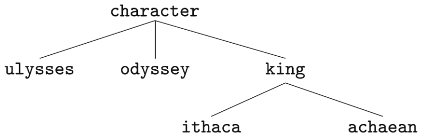
- all forms of data are called *terms*
- constants, i.e., atoms or numbers are terms
- facts, like `king(menelaus, sparta, achaean)`, are a *compound term* or a structure, that is, a term composed of other terms (called subterms)

→ arguments of this compound term are constants

→ can also be other compound terms, as in `character(priam, iliad, king(troy, trojan))` where the arguments of the predicate character/3 are two atoms and a compound term

Prolog examples: tree of terms

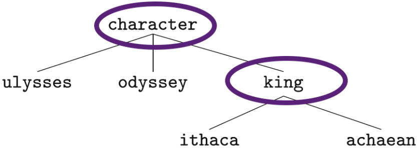
```
character(ulysses, odyssey, king(ithaca, achaean))
```



- nodes of the tree are equivalent to the functors of a term

Prolog examples: tree of terms

```
character(ulysses, odyssey, king(ithaca, achaean))
```



- use trees to represent compound terms
- nodes of the tree are equivalent to the functors of a term

Prolog examples: compound terms

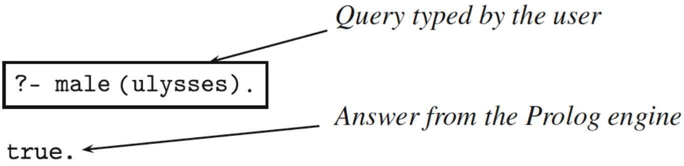
Compound Term

- functor – the name of the relation – and arguments
- leftmost functor of a term is the principal functor
- same principal functor with a different arity corresponds to different predicates: `character/3` is thus different from `character/2`
- constant is a special case of compound term with no arguments and an arity of 0 (can be referred to as `abc/0`)

Prolog examples: query I

Query

- request to prove or retrieve information from the knowledge base
- Prolog answers yes if it can prove it, that is, here if the fact is in the database, or no if it cannot: if the fact is absent (case of asking for a fact tu be proven)
- question *Is Ulysses a male?*



Prolog examples: query II

```
?- male(penelope).
false.
```

- expressions `male(ulysses)` or `male(penelope)` are *goals* to prove
- some questions require more goals, such as *Is Menelaus a male and is he the king of Sparta and an Achaean?*

```
?- male(menelaus), king(menelaus, sparta, achaean).
true.
```

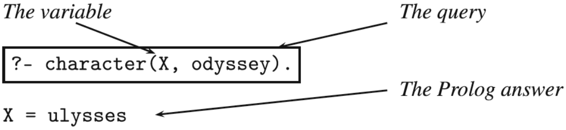
- where "," is the conjunction operator → indicates that Prolog has to prove both goals
- compound queries: conjunction of two or more goals
  `?- G1, G2, G3,..., Gn.`
- Prolog proves that all the goals $G_1 \ldots G_n$ are true

Prolog examples: logical variables I

Logical variables

- begin with an uppercase letter, for example, `X`, `Xyz`, or an underscore
- stand for any term: constants, compound terms, and other variables
- term containing variables such as `character(X, Y)` can *unify* with a compatible fact, such as `character(penelope, odyssey)`, with the *substitutions* `X = penelope` and `Y = odyssey`
- Prolog resolution algorithm searches terms in the database that unify with it → substitutes the variables to the matching arguments
- question *What are the characters of the Odyssey?*

