Prolog Course at ENAC

Jérôme Mengin et Jan-Georg Smaus

Year 2018/19

Logic program examples and references

An excerpt from a movie database

(1) release('The Blues Brothers', france, 1980).
(2) release('Soul Kitchen', germany, 2009).
(3) release('Soul Kitchen', france, 2010).
(4) release('Das Leben der Anderen', germany, 2006).
(5) release('Das Leben der Anderen', france, 2007).
(6) director('John Landis', 'The Blues Brothers').
(7) director('Fatih Akin', 'Soul Kitchen').
(8) cast('The Blues Brothers', 'Dan Aykroyd', "'Joliet" Jake Blues').
(9) cast('The Blues Brothers', 'Aretha Franklin', 'Mrs. Murphy').
(10) cast('Soul Kitchen', 'Adam Bousdoukos', 'Zinos Kazantsakis').
(11) cast('Soul Kitchen', 'Moritz Bleibtreu', 'Illias Kazantsakis').
(12) cast('Soul Kitchen', 'Anna Bederke', 'Lucia Faust').
(13) cast('Das Leben der Anderen', 'Martina Gedeck', 'Christa-Maria Sieland').

An excerpt from a movie database

(1) release('The Blues Brothers', france, 1980).
(2) release('Soul Kitchen', germany, 2009).
(3) release('Soul Kitchen', france, 2010).
(4) release('Das Leben der Anderen', germany, 2006).
(5) release('Das Leben der Anderen', france, 2007).
(6) director('John Landis', 'The Blues Brothers').
(7) director('Fatih Akin', 'Soul Kitchen').
(8) cast('The Blues Brothers', 'Dan Aykroyd', "'Joliet" Jake Blues').
(9) cast('The Blues Brothers', 'Aretha Franklin', 'Mrs. Murphy').
(10) cast('Soul Kitchen', 'Adam Bousdoukos', 'Zinos Kazantsakis').
(11) cast('Soul Kitchen', 'Moritz Bleibtreu', 'Illias Kazantsakis').
(12) cast('Soul Kitchen', 'Anna Bederke', 'Lucia Faust').
(13) cast('Das Leben der Anderen', 'Martina Gedeck', 'Christa-Maria Sieland').

Queries A *query* to find the year of the release of Soul Kitchen: release('Soul Kitchen', C, Y)?
$\Rightarrow$ Prolog's answer: $C = \text{germany} \wedge Y = 2009 \vee C = \text{france} \wedge Y = 2010$.

An excerpt from a movie database

$\Rightarrow$ Write queries (and give the answers) to find the following :

1. the director of The Blues Brothers;
2. if Aretha Franklin played in a film by John Landis;
3. actors of movies by Fatih Akin;
4. the directors of movies in which Dan Aykroyd and Anna Bederke were co-stars;
5. if Anna Bederke played in a movie by John Landis or Fatih Akin;
6. actors who are also directors?
7. actors who played with Dan Aykroyd.

An excerpt from a movie database

Note on logical syntax:

- $A \leftarrow B$ is read: "A is true if B is true" (logical implication)
- $B \wedge C$ is true if both B and C are true (logical conjunction)
- $B \vee C$ is true if at least one of B or C are true (logical disjunction)

Rules A *rule* to define a relation directed: $\mathsf{directed}(D, A)$ is true if A played in a movie directed by D:

(16) $\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R).$

An excerpt from a movie database

Note on logical syntax:

- $A \leftarrow B$ is read: "A is true if B is true" (logical implication)
- $B \wedge C$ is true if both B and C are true (logical conjunction)
- $B \vee C$ is true if at least one of B or C are true (logical disjunction)

Rules A *rule* to define a relation directed: $\mathsf{directed}(D, A)$ is true if A played in a movie directed by D:

(16) $\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R).$

$\Rightarrow$ Add rules to the database to define the following:

1. $\mathsf{co\_star}(A1, A2)$ – the actor/actress played in the same movie
2. the films in which played some actor who played in a film by Fatih Akin or John Landis

A short history of logic programming

1970s: Kowalski (Edinburgh):
the logical formula $\varphi \leftarrow \psi_1 \wedge \ldots \wedge \psi_n$ (φ is true if all the ψ_is are) has a procedural meaning: "In order to prove φ, it is sufficient to prove $\psi_1, \ldots, \psi_n$".
Colmerauer (Aix-Marseille): Prolog,
theorem prover based on the same ideas as Kowalski.

A short history of logic programming

1970s: Kowalski (Edinburgh):
the logical formula $\varphi \leftarrow \psi_1 \wedge \ldots \wedge \psi_n$ (φ is true if all the ψ_is are) has a procedural meaning: "In order to prove φ, it is sufficient to prove $\psi_1, \ldots, \psi_n$".
Colmerauer (Aix-Marseille): Prolog,
theorem prover based on the same ideas as Kowalski.

end of the 70s: Warren (Edinburgh): Prolog-10,
a fast Prolog implementation; the underlying ideas still are at the core of numerous recent implementations.

A short history of logic programming

begining of 21st century:

- a standard Prolog language (syntax "of Edinburgh");
- far from the ideal of logic programming (a small subset of classical logic);
- **extension to constraints programming**
- numerous implementations, some are open source or free, some have good IDE...;
- interface Prolog/other langages (C, Java,...).
- Prolog widely used, eg.:
  - Prolog Development Center: airport scheduling (teams, runways, shopfloor,...), environmental disaster management,...
  - RDF analysis (Resource Description Framework, W3C)
  - youbet.com: analysis of information coming from a number of webpages, rules easy to maintain when these pages are modified

A few books

To start with:

Learn Prolog Now! Patrick Blackburn, Johan Bos and Kristina Striegnitz. College Publications, 2006.
On-line version, with lecture slides: `learnprolognow.org`

The Art of Prolog. Leon Sterling and Ehud Shapiro.
MIT Press, 1999 (3rd edition).
$\Rightarrow$ very logical approach (available in French)

Prolog : Programming for Artificial Intelligence. Ivan Bratko.
Addison Wesley, 2001 (3rd edition).
$\Rightarrow$ more computer science oriented (available in French)

More advanced topics:

Programming in Prolog William Clocksin and Christopher Mellish.
Springer, 1987.

The Craft of Prolog Richard O'Keefe. MIT Press, 1990.

First steps in Prolog

How to start Prolog

The GNU Prolog system is documented at `gprolog.org`. That website has a manual. Gnu Prolog is free, and there are binaries available for most operating systems. GNU Prolog can be used in interactive mode, like a command interpreter (shell): the user types in queries, called *goals*, and the system replies with solutions to the queries, and prompts the user for a new query.

This interactive system must be started from a terminal using a Unix command interpreter (sh, csh, ksh,...); the command to start Gnu Prolog is `gprolog`.

How to start Prolog

Programs are written in files, using standard text editors. The name of the program files must end with .pl. The predicate consult is used to load programs: consult('my_prog.pl'). There are a few shortcuts for this type of very frequent goals: consult(my_prog). or even [my_prog]. or ['my_prog.pl']. This shows two things:

- the ' around the file name are needed because the name contains some special characters (the "." here);
- every query is terminated with a ".".

After a query has been submitted, the Prolog system will try to compute a first answer, that may be yes, no, or a list of values for the variables that appear in the query: in this case, to obtain more solutions, one must type in a semicolon.

Exiting Prolog The query halt. terminates the prolog interpreter.

How to start Prolog

Exercise 1 This exercise uses the "movie" database[1], you should download it, have a look at its content with a text editor and "consult" it with prolog.

Question 1.1 Use prolog to find actors and actresses who have:

- been directed by Brian de Palma;
- been directed by Tim Burton and also by Francis Ford Coppola;
- played in at least two different movies by Sofia Coppola;
- been directed by Tim Burton and not by Francis Ford Coppola;
- played in exactly two different movies by Sofia Coppola.

[1] www.irit.fr/~Jerome.Mengin/teaching/prolog/movie_bd.pl

How to start Prolog

If you are familiar with first-order logic, here is a comparison of the traditional logic syntax and Prolog:

Syntax of Edinburgh / GNU Prolog
in order of increasing priorities:

Connective	Meaning	Logic	Prolog
implication	"if"	$\leftarrow$	:-
disjunction	"or"	$\vee$	;
conjunction	"and"	$\wedge$	,
negation	"not"	$\neg$	\+
equality	"equals"	$=$	=
inequality	"different"	$\neq$	\=

Remark Negation is a subtle issue in Prolog and will be discussed later.

Syntax of Prolog programms

Basic syntactical constructs

atom: a name used to name *relations* and *term constructors* (including term *constants*); it can be

- a sequence of letters, digits that starts with a lowercase letter, and can also contain the underscore "_"
- a sequence of characters enclosed between two single quotes

Examples: cast(predicate) 'The Blues Brothers' germany (term constants)

variable: string of letters, digits, "_" that starts with an uppercase letter, or with "_"

Examples: X1 Toto12_urt___cur4 _123urc_

numerical constants: usual representations for (signed) integers and floating point numbers

Composed syntactical constructs

term: an expression that represents a *data object*:

term constants, numerical constants, variables

+ compound terms of the form $f(t_1, \ldots, t_n)$

where f is an atom (term constructor) and t_1, ..., t_n are terms

(Lists are a special kind of compound terms.)

goal: expression that can be *true* of *false*;

has the form $p(t_1, \ldots, t_n)$

where p is a relation (predicate) and t_1, ..., t_n are terms.

