Prolog Course at ENAC

Jan-Georg Smaus

Year 2022/23

1: Logic program examples and references

1.1: 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').

1.1: 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).

C = germany
Y = 2009 ? ;

C = france
Y = 2010
```

1.1: An excerpt from a movie database

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

1. the director of The Blues Brothers;

1.1: An excerpt from a movie database

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

1. the director of The Blues Brothers;

```
| ?- director(D,'The Blues Brothers').

D = 'John Landis' ?

yes
```

2. if Aretha Franklin played in a film by John Landis;

1.1: An excerpt from a movie database

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

1. the director of The Blues Brothers;

```
| ?- director(D,'The Blues Brothers').

D = 'John Landis' ?

yes
```

2. if Aretha Franklin played in a film by John Landis;

```
| ?- cast(M,'Aretha Franklin',R), director('John Landis',M).

M = 'The Blues Brothers'
R = 'Mrs. Murphy' ? ;

no
```

1.1: An excerpt from a movie database

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

1. the director of The Blues Brothers;

```
| ?- director(D,'The Blues Brothers').

D = 'John Landis' ?

yes
```

2. if Aretha Franklin played in a film by John Landis;

```
| ?- cast(M,'Aretha Franklin',R), director('John Landis',M).

M = 'The Blues Brothers'
R = 'Mrs. Murphy' ? ;

no
```

Observations:

- In both cases, there is only one answer, but in the first case, the Prolog interpreter detects this and does not ask user if she wants alternatives.
- We get the *name* of the movie and even the *role* although we might not care.

1.1: An excerpt from a movie database

3. actors of movies by Fatih Akin;

3. actors of movies by Fatih Akin;

```
| ?- director('Fatih Akin',M), cast(M,A,R).

A = 'Adam Bousdoukos'
M = 'Soul Kitchen'
R = 'Zinos Kazantsakis' ? ;

A = 'Moritz Bleibtreu'
M = 'Soul Kitchen'
R = 'Illias Kazantsakis' ? ;

A = 'Anna Bederke'
M = 'Soul Kitchen'
R = 'Lucia Faust'

yes
```

   We get the names of the movies and roles although we might not care.

4. the directors of movies in which Dan Aykroyd and Anna Bederke were co-stars;

1.1: An excerpt from a movie database

3. actors of movies by Fatih Akin;

```
| ?- director('Fatih Akin',M), cast(M,A,R).

A = 'Adam Bousdoukos'
M = 'Soul Kitchen'
R = 'Zinos Kazantsakis' ? ;

A = 'Moritz Bleibtreu'
M = 'Soul Kitchen'
R = 'Illias Kazantsakis' ? ;

A = 'Anna Bederke'
M = 'Soul Kitchen'
R = 'Lucia Faust'

yes
```

   We get the names of the movies and roles although we might not care.

4. the directors of movies in which Dan Aykroyd and Anna Bederke were co-stars;

```
| ?- director(D,M), cast(M,'Dan Aykroyd',R1), cast(M,'Anna Bederke',R2).

no
```

   Apparently there are no such movies.

1.1: An excerpt from a movie database

5. if Anna Bederke played in a movie by John Landis or Fatih Akin;

1.1: An excerpt from a movie database

5. if Anna Bederke played in a movie by John Landis or Fatih Akin;

```
| ?- cast(M,'Anna Bederke',R), director('Fatih Akin',M).

M = 'Soul Kitchen'
R = 'Lucia Faust' ?

yes
| ?- cast(M,'Anna Bederke',R), director('John Landis',M).

no
```

We try the directors one by one and get an answer more specific than what we asked for.

5. if Anna Bederke played in a movie by John Landis or Fatih Akin;

```
| ?- cast(M,'Anna Bederke',R), director('Fatih Akin',M).

M = 'Soul Kitchen'
R = 'Lucia Faust' ?


yes
| ?- cast(M,'Anna Bederke',R), director('John Landis',M).

no
```

We try the directors one by one and get an answer more specific than what we asked for.
Alternative :

```
| ?- cast(M,'Anna Bederke',R), (director('Fatih Akin',M); director('John Landis',M)).