Prolog examples: logical variables II



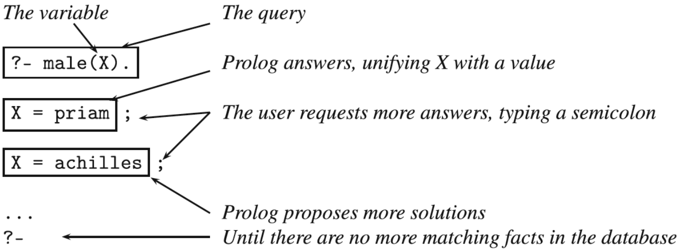
Prolog examples: logical variables III

- question *What is the city and the party of king Menelaus?*

```
?- king(menelaus, X, Y).
X = sparta, Y = achaean

?- character(menelaus, X, king(Y, Z)).
X = iliad, Y = sparta, Z = achaean

?- character(menelaus, X, Y).
X = iliad, Y = king(sparta, achaean)
```

- multiple solutions → Prolog considers the first fact to match the query in the knowledg
- type “;" to get the next answers until there is no more solution

Prolog examples: logical variables IV



Prolog examples: shared variables I

Shared variables

- goals in a conjunctive query can share variables
- constrain arguments of different goals to have the same value
- question *Is the king of Ithaca also a father?*
- conjunction of two goals `king(X, ithaca, Y)` and `father(X, Z)`, with `X` shared between the goals

```
?- king(X, ithaca, Y), father(X, Z).
X = ulysses, Y = achaean, Z = telemachus
```

Prolog examples: shared variables II

- not interested in the name of the child although Prolog responds with `Z = telemachus`.
- can indicate we do not need to know the values of Y and Z using anonymous variables

```
?- king(X, ithaca, _), father(X, _).
X = ulysses
```

Prolog examples: rules I

Rules

- enable to derive a new property or relation from a set of existing ones
- a term called the *head* or *consequent* followed by symbol :- (read if) and a conjunction of goals called *antecedent* or *body*

$$HEAD : -G_1, G_2, G_3, \ldots G_n.$$

- the head is true if the body is true
- variables are shared between the body and the head

Prolog examples: rules II

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).

?- son(telemachus, Y).
Y = ulysses;
Y = penelope;
```

Rules

Flexible way to *deduce* new information from a set of facts.

The `parent/2` predicate is another example of a family relationship that is easy to define using rules. Somebody is a parent if s/he is either a mother or a father:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Prolog examples: rules III

- Rules can call other rules as with `grandparent/2`
- Z is an intermediate variable shared between goals. $\rightarrow$ enables to find the link between the grandparent and the grandchild: a mother or a father

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

- generalize the `grandparent/2` predicate and write `ancestor/2`
- two rules, one of them being recursive

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Prolog examples: rules IV

- recursion pattern is quite common for Prolog rules
- one or more rules express a general case using recursion
- another set of rules or facts describes simpler conditions without recursion $\rightarrow$ correspond to boundary cases and enable the recursion to terminate

Prolog clauses

Facts and rules are also called *clauses*

Prolog Program

Prolog program

program a sequence of Prolog clauses

interpreted as a *conjunction* of clauses

logic theory constituting a *logic theory* made of Horn clauses written according the Prolog syntax

Prolog Execution I

Aim of a Prolog computation

- given a Prolog program P and the goal `?- p(t1,t2,...,tm)` (also called *query*)
- if `X1,X2,...,Xn` are the variables in terms `t1,t2,...,tm`
- the meaning of the goal is to query P and find whether there are some values for `X1,X2,...,Xn` that make `p(t1,t2,...,tm)` true

→ thus, the aim of the Prolog computation is to find a substitution $\sigma =$`X1/s1,X2/s2,...,Xn/sn` such that $P \models \texttt{p}(\texttt{t1}, \texttt{t2}, \ldots, \texttt{tm})\sigma$

Prolog Execution II

Prolog search strategy

- as a logic programming language, Prolog adopts SLD resolution
- as a search strategy, Prolog applies resolution in a strictly linear fashion
  - *goals* are replaced *left-to-right*, sequentially
  - *clauses* are considered in *top-to-bottom* order
  - *subgoals* are considered *immediately* once set up

→ resulting in a *depth-first* search strategy

Prolog Execution III

Prolog backtracking

- in order to achieve completeness, Prolog saves choicepoints for any possible alternative still to be explored
- and goes back to the nearest choice point available in case of failure
- exploiting automatic backtracking

Example 1: The parent.pl Knowledge Base I