Logical constructs

formula: *goals* assembled with connectives
$\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation).

rule: $\underbrace{G}_{\text{head}} \leftarrow \underbrace{\varphi}_{\text{body}}.$

where G is a goal and φ is a formula.

fact: rule with $\varphi = \top$ (always true); writtten G.

clause: rule of the form $G \leftarrow L_1 \wedge \ldots \wedge L_m$
where $L_1, \ldots L_m$ are *literals*,
that is, goals and negated (discussed later) goals. (Thus a Prolog clause does not contain any disjunction.)

Logical constructs : Quantification

$\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R)$. is read:
"*for all* D, A, D has directed A if *there exists* some M, the director of which is D, and in which A played"
In logic, we would write:
$\forall D, A(\mathsf{directed}(D, A) \leftarrow \exists M(\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R)))$

- The variables that appear in the head of a rule have an implicit *universal* quantification / meaning.
  It is understood that the rule is true for all possible values of these variables.
- The variables that appear only in the body of a rule have an implicit *existential* quantification/meaning within the body of the rule.
  It is understood that the head of the rule is true if there is at least one value for each of these variables for which the body of the rule is true.

This is not an ad-hoc interpretation of logic programming but has a clear logical explanation ...

Logical constructs : Quantification

Caveat: We are still looking at logic programs without negation, but in the following *logical transformation*, negation comes into play: the clause (with only positive literals!)

$$\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R).$$

is a-priori quantified universally:

$$\forall D, A, M, R \; (\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R))$$

but

Logical constructs : Quantification

Caveat: We are still looking at logic programs without negation, but in the following *logical transformation*, negation comes into play: the clause (with only positive literals!)

$$\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R).$$

is a-priori quantified universally:

$$\forall D, A, M, R\ (\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R))$$

but

$$\begin{aligned}
&\forall D, A, M, R\ (\mathsf{directed}(D, A) \leftarrow \mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R)) \equiv \\
&\forall D, A, M, R\ (\mathsf{directed}(D, A) \vee \neg(\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R))) \equiv \\
&\forall D, A\ (\mathsf{directed}(D, A) \vee \forall M, R\ \neg(\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R))) \equiv \\
&\forall D, A\ (\mathsf{directed}(D, A) \vee \neg\exists M, R\ (\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R))) \equiv \\
&\forall D, A\ (\mathsf{directed}(D, A) \vee \leftarrow \exists M, R\ (\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R)))
\end{aligned}$$

Predicates and programs

predicate (or relation): characterized by it *name* and its *arity* (number of arguments);
the predicate of name p and arity n, denoted p/n, is defined by a set of *rules / facts* of the form:
$p(t_1, \ldots, t_n) \leftarrow \varphi$ or $p(t_1, \ldots, t_n)$.
Predicates are to Prolog what functions / procedures are to more conventional programming languages.

Predicates and programs

predicate (or relation): characterized by it *name* and its *arity* (number of arguments);
the predicate of name p and arity n, denoted p/n, is defined by a set of *rules / facts* of the form:
$p(t_1, \ldots, t_n) \leftarrow \varphi$ or $p(t_1, \ldots, t_n)$.
Predicates are to Prolog what functions / procedures are to more conventional programming languages.

Remark Any rule is equivalent to a set of clauses
because of properties of $\neg$, $\wedge$, $\vee$ (Boolean algebra), and because:
$\psi \leftarrow \varphi_1 \vee \varphi_2$ is equivalent to $\left\{ \begin{array}{l} \psi \leftarrow \varphi_1 \\ \psi \leftarrow \varphi_2 \end{array} \right\}$ and
negated formulas in bodies can also be compiled away (see later).

Predicates and programs

logic program: a set of definitions of predicates.

> **Remark** Clauses or rules that define a predicate p/n must not be interleaved with rules or clauses that define other predicates.
> (If the definition of predicate p is scattered at different places in a file, Prolog considers that they are successive definitions of the predicate p, each definition canceling the previous one.)

query: a formula.

Declarative semantics of pure logic programming:
Given a logic program $\mathbf{P}$ and a query φ, let U be the vector of all the variables that appear in φ, we want to know what are the values of U for which $\mathbf{P} \models \varphi$.
(Due to negation and some other issues, the reality deviates from this.)

Lists

Observations:

- The introduction to the syntax above suggests that atoms used as term constants, term constructors and predicate symbols are completely user-defined.
- We have not seen any term contructors yet.
- If the number of atoms is finite, one needs term constructors if one wants to generate an arbitrary number of terms. Example: father(. . . father('Moritz Bleibtreu') . . .)

Lists

Observations:

- The introduction to the syntax above suggests that atoms used as term constants, term constructors and predicate symbols are completely user-defined.
- We have not seen any term contructors yet.
- If the number of atoms is finite, one needs term constructors if one wants to generate an arbitrary number of terms. Example: father(. . . father('Moritz Bleibtreu') . . .)

Linked lists are the simplest example of a *recursive* datastructure, allowing for arbitrarily big terms. They are widely used in functional and logic programming.

A list can store any number of data objects.

Lists

The basic syntax for lists uses a *constant* nil and a *term constructor* cons. Inductive definition:

- nil is the empty list.
- If l is a list and h is an arbitrary term, then $\mathsf{cons}(h, l)$ is a list. We call h the *head* and l the *tail* of the list $\mathsf{cons}(h, l)$.

Example: $\mathsf{cons}(3, \mathsf{cons}(2, \mathsf{cons}(4, \mathsf{cons}(1, \mathsf{nil}))))$

Lists

Since this notation is cumbersome, some syntactic sugar is introduced: $\mathsf{cons}(h, l)$ is written $[h \mid l]$, and $[h_1 \mid [h_2 \mid l]]$ can be simplified to $[h_1, h_2 \mid l]]$ (recursively).

Examples of prolog lists: $[1, 2, 3, [-1, \mathsf{a}, [\,]], \text{'movie\_bd'}]$ $[\,]$ $[\_, X, Y, 1]$

Lists

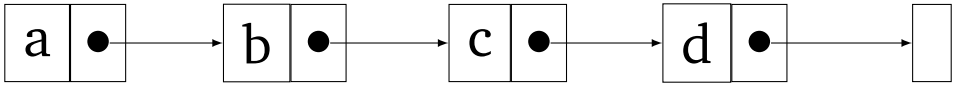
Since this notation is cumbersome, some syntactic sugar is introduced: $\mathsf{cons}(h, l)$ is written $[h \mid l]$, and $[h_1 \mid [h_2 \mid l]]$ can be simplified to $[h_1, h_2 \mid l]]$ (recursively).

Examples of prolog lists: $[1, 2, 3, [-1, \mathsf{a}, [\,]], \mathsf{'movie\_bd'}]$ $[\,]$ $[\_, X, Y, 1]$

To summarise:

- A list is enclosed in squared brackets
- The elements are separated by commas ","
- Elements of all types can be put in a list
- A list can contain other lists
- Character strings are lists:
  Prolog interprets `"abcd"` as the list of ASCII codes $[97, 98, 99, 100]$.

Lists

A list: 

Prolog representation: $[\, a \mid [\, b \mid [\, c \mid d \mid [\ ]\,]\,]\,]$

For instance, $[1,2,3,4] = [1,2,3|\,[4]\,] = [1|\,[2|\,[3|\,[4|\,[\,]\,]\,]\,]\,]$.

(But $[1,2,3,4] \neq [\,[1,2]\mid[3,4]\,]$. Why ?)

Lists

Examples of filtering with the recursive form of lists:

- isFirstElmtOf(X, L): true if X is the first element of the list $L \Rightarrow$

$$\mathsf{isFirstElmtOf}(X, L) \leftarrow L = [X|R].$$

  or simply

$$\mathsf{isFirstElmtOf}(X, [X|R]).$$

- isSecondElmtOf(X, L) ...

Procedureal semantics of Prolog programms

Unification

Logic programming emerged from the *procedural interpretation* of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Unification

Logic programming emerged from the *procedural interpretation* of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?

Unification

Logic programming emerged from the *procedural interpretation* of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

Unification

Logic programming emerged from the *procedural interpretation* of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

Q: What if there are terms in the head of the rule? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$

Unification

Logic programming emerged from the *procedural interpretation* of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

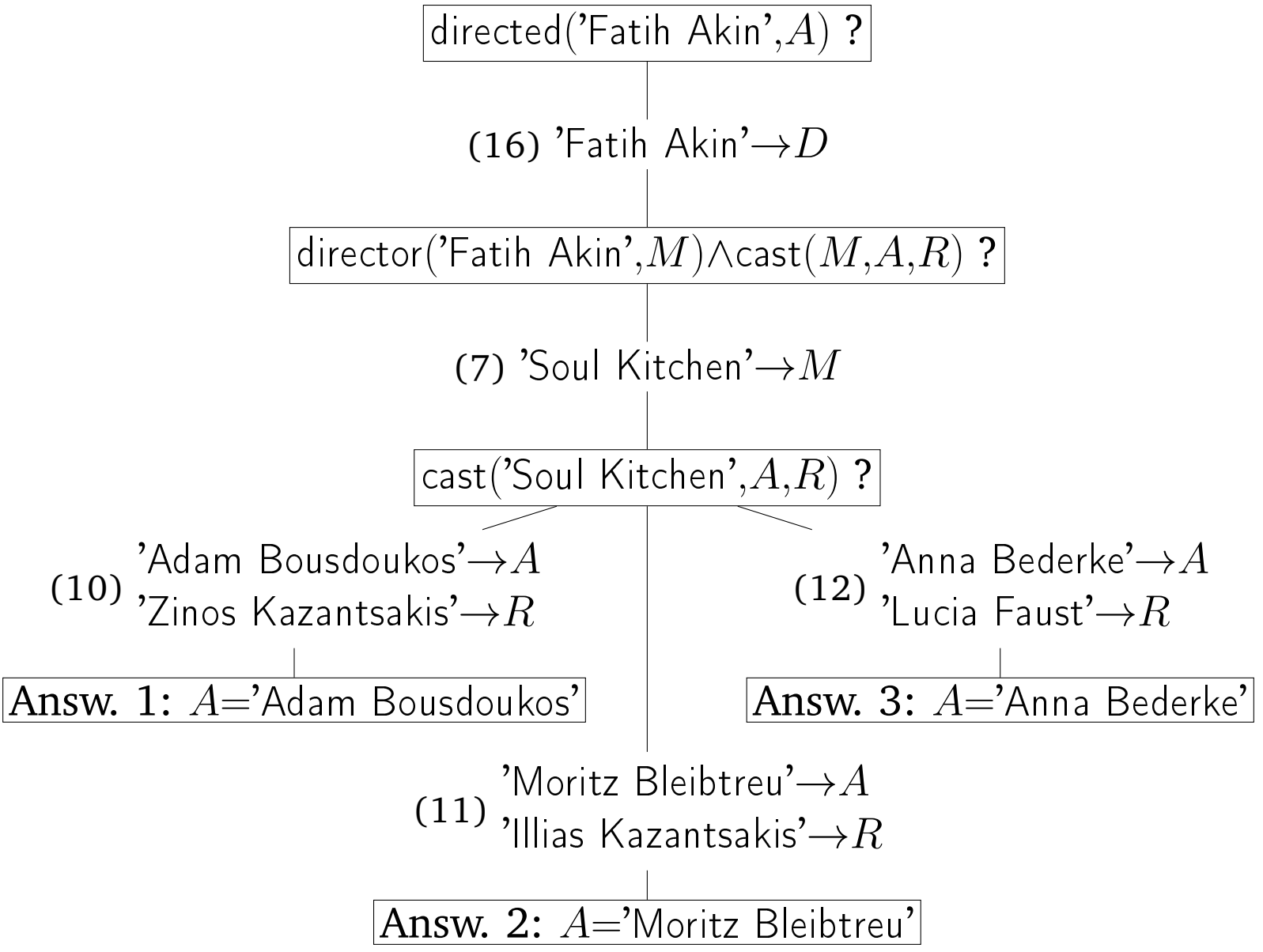
Q: What if there are terms in the head of the rule? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$
A: try to *unify* (match) the u_i's and the t_i's, and make the appropriate substitutions in φ_p

Example $p(X, [X, Y, X])$ can be unified with $p(2, L)$:
$2 \rightarrow X$, $[2, Y, 2] \rightarrow L$.

Unification

Procedural semantics: search tree . . .

Example search tree



Example search tree

- Backward chaining ; goals in a conjunction are *proved / executed* from left to right.
- Executing a goal means:
  - replacing it with a condition that makes it true, or
  - simply deleting it if it appears in the database (possibly after some variable instantiation), or
  - returning “false" if it is not possible to make it true.
- Facts / rules of the program are tried in the order in which they appear in the program.
- Prolog systems use a depth-first search strategy.

Example search tree

- Backward chaining ; goals in a conjunction are *proved / executed* from left to right.
- Executing a goal means:
  - replacing it with a condition that makes it true, or
  - simply deleting it if it appears in the database (possibly after some variable instantiation), or
  - returning "false" if it is not possible to make it true.
- Facts / rules of the program are tried in the order in which they appear in the program.
- Prolog systems use a depth-first search strategy.

Question: how many leaves have the trees for the queries

- $\mathsf{cast}(M, A, S) \wedge \mathsf{cast}(M, \text{'Anna Bederke'}, R)$, and
- $\mathsf{cast}(M, \text{'Anna Bederke'}, R) \wedge \mathsf{cast}(M, A, S)$?

Recursion (on lists)

Recursive programming

We wish to retrieve the last element of a list: $\mathsf{isLastElmtOf}(X, L)$ must be true if X is the last element of L

- the linked list must be scanned until its last element is reached
- we do not know in advance how many steps will be needed
  $\Rightarrow$ *recursive programming:*

Recursive programming : The classical example

We wish to retrieve the last element of a list: $\mathsf{isLastElmtOf}(X, L)$ must be true if X is the last element of L

- the linked list must be scanned until its last element is reached
- we do not know in advance how many steps will be needed
  $\Rightarrow$ *recursive programming:*

X is last element of $[\ Y \mid R\ ]$ if and only if X is last element of R
Termination: X is the last element of $[X]$

In Prolog:
$\mathsf{isLastElmtOf}(X, L) \leftarrow L = [X] \vee (L = [Y|R] \wedge \mathsf{isLastElmtOf}(X, R))$.

Example Write a definition for a predicate $\mathsf{member}/2$, such that $\mathsf{member}(X, L)$ is true if X is element of list L.

Negation

Negation as failure

Procedural meaning given to negation:

In order to prove $\neg\varphi$, try to prove that φ *cannot be proved*.

Consider the movie database excerpt above, and a predicate defining movies in which played actors who were never directed by John Landis:

$p(M) \leftarrow \mathsf{cast}(M, A, R) \wedge \neg\mathsf{directed}(\text{'John Landis'}, A)$.

Draw the search tree for the query $p(\text{'Soul Kitchen'})?$.

Negation as failure

Procedural meaning given to negation:

In order to prove $\neg\varphi$, try to prove that φ *cannot be proved*.

Consider the movie database excerpt above, and a predicate defining movies in which played actors who were never directed by John Landis:

$p(M) \leftarrow \mathsf{cast}(M, A, R) \wedge \neg\mathsf{directed}(\text{'John Landis'}, A).$

Draw the search tree for the query $p(\text{'Soul Kitchen'})?$.

Negation as failure is <u>not</u> logical negation !

The above definition is *logically* equivalent to:

$p(M) \leftarrow \neg\mathsf{directed}(\text{'John Landis'}, A) \wedge \mathsf{cast}(M, A, R).$

But if we submit the query $p(\text{'Soul Kitchen'})?$...

Negation as failure

Write queries (and give the answers) to find the following :

1. actors who played in more than one movie;
2. directors who where never an actor;
3. actors who never played in a movie directed by Fatih Akin.

Negation as failure

Add rules to the database to define the following:

1. the actors who played in at least two films by Woody Allen
2. movies in which played actors who were never directed by John Landis.

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

- for all X, $p(X)$ true if the query $q(Y)$ returns “no”;

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

- for all X, $p(X)$ true if the query $q(Y)$ returns "no";
- that is, for all X, $p(X)$ is true if *there is no* Y for which $q(Y)$ can be proved

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

- for all X, $p(X)$ true if the query $q(Y)$ returns "no";
- that is, for all X, $p(X)$ is true if *there is no* Y for which $q(Y)$ can be proved
- in logic, we would write: $\forall X(p(X) \leftarrow \neg(\exists Y\, q(Y)))$,
  which is equivalent to $\forall X(p(X) \leftarrow \forall Y(\neg q(Y)))$

However, because of the procedural meaning of negation (Prolog starts a new search tree when evaluating a negation), the quantification of the variables that appear within the scope of a negation is not always known at the time where the program is written.

For instance, consider $p(X) \leftarrow r(X, Y) \wedge \neg q(Y)$:

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

- for all X, $p(X)$ true if the query $q(Y)$ returns "no";
- that is, for all X, $p(X)$ is true if *there is no* Y for which $q(Y)$ can be proved
- in logic, we would write: $\forall X(p(X) \leftarrow \neg(\exists Y q(Y)))$,
  which is equivalent to $\forall X(p(X) \leftarrow \forall Y(\neg q(Y)))$

However, because of the procedural meaning of negation (Prolog starts a new search tree when evaluating a negation), the quantification of the variables that appear within the scope of a negation is not always known at the time where the program is written.

For instance, consider $p(X) \leftarrow r(X, Y) \wedge \neg q(Y)$:

- the intended meaning may be:
  for all X, $p(X)$ is true if *there exists* some Y for which $r(X, Y)$ can be proved and the query $q(Y)$ returns "no";

Quantification

In general, negation inverses the quantification of variables within its scope in the body of a rule:
$p(X) \leftarrow \neg q(Y)$ is interpreted as follows:

- for all X, $p(X)$ true if the query $q(Y)$ returns "no";
- that is, for all X, $p(X)$ is true if *there is no* Y for which $q(Y)$ can be proved
- in logic, we would write: $\forall X(p(X) \leftarrow \neg(\exists Y\, q(Y)))$,
  which is equivalent to $\forall X(p(X) \leftarrow \forall Y(\neg q(Y)))$

However, because of the procedural meaning of negation (Prolog starts a new search tree when evaluating a negation), the quantification of the variables that appear within the scope of a negation is not always known at the time where the program is written.