```
parent(joey,luca).
parent(joey,simone).
parent(lino,joey).
parent(mirella,joey).
```

A logic theory

- a simple logic *program*
- with four *ground facts*
- representing one sort of relation between elements of the *domain of discourse*

? is there anything we can do with this program?

?? can we *compute* anything?

Example 1: The `parent.pl` Knowledge Base II

Constants & predicates

- `joey`, `luca`, `simone`, `lino` and `mirella` are *constant* used in the program as *ground terms* to denote the element of the domain
- `parent` is the *predicate* used in the program to talk about the domain of discourse—`parent/2` says that `parent` is the *predicate symbol* with *arity* 2

Example 1: The `parent.pl` Knowledge Base III

Goals

- since the only predicate in the program is `parent/2`, we cannot prove anything else, in principle—except for tautologies, or built-in Prolog predicates
- possible goals
  1. `:- parent(joey,luca).`
  2. `:- parent(joey,lino).`
  3. `:- parent(joey,X).`
  4. `:- parent(X,joey).`
  5. `:- parent(X,Y).`

Let us try the above *queries* in tuProlog

tuProlog in Short I

What is tuProlog?

- tuProlog is a light-weight Prolog system for distributed applications and infrastructures [Denti et al., 2001]
- intentionally designed around a minimal core
- to be either statically or dynamically *configured* by loading/unloading libraries of predicates
- tuProlog natively supports multi-paradigm programming [Denti et al., 2005], providing a clean, seamless integration model between Prolog and mainstream object-oriented languages

tuProlog in Short II

Where is tuProlog?

UniBo `http://tuprolog.unibo.it`

Documentation `http://pika-lab.gitlab.io/tuprolog/2p-in-kotlin/`

Playground `https://pika-lab.gitlab.io/tuprolog/2p-kt-web/`

tuProlog in Short III

What to download?

From https://github.com/tuProlog/2p-kt/releases/tag/0.15.2

v. 0.15.0

ide-distro https://github.com/tuProlog/2p-kt/releases/download/0.15.2/2p-ide-0.15.2-redist.jar

distro https://github.com/tuProlog/2p-kt/releases/download/0.15.2/2p-repl-0.15.2-redist.jar

guide https://pika-lab.gitlab.io/tuprolog/2p-in-kotlin/wiki/Users%20Guide/

Let us try the above *queries* in tuProlog

Prolog Interactive Programming in tuProlog I

tuProlog Playground

- learning environment
- no need to install software
- limited functionalities

Prolog Interactive Programming in tuProlog II

- set a simple thoery

```
parent(joey,luca).
parent(joey,simone).
parent(lino,joey).
parent(mirella,joey).
```

- then try the following queries:

  1. `?- parent(joey,luca).`
  2. `?- parent(joey,lino).`
  3. `?- parent(joey,X).`
  4. `?- parent(X,joey).`
  5. `?- parent(X,Y).`

  and see what happens, by responding with

  - ⟨next⟩ to have more answers
  - `X` to delete the query's answers

Prolog Interactive Programming in tuProlog III

Remarks on interaction

1. success
2. failure
3. computed substitution
4. unification
5. backtracking

all no input / output parameters: no *direction* required for arguments in principle thanks to unification

Example 2: `grandparent.pl` I

Adding a rule

- add the rule