For instance, consider $p(X) \leftarrow r(X, Y) \wedge \neg q(Y)$:

- the intended meaning may be:
  for all X, $p(X)$ is true if *there exists* some Y for which $r(X, Y)$ can be proved and the query $q(Y)$ returns "no";
- in logic, we would write: $\forall X(p(X) \leftarrow \exists Y(r(X, Y) \wedge \neg q(Y)))$

Quantification

However, suppose the program also contains the following facts:
$r(a, b), r(c, d), q(d), r(e, \_)$, then:
$p(a)$? ; $p(c)$? $p(e)$?

Quantification

Recall: Any rule is equivalent to a set of clauses
because of properties of $\neg$, $\wedge$, $\vee$ (Boolean algebra), and because:

$\psi \leftarrow \varphi_1 \vee \varphi_2$ is equivalent to $\left\{ \begin{array}{l} \psi \leftarrow \varphi_1 \\ \psi \leftarrow \varphi_2 \end{array} \right\}$

and

$\psi \leftarrow \neg\varphi$ is equivalent to $\left\{ \begin{array}{l} \psi \leftarrow \neg q(Y) \\ q(Y) \leftarrow \varphi \end{array} \right\}$

where q is a "new" predicate, and Y is the (sequence of) variable(s) that appear in φ and not in ψ

négation : pour que le but soit le plus instancié possible

More details on running Prolog

Syntax of Edinburgh / GNU Prolog
in order of increasing priorities:

Connective	Meaning	Logic	Prolog
implication	"if"	$\leftarrow$	:-
disjunction	"or"	$\vee$	;
conjunction	"and"	$\wedge$	,
negation	"not"	$\neg$	\+
equality	"equals"	$=$	=
inequality	"different"	$\neq$	\=

Remark Most prolog interpreters require that the logical negation connective \+ is followed, in some contexts, by two pairs of parentheses. It is safe to always put two pairs of parentheses.

Question 0.2 Using a text editor, write — at the beginning of the movie database or in a separate file — definitions for predicates specified as follows:

- play_in/2, such that play_in(A, M) is true if A is an actor or an actress in movie M;
- directed_by/2, such that directed_by(A, D) is true if A played in a movie directed by D.
- common_costar/2, such that common_costar(A_1, A_2) is true if there is a third actor or actress, different from A_1 and A_2, who played with both A_1 and A_2 but in two different movies.

(Do not forget to "re-consult" your file every time you modify it.)

A few debugging tools

Step-by-step execution It is possible to see each “call” to a given predicate during execution of a program. The predicate `spy/1` is used to declare predicates to be "traced", e.g. `spy(isAscendantOf)`. It puts the interactive system in tracing mode. The predicate `debug/0` also puts the system in tracing mode. A call to `nodebug/0` exits this mode.

In tracing mode, the interpreter stops when "calling" a traced predicate and when "exiting" the call, that is, when it has found instances of the variables that make the call "true"; at each stop, the interpreter waits for an instruction:

`l` to “leap” to the next call to, or return from, a marked predicate;
`A` to see all alternatives (branches that still have to be explored);
`a` to abort execution;
`g` to see the ancestor calls;
`h` to get some help.

A few debugging tools

Exercise 1 Trace the calls to the predicates movie and director during the execution of the queries:
$\mathsf{movie}(M, 2006) \wedge \mathsf{director}(M, \mathsf{ethan\_coen})$;
$\mathsf{movie}(M, 2006) \wedge \neg\mathsf{director}(M, \mathsf{ethan\_coen})$.

A few debugging tools

Printing messages The predicate write/1 can be used to print out a term on the screen, e.g.
$p(X) \leftarrow \mathsf{write}('X =') \wedge \mathsf{write}(X) \wedge q(X)$.

The anonymous variable The variable _ is anonymous: it does not really have a name; each occurrence refers to a different variable.
It should be used whenever a variable has only one occurrence in a clause,
otherwise Prolog will write a *"singleton variable..."* message.

For instance, the clause $p(X, Y) \leftarrow q(X) \wedge r(X, Z) \wedge s(Z, U)$ should be written $p(X, \_) \leftarrow q(X) \wedge r(X, Z) \wedge s(Z, \_)$.

Exercise 2 Rewrite the programs of the exercises above so that you do not have the "singleton variable..." message when you consult them.

Always useful: working with numbers in Prolog

- Most usual arithmetic expressions are part of the Prolog language,
  for instance $\mathsf{sqrt}(4 * X - 3) + 2.3$.

Always useful: working with numbers in Prolog

- Most usual arithmetic expressions are part of the Prolog language,
  for instance $\mathsf{sqrt}(4 * X - 3) + 2.3$.

However :

- Prolog was first designed to manipulate non numerical data

$\Rightarrow$ the *evaluation* of arithmetic expressions is not automatic.
For instance, try the query $X = \mathsf{sqrt}(4 * 5 - 3) + 2.3$.

Always useful: working with numbers in Prolog

- Most usual arithmetic expressions are part of the Prolog language,
  for instance $\mathsf{sqrt}(4 * X - 3) + 2.3$.

However :

- Prolog was first designed to manipulate non numerical data

$\Rightarrow$ the *evaluation* of arithmetic expressions is not automatic.
For instance, try the query $X = \mathsf{sqrt}(4 * 5 - 3) + 2.3$.

- A Prolog predicate does not *return* a value

Always useful: working with numbers in Prolog

- Most usual arithmetic expressions are part of the Prolog language,
  for instance $\mathsf{sqrt}(4 * X - 3) + 2.3$.

However :

- Prolog was first designed to manipulate non numerical data

$\Rightarrow$ the *evaluation* of arithmetic expressions is not automatic.
For instance, try the query $X = \mathsf{sqrt}(4 * 5 - 3) + 2.3$.

- A Prolog predicate does not *return* a value

$\Rightarrow$ **The binary predicate** is
  - $*$ evaluates an arithmetic expression, and
  - $*$ instantiates a variable with the result.

For instance, to the query: X is $\mathsf{sqrt}(4 * 2 - 3) + 2.3$
Prolog replies: $X = 4.53\ldots$

is

Always useful: working with numbers in Prolog

The following infix binary predicates expect arithmetic expressions on both sides: <, >, =<, >=, =:=, =\=.
They evaluate the two expressions, and compare the results.
(`X=:=Y` is true if the value of `X` is equal to the value of `Y`,
and `X=\=Y` is true if the value of `X` is different from that of `Y`).

Always useful: working with numbers in Prolog

Exercise 3 Define a predicate that can compute the value of $n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$.

Exercise 4 Define a predicate `quadEq/4` that computes the solutions of a quadratic equation: `quadEq`(A, B, C, X) should be true if $AX^2 + BX + C = 0$.

Two puzzles

Exercise 5 Once upon a time a farmer went to the market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river and hired a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans. If left alone, the fox would eat the goose, and the goose would eat the beans. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact.[2] You will later write a Prolog program to discover how he did it, with a predicate that computes a sequence of crossings that leads from the initial state (the farmer and his goods on one side of the river) to the final state (the farmer and his goods on the other side). A state of the problem can be described with a list of the elements on the initial side of the river. For instance, the initial state could be described by [f, g, x, b], the final state is just [ ].

Question 5.1 Define a predicate safeState/1 such that safeState(E) is

[2] en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

true if E represents a state where the fox is not left alone with the goose, and the goose is not left with the bag of beans. (Use the built-in predicate member/2.)

Two puzzles

Exercise 6 Cannibals ambush a safari in the jungle and capture three men. The cannibals give the men a single chance to escape uneaten.

The captives are lined up in order of height, and are tied to stakes. The man in the rear can see the backs of his two friends, the man in the middle can see the back the man in front, and the man in front cannot see anyone. The cannibals show the men five hats. Three of the hats are black and two of the hats are white.

Blindfolds are then placed over each man's eyes and a hat is placed on each man's head. The two hats left over are hidden. The blindfolds are then removed and it is said to the men that if one of them can guess what color hat he is wearing they can all leave unharmed.

The man in the rear who can see both of his friends' hats but not his own says, "I don't know". The middle man who can see the hat of the man in front, but not his own says, "I don't know". The front man who cannot see ANYBODY'S hat says "I know!"

How did he know the color of his hat and what color was it?[3]

[3]Excerpt from mathsisfun.com

Two puzzles

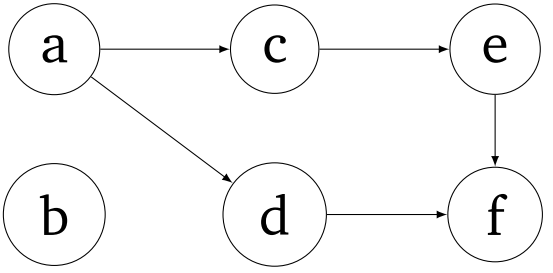
(Hint: Prolog's negation as failure is perfect to represent "I cannot guess the color of my hat knowing. . . ". You can define a first predicate that enumerates the possible hat combinations.)

About Prolog search strategy

Non terminating queries

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

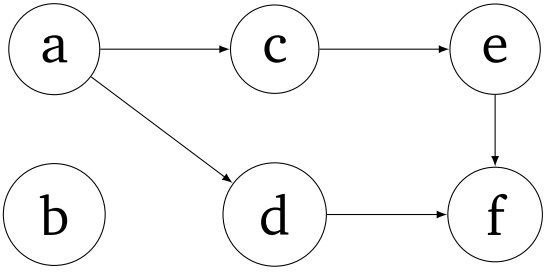
edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).