  ```
  grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
  ```

  to the knowledge base
- now the logic program is a collection of *facts* and *rules*

! it is a so-called *universal rule*

Example 2: grandparent.pl II

Test the program

- now try the following queries:
  1. `?- grandparent(lino,luca).`
  2. `?- grandparent(lino,joey).`
  3. `?- grandparent(joey,X).`
  4. `?- grandparent(lino,X).`
  5. `?- grandparent(X,Y).`

  and discuss all the results

Example 3: `sibling.pl` I

Adding another rule

- add the rule
  ```
  sibling(Y,Z) :- parent(X,Y), parent(X,Z), Y\=Z.
  ```
  to the previous logic theory
- all the previous theorems are true: all previous computations are the same
- just adding new theorems based on a new rule

! operator \=/2 represent an *explicit* computation over terms
  - succeeding when the two arguments are terms that do not unify
  - all other computations over terms till now were *implicitly* driven by goal unification

Example 3: sibling.pl II

Test the program

- now try the following queries:
  1. `?- sibling(simone,luca).`
  2. `?- sibling(lino,joey).`
  3. `?- sibling(luca,X).`
  4. `?- sibling(lino,X).`
  5. `?- sibling(X,Y).`

  and discuss all the results

Exercise 1 I

For below english sentences write applicable Prolog facts, rules & goals.

1. Maria reads logic programming book by author Peter Lucas.
2. Anyone likes shopping if she is a girl.
3. Who likes shopping?
4. Kirke hates any city if it is big and crowdy.

Exercise 1 II

1. Maria reads logic programming book by author Peter Lucas.
   ```
   read(maria, book(author('Peter', 'Lucas'), lp)).
   ```
2. Anyone likes shopping if she is a girl.
   ```
   like(shopping, X) :- girl(X).
   ```
3. Who likes shopping?
   ```
   ?- like(shopping, X).
   ```
4. Kirke hates any city if it is big and crowdy.
   ```
   hate(X, kirke) :- city(X), big(X), crowdy(X).
   ```

Exercise 2 I

Assume given a set of facts of the form `father(name1,name2)` (name1 is the father of name2).

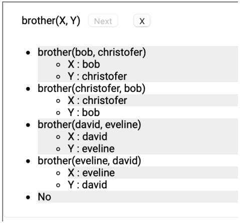
```
father(julian,bob).
father(julian,christofer).
father(bob,david).
father(bob,eveline).
father(christofer,felix).
```

- Define predicate `brother(X,Y)` which holds iff `X` and `Y` are brothers
- Define predicate `cousin(X,Y)` which holds iff `X` and `Y` are cousins
- Define predicate `grandson(X,Y)` which holds iff `X` is a grandson of `Y`
- Define predicate `descendent(X,Y)` which holds iff `X` is a descendents of `Y`

Exercise 2 II

- Define a predicate `brother(X,Y)` which holds iff `X` and `Y` are brothers

```
brother(X,Y) :- father(Z,X), father(Z,Y), not(X=Y).
```



Exercise 2 III

- Define predicate `cousin(X,Y)` which holds iff `X` and `Y` are cousins

```
cousin(X,Y) :- father(Z,X), father(W,Y), brother(Z,W).
```

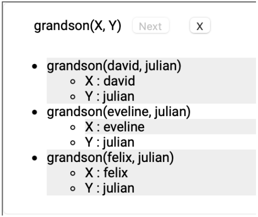
· cousin(X, Y) Next X

- cousin(david, felix)
  - X : david
  - Y : felix
- cousin(eveline, felix)
  - X : eveline
  - Y : felix
- cousin(felix, david)
  - X : felix
  - Y : david
- cousin(felix, eveline)
  - X : felix
  - Y : eveline
- No

Exercise 2 IV

- Define predicate `grandson(X,Y)` which holds iff `X` is a grandson of `Y`

```
grandson(X,Y) :- father(Z,X), father(Y,Z).
```



Exercise 2 V

- Define predicate `descendent(X,Y)` which holds iff `X` is a descendents of `Y`

```
descendent(X,Y) :- father(Y,X).

descendent(X,Y) :- father(Z,X), descendent(Z,Y).
```

Exercise 3 I

A Simple Thought (Basic Inference)

People wish to live in peace. Men, women and children are people. I am a woman (or a man). Therefore I wish to live in peace.

1. Use Prolog to prove this statement!

Exercise 3 II

```
wish_to_live_in_peace(X) :- people(X).
people(X) :- man(X).
people(X) :- woman(X).
people( X) :- child(X).
woman(me).
```

- Goal to prove:

```
?-wish_to_live_in_peace(me). ==> true
```

Lists in Prolog I

Prolog lists

Lists are data structures essential to many programs. A Prolog list is a sequence of an arbitrary number of terms separated by commas and enclosed within square brackets.

For example:

- `[a]` is a list made of an atom
- `[a, b]` is a list made of two atoms
- `[a, X, father(X, telemachus)]` is a list made of an atom, a variable, and a compound term
- `[[a, b], [[[father(X, telemachus)]]]]` is a list made of two sublists
- `[]` is the atom representing the empty list

Lists in Prolog II

Lists in general

- list are defined via two constructors

  nil the empty list, containing no elements

  cons the constructor *cons*, taking an element H and a list T, and generating the list $cons(H, T)$

e.g. $cons(a, cons(b, cons(c, nil)))$ would represent list a, b, c

- typical *recursive data structures*
- used to represent *sequences* of any sort

Lists in Prolog III

Prolog lists

- compound terms and the square bracketed notation is only a shortcut $\rightarrow$ list functor is a dot: `./2`
- in Prolog, list are defined via two analogous constructors
  - `[]` represents the empty list, containing no elements—a *constant*
  - `.` stands for *cons*, taking an element `H` and a list `T`, and generating the list `.(H,T)`—a *functor* of arity 2
- Prolog *sequence notation* simplifies writing lists
  - `.(H,T)` can be written as `[H|T]`
  - `.(H,.(H',T'))` can be written as `[H,H'|T']`
  - there, empty list can be omitted

e.g. `[a,b,c]` would represent list a, b, c in Prolog, where
  - a is the head of the list
  - `[b,c]` is the tail of the list
  - $mgu([\mathtt{a}, \mathtt{b}, \mathtt{c}], [\mathtt{H}|\mathtt{T}]) = \{\mathtt{H}/\mathtt{a}, \mathtt{T}/[\mathtt{b}, \mathtt{c}]\}$

Computing with Lists I

Recursion

- being recursive data structures, lists are typically handled by recursive rules
- which incidentally is also the *only* way to handle repeated operations over sequences in Prolog, where there is nothing like a *cycle* programming construct

Recursion scheme in Prolog

- since Prolog search strategy is depth-first
- in particular, with clauses used orderly, top-down
- *termination* is handled with a fact, typically coming *before* the recursive rule
- as already seen in the cases of `num/1` and `plus/3` above

Typical Example: member I

member/2

Checking whether the first argument is a term that is a member of the list in the second argument

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :-  member(X,Ys).
```

- goals
  1. `?- member(b,[a,b,c])`
  2. `?- member(X,[a,b,c]).`
  3. `?- member(g(X),[f(a),g(b),f(c),g(d)]).`
  4. `?- member(z,[X|T]).`

Typical Example: `member` II

- remarks
  - search strategy: left to right through the list
  - devising out all the members of the list
  - conditional membership—given a certain computed substitution
  - generation of lists

Prolog cut predicate I

Cut predicate "!"

- device to prune some backtracking alternatives
- $\rightarrow$ the right rule has been found, no further attempts must be made
- $\rightarrow$ avoid unnecessary computations
- modifies the way Prolog explores goals and enables a programmer to control the execution of programs
- when executed in the body of a clause, the cut always succeeds and removes backtracking points set before it in the current clause

Prolog cut predicate II

Let us suppose that a predicate P consists of three clauses:

$$P :- A_1, ..., A_i, !, A_{i+1} ..., An.$$

$$P :- B_1, ..., B_m.$$

$$P :- C_1, ..., C_p.$$

Executing the cut in the first clause has the following consequences:

1. all other clauses of the predicate below the clause containing the cut are pruned $\rightarrow$ the two remaining clauses of P will not be tried
2. all the goals to the left of the cut are also pruned $\rightarrow$ $A_1, ..., A_i$ will no longer be tried
3. however, it will be possible to backtrack on goals to the right of the cut

$P :-$ ~~$A_1, ..., A_i$~~$, !, A_{i+1} ..., An.$

~~$P :- B_1, ..., B_m.$~~

~~$P :- C_1, ..., C_p.$~~

Prolog cut predicate III

Cut to express determinism

Deterministic predicates always produce a definite solution; it is not necessary then to maintain backtracking possibilities.