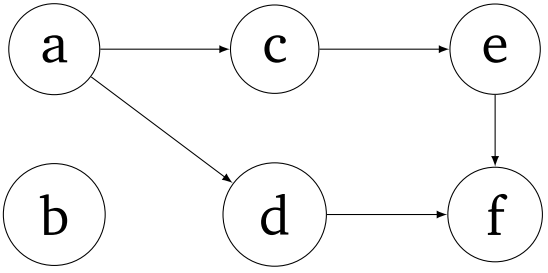


Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



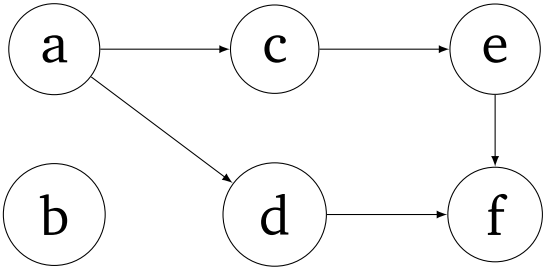
Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (edge$(X, Z) \wedge$ path(Z, Y)).

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

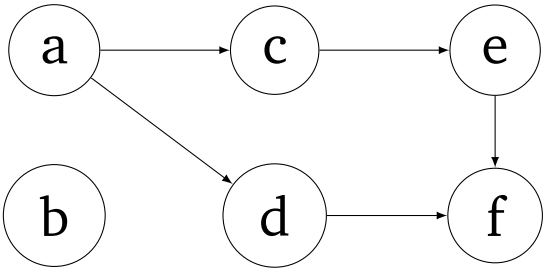
path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (edge$(X, Z) \wedge$ path(Z, Y)).

Draw the search trees for the following queries:
path(a, e)? path(e, a)? path(a, U)?

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (edge$(X, Z) \wedge$ path(Z, Y)).

Draw the search trees for the following queries:
path(a, e)? path(e, a)? path(a, U)?

Now, draw the search trees for the query path(a, f)? if we re-define path/2 as follows:
path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (path$(X, Z) \wedge$ edge(Z, Y)).

Non terminating queries : Retrieval in a graph

Is it better with the following definition?

$\mathsf{path}(X, Y) \leftarrow \mathsf{edge}(X, Y) \vee (\mathsf{edge}(Z, Y) \wedge \mathsf{path}(Z, Y))$.

$\Rightarrow$ In general, it is better to instantiate variables before the recursive call(s).

Prolog negation is not logical negation (again)

Consider the previous graph, but suppose edges mean "attacks" between nodes. We say that a node is safe if:

- it is not attacked at all; or
- it is not attacked by any safe node.

Prolog definition: $\mathsf{safe}(X) \leftarrow \neg(\mathsf{edge}(Y, X) \wedge \mathsf{safe}(Y))$.

Draw the search tree for the following queries:

$\mathsf{safe}(a)?$ $\mathsf{safe}(c)?$ $\mathsf{safe}(f)?$ $\mathsf{safe}(U)?$

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list $\Rightarrow$ Define a predicate `path`/3 such that `path`(X, Y, P) is true if P is a list of vertices on a path from X to Y:

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list
$\Rightarrow$ Define a predicate $\mathsf{path}/3$ such that $\mathsf{path}(X, Y, P)$ is true if P is a list of vertices on a path from X to Y:

$$\mathsf{path}(X, Y, P) \leftarrow \mathsf{edge}(X, Y) \wedge P = [\,] \\ \vee(\mathsf{edge}(X, Z) \wedge \mathsf{path}(Z, Y, P1) \wedge P = [Z|P1]).$$

Draw the search tree for the queries $\mathsf{path}(a, e, P)$, $\mathsf{path}(a, b, P)$ and $\mathsf{path}(a, f, P)$.

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list
$\Rightarrow$ Define a predicate path/3 such that $\mathsf{path}(X, Y, P)$ is true if P is a list of vertices on a path from X to Y:

$$\mathsf{path}(X, Y, P) \leftarrow \mathsf{edge}(X, Y) \wedge P = [\,] \\ \vee(\mathsf{edge}(X, Z) \wedge \mathsf{path}(Z, Y, P1) \wedge P = [Z|P1]).$$

Draw the search tree for the queries $\mathsf{path}(a, e, P)$, $\mathsf{path}(a, b, P)$ and $\mathsf{path}(a, f, P)$.

Exercise 1 On the graph example again.

Question 1.1 What happens if we add an edge from c to a?

Question 1.2 Re-define your predicate path/3, so that it gives "some" paths without does not looping even if the graph has cycles.
(Hint: use a list of "forbidden" nodes, and an intermediate predicate path/4.)

Recursive predicates that “construct” a list

More difficult: give a solution that enumerates all paths (is complete) when the graph has cycles.

Exercise 2 Write definitions for the following predicates to manipulate lists:

select/3**:** a predicate that can be used to “delete" an occurrence of an element of a list. For instance, to the query $\mathsf{select}(X, [a, b, c, a], R)$ Prolog should answer:
$X = a \wedge R = [b, c, a] \vee X = b \wedge R = [a, c, a] \vee X = c \wedge L = [a, b, a] \vee X = a \wedge L = [a, b, c]$.

concat/3**:** $\mathsf{concat}(L, M, R)$ is true if R is the list that contain the elements of L followed by those of M.

Exercise 3 Define a predicate to compute the length of a list, and another predicate to *generate* a list of a given length.

When the result is at a leaf of the Prolog's search tree

It often happens that the result that we want to construct is only available at the leaves of the search tree.
Suppose for instance that we want a predicate that can "reverse" a list:
$\mathsf{reverse}([a, b, c], R) \Rightarrow R = [c, b, a]$
First solution with append: $\mathsf{append}(L_1, L_2, L_3)$ is true if "$L_3 = L_1.L_2$".
$\mathsf{reverse}(L, M) \leftarrow L = [\,] \wedge M = [\,]$
$\quad \vee L = [X|R] \wedge \mathsf{reverse}(R, S) \wedge \mathsf{append}(S, [X], M).$
Number of operations in $O(|L|^2)$.

When the result is at a leaf of the Prolog's search tree

Better solution?

$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$

When the result is at a leaf of the Prolog's search tree

Better solution?

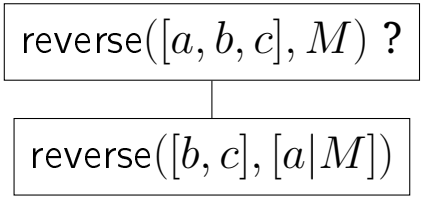
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$

$\mathsf{reverse}([a, b, c], M)$?

When the result is at a leaf of the Prolog's search tree

Better solution?

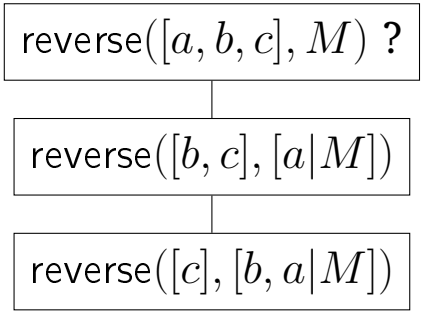
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

Better solution?

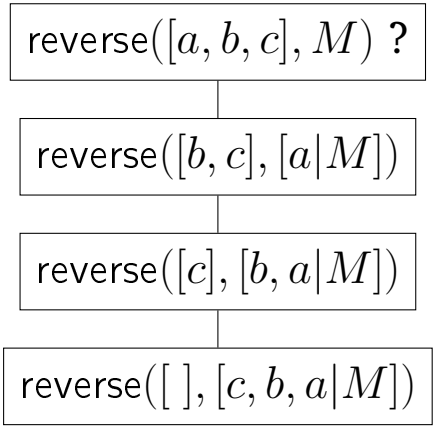
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

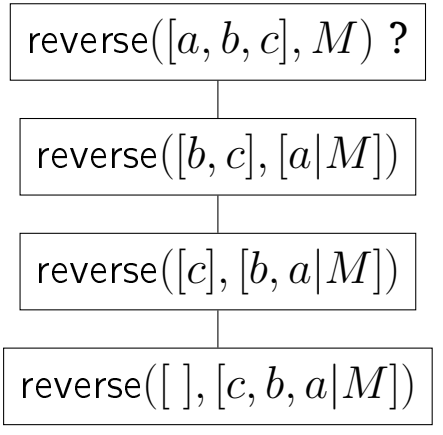
Better solution?

$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

Better solution?
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



With an *accumulator*:
$\mathsf{reverse}(L,A,M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R,[X|A],M) \vee L = [\,] \wedge M = A$
$\mathsf{reverse}(L, M) \leftarrow \mathsf{reverse}(L, [\,], M)$
Number of operations in $O(|L|)$

Remark The predicates $\mathsf{member}/2$, $\mathsf{append}/3$, $\mathsf{select}/3$ and $\mathsf{reverse}/2$ are usually predefined in Prolog.

List of solutions

Sometimes we would be happy to compute a list of all solutions to a given predicate.
Suppose for instance we want all movies directed by Woody Allen, sorted in alphabetical ordering.

?? Difficult:

- all solutions to the query director('Allen, Woody', M) are in different branches of Prolog's search tree
- all branches of the search tree are independent of one another

List of solutions

Good implementations of Prolog provide a *meta-predicate* that lists solutions to a given query:

findall$(T, G(T), L)$**:** here $G(T)$ means that G is goal (formula) in which the term T appears; then the call will

- call the goal B;
- for each solution found, instantiate the term T according to the solution;
- construct the list L of these instances T.