A simple example of it is given by the minimum of two numbers:

```
minimum(X, Y, X) :- X < Y.
minimum(X, Y, Y) :- X >= Y.
```

Once the comparison is done, there is no means to backtrack because both clauses are mutually exclusive. This can be expressed by adding two cuts:

```
minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y) :- X >= Y, !.
```

Prolog cut predicate IV

Some programmers would rewrite `minimum/3` using a single cut:

```
minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y).
```

- once Prolog has compared `X` and `Y` in the first clause, it is not necessary to compare them again in the second one.
- latter program may be more efficient in terms of speed, BUT it is obscure
- first version cuts respect the logical meaning of the program and do not impair its legibility

→ green cuts

- second predicate is to avoid writing a condition explicitly: error-prone

→ red cuts

Sometimes red cuts are crucial to a program but when overused, they are a bad programming practice.

Negation I

Negation as failure

A logic program contains no negative information, only queries that can be proven or not. The Prolog built-in negation corresponds to a query failure: the program cannot prove the query.

- negation symbol "\+"
- If *G* succeeds then \+ *G* fails
- If *G* fails then \+ *G* succeeds

The Prolog negation is defined using a cut:

```
\+ (P) :- P, !, fail.
\+ (P) :- true.
```

where `fail/0` is a built-in predicate that always fails

Negation II

Most of the time, it is preferable to ensure that a negated goal is ground: all its variables are instantiated. Let us illustrate it with the somewhat odd rule:

```
mother(X, Y) :- \+ male(X), child(Y, X).
```

and facts:

```
child(telemachus, penelope).
male(ulysses).
male(telemachus).
```

query ?- mother(X, Y).

fails because the subgoal male(X) is not ground and unifies with the fact male(ulysses).

Negation III

If the subgoals are inverted:

```
mother(X, Y) :- child(Y, X), \+ male(X).
```

query ?- mother(X, Y).

succeeds. Because the term child(Y, X) unifies with the substitution X = penelope and Y = telemachus, and since male(penelope) is not in the kb, the goal succeeds.

Missing

Many interesting things still missing

- that are relevant for Prolog programming
  - operator definition
  - conditionals
  - closed world assumption (CWA)
  - arithmetic
  - meta programming
- and many more

! however, this is not a Prolog course

- and we already discussed whatever could be useful for our purposes

References I

[Act, 2021] Act, A. I. (2021).
Proposal for a regulation of the european parliament and the council laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts.
EUR-Lex-52021PC0206.

[Apt, 2005] Apt, K. R. (2005).
The logic programming paradigm and Prolog.
In Mitchell, J. C., editor, *Concepts in Programming Languages*, chapter 15, pages 475–508. Cambridge University Press, Cambridge, UK
http://www.cambridge.org/academic/subjects/computer-science/programming-languages-and-applied-logic/concepts-programming-languages?format=AR.

[Borning et al., 1989] Borning, A., Maher, M. J., Martindale, A., and Wilson, M. (1989).
Constraint hierarchies and logic programming.
In Levi, G. and Martelli, M., editors, *Sixth International Conference on Logic Programming*, volume 89, pages 149–164, Lisbon, Portugal. MIT Press.

References II

[Buchanan and Shortliffe, 1984] Buchanan, B. G. and Shortliffe, E. H. (1984).
Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.
The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc.
http://tocs.ulb.tu-darmstadt.de/31349005.pdf.