For instance, with the movie database excerpt:

findall$(A,$ directed('Fatih Akin', $A), L)? \Rightarrow L =$ ['Adam Bousdoukos', 'Moritz Bl

findall$([D, M, Y],$ director$(D, M), L). \Rightarrow L = \ldots$

List of solutions

bagof$(T, G(T), L)$**:** similar to findall, but the results are grouped according to the values of variables of G which do not appear in T

On the example: bagof$(A,$ directed$(D, A), L)? \Rightarrow \ldots$

Exercise 4 Consider the graph above again: define a predicate reachable/2, such that reachable(X, L) is true if L is the list of vertices to which there is a path from X.

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, recursion needs to store in a stack the successive values of the parameters with which the recursive predicate is called.

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, recursion needs to store in a stack the successive values of the parameters with which the recursive predicate is called.

For instance, consider the following program, which prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, recursion needs to store in a stack the successive values of the parameters with which the recursive predicate is called.

For instance, consider the following program, which prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

The query $\mathsf{count}(1, 1000000)$ works fine

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, recursion needs to store in a stack the successive values of the parameters with which the recursive predicate is called.

For instance, consider the following program, which prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

The query $\mathsf{count}(1, 1000000)$ works fine

But if we define a predicate to count backwards:

$\mathsf{revcount}(X, X).$

$\mathsf{revcount}(X,N) \leftarrow X < N \wedge Y \mathsf{is} X + 1 \wedge \mathsf{revcount}(Y,N) \wedge \mathsf{write}(Y) \wedge \mathsf{nl}$

the query $\mathsf{revcount}(1,1000000)$ leads to a crash: stack overflow!

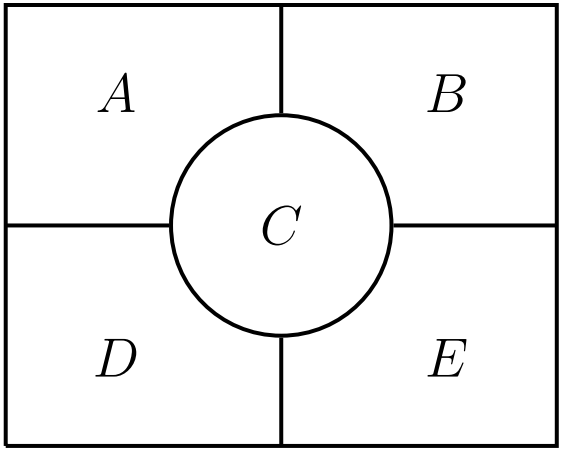
Last call optimization

Explanation Most compilers for functional / logic programming languages are able to transform a recursion into an iteration, if the recursive call is the last one in the rule.

In prolog, the recursion is usually transformed into an iteration if:

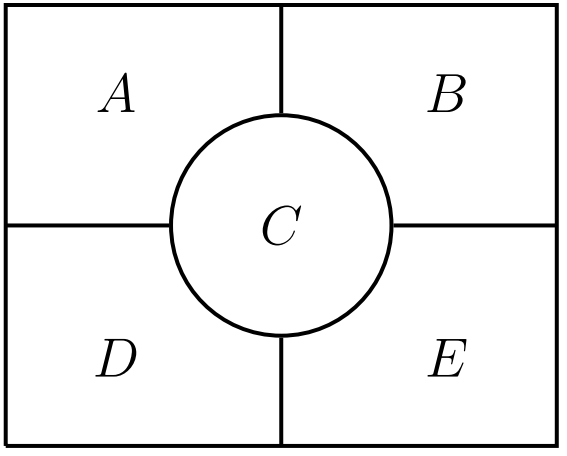
- the recursive call is the last one in the rule, and
- no other rule can be tried after the recursive call
  (the other rules must therefore in general appear *before* the recursive one
- no more backtrack is possible with the other goals in the rule

Generate and test



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with fewer colors?)

Generate and test : The "Map coloring" example



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with fewer colors?)

$\mathsf{solution}(A, B, C, D, E) \leftarrow \mathsf{generate}(A, B, C, D, E) \wedge \mathsf{test}(A, B, C, D, E).$

$\mathsf{generate}(A, B, C, D, E) \leftarrow$

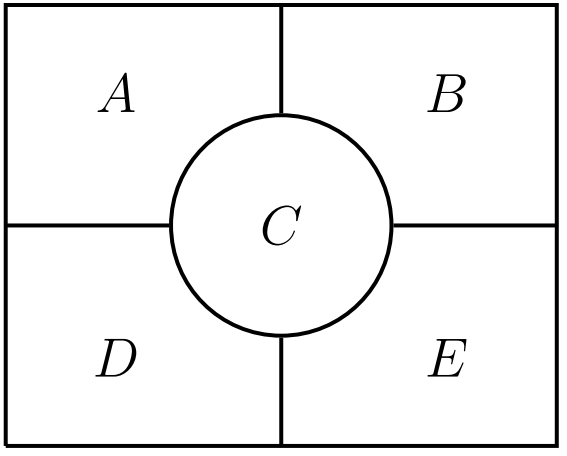
$\quad \mathsf{color}(A) \wedge \mathsf{color}(B) \wedge \mathsf{color}(C) \wedge \mathsf{color}(D) \wedge \mathsf{color}(E).$

$\mathsf{color}(X) \leftarrow X = \mathsf{green} \vee X = \mathsf{purple} \vee X = \mathsf{orange}.$

$\mathsf{test}(A, B, C, D, E) \leftarrow A \neq B \wedge B \neq E \wedge E \neq D \wedge D \neq A$

$\quad \wedge C \neq A \wedge C \neq B \wedge C \neq E \wedge C \neq D.$

Generate and test : The "Map coloring" example



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with fewer colors?)

$\mathsf{solution}(A, B, C, D, E) \leftarrow \mathsf{generate}(A, B, C, D, E) \wedge \mathsf{test}(A, B, C, D, E).$
$\mathsf{generate}(A, B, C, D, E) \leftarrow$
$\quad \mathsf{color}(A) \wedge \mathsf{color}(B) \wedge \mathsf{color}(C) \wedge \mathsf{color}(D) \wedge \mathsf{color}(E).$
$\mathsf{color}(X) \leftarrow X = \mathsf{green} \vee X = \mathsf{purple} \vee X = \mathsf{orange}.$
$\mathsf{test}(A, B, C, D, E) \leftarrow A \neq B \wedge B \neq E \wedge E \neq D \wedge D \neq A$
$\quad \wedge C \neq A \wedge C \neq B \wedge C \neq E \wedge C \neq D.$

$\Rightarrow$ How many leaves does the search tree have?
(In general, this problem is called graph coloring.)

Exercises

Exercise 5 Consider the movie database. Let us define the *degree of movie separation* between two actors or actresses A_1 and A_2 as follows: it is 0 if they played in the same movie at least once; it is 1 if they did not play in the same movie, but played in movies that have at least one common actor or actress; it is 2 if it is not 1 and there are two other actors or actresses B_1 and B_2 who played in the same movie, and such that A_1 and B_1 (respectively A_2 and B_2) played in the same movie; and so on... That is, it is the length of the shortest "movie path" between them. By definition, the degree will be infinite if there is no "movie connection" between two persons.

Exercises : On graph traversal

Question 5.1 Define a predicate that can be used to find actors and actresses who have a degree of movie separation of 2 with a given actor or actress A.

Question 5.2 Define a predicate that can compute the degree of movie separation between two given actors or actresses.

Exercises : On graph traversal

Exercise 6 The "flights" Prolog database [4] contains Prolog facts with the following information:

- flying times, for instance
  $\mathsf{vol}(\mathsf{it}, 386, \mathsf{blagnac}, \mathsf{cdg}, [14, 30], [15, 30])$
  indicates that flight 386 of the company “it” takes off from Blagnac airport at 14h30 and lands at Charles-de-Gaulle at 15h30; in particular, times of the day are represented using lists giving the hour and minutes.
- flight prices, for instance
  $\mathsf{tarif}(\mathsf{it}, \mathsf{toulouse}, \mathsf{paris}, 500)$
  indicates that a ticket to fly from Toulouse to Paris with “it” costs 500€;
- the cost of airport taxes, for instance
  $\mathsf{taxe}(\mathsf{toulouse}, 100)$
  indicates that every passenger using an airport in Toulouse must pay a 100€ tax every time;

[4] www.irit.fr/~Jerome.Mengin/teaching/prolog/vols-payants-bd.pl

Exercises : On graph traversal

- location of airports, for instance
  aeroport(toulouse, blagnac)
  indicates that Blagnac airport is near Toulouse.

Question 6.1 Load the file, and submit queries to get: all flight numbers from Blagnac to Orly, flight numbers from Marseille to Paris, all London airports.

Question 6.2 Define a relation directConnection/4 such that connection(A_D, is true if there is a direct flight from airport A_D to airport A_A leaving after time H_D and arriving before H_A.

Question 6.3 Define now a relation route/4 such that route(A_D, H_D, A_A, H is true if there is a sequence of flights to go from airport A_D to airport A_A leaving after time H_D and arriving before H_A. In case of stopovers, there must be at least 30 minutes if flights arrive and start from the same airport, and 2 hours if one must change airport in the same city.

Question 6.4 Extend the preceding relation so that one can get the price of the ticket, and the itinerary. (Airport taxes are paid for the starting and arrival airports, as well as once for each stopover.)

Exercises : On graph traversal

Exercise 7 Once upon a time a farmer went to the market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river and hired a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans. If left alone, the fox would eat the goose, and the goose would eat the beans. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact.[5] You are asked to write a Prolog program to discover how he did it. More precisely, you must define a predicate that computes a sequence of crossings that leads from the initial state (the farmer and his goods on one side of the river) to the final state (the farmer and his goods on the other side). A state of the problem can be described with a lists of the elements on the initial side of the river. For instance, the initial state could be described by $[\mathsf{f}, \mathsf{g}, \mathsf{x}, \mathsf{b}]$. The main predicate of your program, plan/1, must be defined so that the query:
$?-\mathsf{plan}(L)$.

[5] en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

yields a list of successive states that lead from the initial state to the final state: $L =$ [[f, x, g, b] , [x, b] , [f, x, b] . . . [ ]].

Question 7.1 Define a predicate safeState/1 such that safeState(E) is true if E represents a state where the fox is not left alone with the goose, and the goose is not left with the bag f beans. (Use the built-in predicate member/2.)

Question 7.2 Define a predicate crossing/2 such that crossing($E1, E2$) is true if it is possible to change from state $E1$ to state $E2$ with a single crossing of the farmer, with or without one good. (Use the built-in predicate select/3.)

Question 7.3 Define a predicate plan/4 such that plan(I, F, P, N) is true if it possible to go from state I to state F with the states in the list P as intermediary states, without going through the states in the list of forbidden states N. (Note: checking that a state is not in N is not trivial, since there are different possibilities to describe one state; for instance, [f, x, b] and [x, f, b] may refer to the same state.)

Question 7.4 Finally define plan/1: plan(P) must be true if P is a se-

quence of states that describe a valid plan from the initial state to the final state.

Exercises : On list operations

Exercise 8 You are asked to write a program that analyses how some objects are composed of other objects: their components, that can themselves be decomposed into components. A small database contains, for each object, the list of its components, using a relation components/2: components(O, L) is true if L is the list of components of object O.

components$(a, [b, c, d, c])$. components$(b, [e, f])$. components$(f, [g, e]$

Thus a has four components, two of type c ; b can itself be decomposed, as well as c and f, whereas d, e, g and h are elementary components.

Question 8.1 Define a predicate allComp/2, such that allComp(O, L) is true if L is the list of all the components, elementary or not, that constitute object O - including O itself. For instance, the query allComp$(a, U$ should yield the answer $U = [a, b, e, f, g, e, c, h, h, h, d, c, h, h, h]$. (The built-in predicate append/3 can be useful.)

Question 8.2 Define a predicate compEl, such that compEl(O, L) is true if L is the list of elementary components of object O. For instance, the query compEl(a, U) should yield $U = [e, g, e, h, h, h, d, h, h, h]$.

Exercises : On the "Generate & test paradigm"

Exercise 9 Ann, Bill, Charlie, Don, Eric own one box each but we don't know which box. We know the size and color of each box: one box is of size 3 and black; one is of size 1 and black; one is of size 1 and white; one is of size 2 and black; the last one is of size 3 and white. We also have some information about the characteristics of the boxes owned by each person :

- Ann and Bill have boxes with the same colour;
- Don and Eric have boxes with the same colour;
- Charlie and Don have boxes with the same size;
- Eric's box is smaller than Bill's.

We want to know who owns which box. In order to solve the problem, you can:

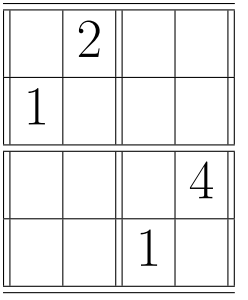
1. Write a Prolog database with the characteristics of the boxes, associating a number to each box, for instance: $\mathsf{box}(2, 1, \mathsf{black})$ represents that the second box is of size 1 and black;
2. Write a predicate to compute the solution of the problem: $\mathsf{solution}(A, B, C, D, E)$ must be true if A is the box owned by Ann, B the one owned by Bill, and so on...

Exercises : On the “Generate & test paradigm”

Exercise 10 The arithmetic cryptographic puzzle: Find distinct digits for S, E, N, D, M, O, R, Y such that S and M are non- zero and the equation SEND+MORE=MONEY is satisfied.

The GNU Prolog finite domain constraints solver

The Sudoku 4×4 in Prolog



$\mathsf{sudoku4}(L) \leftarrow \mathsf{generate}(L) \wedge \mathsf{test}(L).$

$\mathsf{generate}(L) \leftarrow \mathsf{all\_member}(L,[1,2,3,4]).$

(3) $\mathsf{test}(X_{11}, X_{12}, X_{13}, X_{14}, X_{21}, X_{22}, \ldots, X_{42}, X_{43}, X_{44}) \leftarrow$
$\wedge\ \mathsf{all\_diff}([X_{11}, X_{12}, X_{13}, X_{14}]) \wedge \mathsf{all\_diff}([X_{21}, X_{22}, X_{23}, X_{24}])$
$\wedge\ \mathsf{all\_diff}([X_{31}, X_{32}, X_{33}, X_{34}]) \wedge \mathsf{all\_diff}([X_{41}, X_{42}, X_{43}, X_{44}])$
$\wedge\ \mathsf{all\_diff}([X_{11}, X_{21}, X_{31}, X_{41}]) \wedge \mathsf{all\_diff}([X_{14}, X_{24}, X_{34}, X_{44}])$
$\wedge\ \mathsf{all\_diff}([X_{12}, X_{22}, X_{32}, X_{42}]) \wedge \mathsf{all\_diff}([X_{13}, X_{23}, X_{33}, X_{43}])$
$\wedge\ \mathsf{all\_diff}([X_{11}, X_{12}, X_{21}, X_{22}]) \wedge \mathsf{all\_diff}([X_{13}, X_{14}, X_{23}, X_{24}])$
$\wedge\ \mathsf{all\_diff}([X_{31}, X_{32}, X_{41}, X_{42}]) \wedge \mathsf{all\_diff}([X_{33}, X_{34}, X_{43}, X_{44}]).$

(4) $\mathsf{all\_member}(L, D) \leftarrow$
$L = [\,] \vee L = [V|R] \wedge \mathsf{member}(V, D) \wedge \mathsf{all\_member}(R, D).$

(5) $\mathsf{all\_diff}(L) \leftarrow L = [\,] \vee L = [V|R] \wedge \neg\mathsf{member}(V, R) \wedge \mathsf{all\_diff}(R).$

Query:

$\mathsf{sudoku4}([X_{11},2,X_{13},X_{14},1,X_{22},X_{23},X_{24},X_{31},X_{32},X_{33},4,X_{41},X_{42},1,X_{44}]).$

The Sudoku 4×4 in Prolog : The basic "Generate & test" solution

The search tree has $4^{12} = 17$ million leaves !!

The Sudoku 4×4 in Prolog : A more efficient version

Generate only valide lines
(somehow, the "generate" and "test" parts are interleaved).

$\mathsf{sudoku4}(L) \leftarrow \mathsf{generate}(L) \wedge \mathsf{test}(L).$

$\mathsf{generate}(X_{11},X_{12},X_{13},...,X_{42},X_{43},X_{44}) \leftarrow$
$\quad \mathsf{all\_member\_diff}([X_{11},X_{12},X_{13},X_{14}]) \wedge \mathsf{all\_member\_diff}([X_{21},X_{22},X_{23},$
$\wedge \mathsf{all\_member\_diff}([X_{31},X_{32},X_{33},X_{34}]) \wedge \mathsf{all\_member\_diff}([X_{41},X_{42},X_{43},X$

$\mathsf{test}(X_{11},X_{12},X_{13},...,X_{42},X_{43},X_{44}) \leftarrow$
$\quad \mathsf{all\_diff}([X_{11},X_{21},X_{31},X_{41}]) \wedge \mathsf{all\_diff}([X_{14},X_{24},X_{34},X_{44}])$
$\wedge \mathsf{all\_diff}([X_{12},X_{22},X_{32},X_{42}]) \wedge \mathsf{all\_diff}([X_{13},X_{23},X_{33},X_{43}])$
$\wedge \mathsf{all\_diff}([X_{11},X_{12},X_{21},X_{22}]) \wedge \mathsf{all\_diff}([X_{13},X_{14},X_{23},X_{24}])$
$\wedge \mathsf{all\_diff}([X_{31},X_{32},X_{41},X_{42}]) \wedge \mathsf{all\_diff}([X_{33},X_{34},X_{43},X_{44}]).$

$\mathsf{all\_member\_diff}(L,D) \leftarrow L = [\,] \vee$
$L = [V|R] \wedge \mathsf{select}(V,D,S) \wedge \mathsf{all\_member\_diff}(R,S).$

$\mathsf{all\_diff}(L) \leftarrow L = [\,] \vee L = [V|R] \wedge \neg\mathsf{member}(V,R) \wedge \mathsf{all\_diff}(R).$

Remark $\mathsf{select}(V, D, RD)$ is true if $V \in D$ and $RD = D \backslash V$.