[Calegari et al., 2018a] Calegari, R., Denti, E., Dovier, A., and Omicini, A. (2018a).
Extending logic programming with labelled variables: Model and semantics.
Fundamenta Informaticae, 161(1-2):53–74.
Special Issue on the 31th Italian Conference on Computational Logic: CILC 2016
DOI:10.3233/FI-2018-1695.

[Calegari et al., 2018b] Calegari, R., Denti, E., Mariani, S., and Omicini, A. (2018b).
Logic programming as a service.
Theory and Practice of Logic Programming, 18(5-6):846–873.
Special Issue "Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming"
DOI:10.1017/S1471068418000364.

References III

[Colmerauer and Roussel, 1996] Colmerauer, A. and Roussel, P. (1996).
The birth of Prolog.
In Bergin Jr., T. J. and Gibson Jr., R. G., editors, *History of programming languages—II*, volume II, pages 331–367. ACM, New York, NY, USA
DOI:10.1145/234286.1057820.

[Console et al., 1997] Console, L., Lamma, E., Mello, P., and Milano, M. (1997).
Programmazione logica e Prolog.
UTET Libreria
http://www.utetuniversita.it/catalogo/scienze/programmazione-logica-e-prolog-1824.

[Denti et al., 2001] Denti, E., Omicini, A., and Ricci, A. (2001).
tuProlog: A light-weight Prolog for Internet applications and infrastructures.
In Ramakrishnan, I., editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg.
3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings
DOI:10.1007/3-540-45241-9$_1$3.

References IV

[Denti et al., 2005] Denti, E., Omicini, A., and Ricci, A. (2005).
Multi-paradigm Java-Prolog integration in tuProlog.
Science of Computer Programming, 57(2):217–250
DOI:doi:10.1016/j.scico.2005.02.001.

[Dyckhoff et al., 1996] Dyckhoff, R., Herre, H., and Schroeder-Heister, P., editors (1996).
Extensions of Logic Programming, 5th International Workshop, ELP'96, volume 1050 of *Lecture Notes in Computer Science*, Leipzig, Germany. Springer
DOI:10.1007/3-540-60983-0.

[European Commission, 2019] European Commission (2019).
Ethics guidelines for trustworthy AI.
Publications Office
DOI:doi/10.2759/177365.

[Kowalski, 1974] Kowalski, R. A. (1974).
Predicate logic as programming language.
In *Information Processing 74 – Proceedings of the 1974 IFIP Congress*, pages 569–574.
North-Holland Publishing Company
http://dblp.uni-trier.de/rec/html/conf/ifip/Kowalski74.

References V

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982).
An efficient unification algorithm.
ACM Transactions on Programming Languages and Systems, 4(2):258–282
DOI:10.1145/357162.357169.

[Metakides and Nerode, 1996] Metakides, G. and Nerode, A. (1996).
Principles of Logic and Logic Programming, volume 13 of *Studies in Computer Science and Artificial Intelligence*.
Elsevier
http://www.sciencedirect.com/science/bookseries/09243542/13.

[Nilsson and Maluszynski, 1995] Nilsson, U. and Maluszynski, J. (1995).
Logic, Programming and Prolog.
John Wiley & Sons, Inc., New York, NY, USA.

[Rijk, 2002] Rijk, L. M. D. (2002).
Aristotle: Semantics and Ontology. Volume I: General Introduction. The Works on Logic, volume 91 of *Philosophia Antiqua*.
Brill Academic Publishers.

References VI

[Robinson, 1965] Robinson, J. A. (1965).
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1):23–41
DOI:10.1145/321250.321253.

[Saptawijaya and Pereira, 2019] Saptawijaya, A. and Pereira, L. M. (2019).
From logic programming to machine ethics.
In Bendel, O., editor, *Handbuch Maschinenethik*, pages 209–227. Springer VS, Wiesbaden
DOI:10.1007/978-3-658-17483-5_14.

Programmazione Logica e Rudimenti di Prolog

Advanced School in AI in Emilia Romagna

Roberta Calegari

roberta.calegari@unibo.it

Alma Mater Studiorum – Università di Bologna

24 July 2023