The Sudoku 4×4 in Prolog : With the constraint solver

$$
\begin{aligned}
&\mathsf{sudoku4\_fd}(L) \leftarrow L = [X_{11},X_{12},X_{13},...,X_{42},X_{43},X_{44}] \\
&\quad \wedge \mathsf{fd\_domain}(L,[1,2,3,4]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{11},X_{12},X_{13},X_{14}]) \wedge \mathsf{fd\_all\_diff}([X_{21},X_{22},X_{23},X_{24}]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{31},X_{32},X_{33},X_{34}]) \wedge \mathsf{fd\_all\_diff}([X_{41},X_{42},X_{43},X_{44}]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{11},X_{21},X_{31},X_{41}]) \wedge \mathsf{fd\_all\_diff}([X_{14},X_{24},X_{34},X_{44}]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{12},X_{22},X_{32},X_{42}]) \wedge \mathsf{fd\_all\_diff}([X_{13},X_{23},X_{33},X_{43}]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{11},X_{12},X_{21},X_{22}]) \wedge \mathsf{fd\_all\_diff}([X_{13},X_{14},X_{23},X_{24}]) \\
&\quad \wedge \mathsf{fd\_all\_diff}([X_{31},X_{32},X_{41},X_{42}]) \wedge \mathsf{fd\_all\_diff}([X_{33},X_{34},X_{43},X_{44}]) \\
&\quad \wedge \mathsf{fd\_labeling}(L).
\end{aligned}
$$

The Sudoku 4×4 in Prolog : Comparison of the 3 versions

- first version: implemented without thinking about how Prolog evaluates the queries, too slow.
- second version: faster, but the programmer had to think more about Prolog's evaluation mechanism – this is not the aim with logic programming.
- third version: even faster, and written without thinking about how constraints are solved.
  It uses an external constraint solver.

A quick overview of the constraint solver

1. each variable receives an initial *domain*;
   (above: the call $\mathsf{fd\_domain}(L, [1,2,3,4])$ associates the domain $\{1,2,3,4\}$ to all variables in L)

A quick overview of the constraint solver : The basic mechanism

1. each variable receives an initial *domain*;
   (above: the call $\mathsf{fd\_domain}(L, [1,2,3,4])$ associates the domain $\{1,2,3,4\}$ to all variables in L)
2. every encountered *constraint* is stored

A quick overview of the constraint solver : The basic mechanism

1. each variable receives an initial *domain*;
   (above: the call fd_domain$(L, [1, 2, 3, 4])$ associates the domain $\{1, 2, 3, 4\}$ to all variables in L)
2. every encountered *constraint* is stored
   with domain reduction based on local consistency conditions if possible; above:
   - X_{12} instantiated to 2
   - constraint fd_all_diff$([X_{11}, X_{12}, X_{13}, X_{14}])$

   $\Rightarrow$ the value 2 is removed from the domains of X_{11}, X_{13}, X_{14}
3. the call fd_labeling triggers the external constraint solver.

A quick overview of the constraint solver : Domains

A domain D_X is associated with each variable X that appears in a constraint.

Initially: $D_X = [0, \ldots, \mathsf{fd\_max\_integer}] \subseteq \mathbf{N}^+$

```
| ?- X = Y.
Y = X
yes
```

```
| ?- X #= Y.
X = _#0(0..268435455)
Y = _#0(0..268435455)
yes
```

A quick overview of the constraint solver : Domains

A domain D_X is associated with each variable X that appears in a constraint.

Initially: $D_X = [0, \ldots, \mathsf{fd\_max\_integer}] \subseteq \mathbf{N}^+$

```
| ?- X = Y.
Y = X
yes
```

```
| ?- X #= Y.
X = _#0(0..268435455)
Y = _#0(0..268435455)
yes
```

```
| ?- X\=Y.
no
```

```
| ?- X\==Y.
yes
```

```
| ?- X #\= Y.
X = _#2(0..268435455)
Y = _#20(0..268435455)
yes
```

A quick overview of the constraint solver : Domains

The first effect of a constraint is to reduce the domain of the variables:

```
| ?- X + Y #= 5.
X = _#21(0..5)
Y = _#39(0..5)
yes
```

A quick overview of the constraint solver : Domains

The first effect of a constraint is to reduce the domain of the variables:

```
| ?- X + Y #= 5.
X = _#21(0..5)
Y = _#39(0..5)
yes


| ?- X #< 3.
X = _#2(0..2)
yes
```

A quick overview of the constraint solver : Domains

The first effect of a constraint is to reduce the domain of the variables:

```
| ?- X + Y #= 5.
X = _#21(0..5)
Y = _#39(0..5)
yes
```

```
| ?- X #< 3.
X = _#2(0..2)
yes
```

```
| ?- X #< 3 , X+Y #= 6.
X = _#2(0..2)       Y = _#41(4..6)
yes
```

A quick overview of the constraint solver : Domains

```
| ?- X #< 3 , write(X) , nl , write(Y)
, X + Y #= 6.
_#2(0..2)
_22
X = _#2(0..2)      Y = _#41(4..6)
yes
```

A quick overview of the constraint solver : Domains

```
| ?- X #< 3 , write(X) , nl , write(Y)
, X + Y #= 6.
_#2(0..2)
_22
X = _#2(0..2)     Y = _#41(4..6)
yes


| ?- X #< 2 , Y #< 2 , Z #< 2
, X #\= Y , X #\= Z , Y #\= Z.
X = _#2(0..1)  Y = _#22(0..1)  Z = _#42(0..1)
yes
```

A quick overview of the constraint solver : Domains

Remarks:

- the predicates #=, #>, ... do not completely solve the constraints.
- the evaluation of each constraint C only eliminates from the domains of the variables values that do not appear in any solution of C:
  it ensure *local consistency*
  (it is local to *one* constraint)

A quick overview of the constraint solver : Invoking the solver

The predicate fd_labeling solves all the constraints that have been *posted* :

```
| ?- X #< 3 , X + Y #= 6 , fd_labeling([X,Y]).
X = 0    Y = 6 ? ;
X = 1    Y = 5 ? ;
X = 2    Y = 4
yes
```

A quick overview of the constraint solver : Invoking the solver

```
| ?- X #< 2 , Y #< 2 , Z #< 2 , X #\= Y , X #\= Z , Y #\= Z
, fd_labeling([X,Y,Z]).
no
```

The actual constraint solving algorithm will not be studied here...

Remark: all constraints are simultaneously solved, not only the ones that involve the variables that appear in the parameter of fd_labeling:

```
| ?- X #< 2 , Y #< 2 , Z #< 2 , X #\= Y , X #\= Z , Y #\= Z
, fd_labeling([X,Y]).
no
```

A quick overview of the constraint solver : Other predicates

fd_domain(X, L): removes from the domain of X values that are not in L.

fd_domain_bool(L): removes from the domain of each variable in L values that are not in $\{0, 1\}$.

fd_all_different(L): constraints all variables in the list L to have different values.

`fd_all_different([X,Y,Z])` is equivalent to:

```
X #\= Y , X #\= Z , Y #\= Z
```

A quick overview of the constraint solver : Other predicates

```
| ?- fd_domain_bool([X,Y,Z]) , fd_all_different([X,Y,Z]).
X = _#0(0..1)   Y = _#18(0..1)   Z = _#36(0..1)
yes

| ?- fd_domain_bool([X,Y,Z]) , fd_all_different([X,Y,Z])
, fd_labeling([X,Y,Z]).
no
```

fd_atmost(N, L, V): imposes that at most N variables from the list L have value V.

There is also fd_atleast and fd_exactly.

A quick overview of the constraint solver

Example : the 8 queens ...

A quick overview of the constraint solver : Optimisation

The predicate fd_minimize returns the minimum value allowed for a variable X among those possible when constraints are solved with fd_labeling:

```
| ?- X + Y #= 10 , Y #< 3 , fd_minimize(fd_labeling([X,Y]),X)
X = 8    Y = 2
yes
```

How it works:

- each time fd_labeling($[X, Y]$) gives a solution $X = n$, the search is started again with a new constraint $X \#< n$;
- when a failure occurs (either because there are no remaining choice-points for Goal or because the added constraint is inconsistent with the rest of the store) the last solution is recomputed since it is optimal.

There is also fd_maximize.

A quick overview of the constraint solver : Optimisation

Example : graph coloring

```
| ?- X #\= Y , X #\= Z , X #< Max , Y #< Max , Z #< Max
, fd_minimize(fd_labeling([X,Y,Z]), Max).
```

Exercises

Exercise 1 A factory has four workers w1,w2,w3,w4 and four products p1,p2,p3,p4. The problem is to assign workers to products so that each worker is assigned to one product, each product is assigned to one worker, and the profit maximized. The profit made by each worker working on each product is given in the matrix:

	p1	p2	p3	p4
w1	7	1	3	4
w2	8	2	5	1
w3	4	3	7	2
w4	3	1	6	3

Exercises

Exercise 2 Four roommates are subscribing to four newspapers. The table gives the amounts of time each person spends on each newspaper. Akiko gets up at 7:00, Bobby gets up at 7:15, Chloé gets up at 7:15, and Dola gets up at 8:00.

	The Guardian	Le Monde	El Pais	Die Taz
Albert	60	30	2	5
Bobby	75	3	15	10
Chloé	5	15	10	30
Dola	90	1	1	1

Nobody can read more than one newspaper at a time and at any time a newspaper can be read by only one person. Schedule the newspapers such that the four persons finish the newspapers at an earliest possible time.

Exercises

Exercise 3 Write a program that computes a coloring of a graph, using a minimum number of colors, for any graph described by instances of a predicate `edge`/2.