M = 'Soul Kitchen'
R = 'Lucia Faust' ? ;


no
```

At this point it might become clear that:

- "," stands for the logical conjunction ("and", $\wedge$),
- ";" stands for the logical disjunction ("or", $\vee$)

1.1: An excerpt from a movie database

6. actors who are also directors?

1.1: An excerpt from a movie database

6. actors who are also directors?

```
| ?- cast(M,AD,R), director(AD,M2).

no
```

   Apparently there are none.

7. actors who played with Dan Aykroyd.

6. actors who are also directors?

```
| ?- cast(M,AD,R), director(AD,M2).

no
```

Apparently there are none.

7. actors who played with Dan Aykroyd.

```
| ?- cast(M,'Dan Aykroyd',R), cast(M,A,R2).

A = 'Dan Aykroyd'
M = 'The Blues Brothers'
R = '"Joliet" Jake Blues'
R2 = '"Joliet" Jake Blues' ? ;

A = 'Aretha Franklin'
M = 'The Blues Brothers'
R = '"Joliet" Jake Blues'
R2 = 'Mrs. Murphy' ? ;

no
```

You need to make explicit that you don't count Dan Aykroyd as an actor playing with Dan Aykroyd $\Longrightarrow$ negation (see later).

1.1: An excerpt from a movie database

- We have seen "," and ";". Another important logical connective is *implication*: $\rightarrow$.
- In logic programming, it is usually reversed ($\leftarrow$) and written `:-`.
- It is needed to write *rules*, which are used for defining new *relations* from given ones. E.g., $\mathsf{directed}(D, A)$ is true if A played in a movie directed by D

```
directed(D,A) :-
  director(D,M),
  cast(M,A,R).
```

1.1: An excerpt from a movie database

Add rules to the database to define the following:

1. $\mathsf{co_star}(A1, A2)$ – the actor/actress played in the same movie

1.1: An excerpt from a movie database

Add rules to the database to define the following:

1. co_star($A1, A2$) – the actor/actress played in the same movie

```
co_star(A1,A2) :-
  cast(M,A1,R1),
  cast(M,A2,R2).
```

   Problem with this code?

1.1: An excerpt from a movie database

Add rules to the database to define the following:

1. co_star($A1, A2$) – the actor/actress played in the same movie

   ```
   co_star(A1,A2) :-
     cast(M,A1,R1),
     cast(M,A2,R2).
   ```

   Problem with this code?
2. the films in which played some actor who played in a film by Fatih Akin or John Landis

1.1: An excerpt from a movie database

Add rules to the database to define the following:

1. co_star($A1, A2$) – the actor/actress played in the same movie

   ```
   co_star(A1,A2) :-
     cast(M,A1,R1),
     cast(M,A2,R2).
   ```

   Problem with this code?

2. the films in which played some actor who played in a film by Fatih Akin or John Landis

   ```
   akin_or_landis_film(F) :-
     (director('Fatih Akin',F2);
      director('John Landis',F2)),
     cast(F2,A,R2),
     cast(F,A,R).
   ```

1.2: 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.

1.2: 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.

1.2: 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

1.3: 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.

2: First steps in Prolog

2.1: 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`.

2.1: 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.

2.2: First Exercise

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

Exercise 2 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;

2.2: First Exercise

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

Exercise 2 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;

For stating that 2 movies are different, you will need the \= predicate. Observe how its position in the query matters!

This also allows us to improve some previous code:

```
| ?- cast(M,'Dan Aykroyd',R), cast(M,A,R2), A\='Dan Aykroyd'.

co_star(A1,A2) :-
  cast(M,A1,R1),
  cast(M,A2,R2),
  A1 \= A2.
```

3: Syntax of Prolog programms

3.1: Atoms

atom: a name adhering to certain syntactic conventions; it is used in Prolog for several purposes. An atom 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 'The Blues Brothers' germany

3.2: Terms

A **term** is an expression that represents a *data object*. It may be:

variable: string of letters, digits, "_" that starts with an uppercase letter, or with "_"
Examples: X1 Toto12_urt___cur4 _123urc_

numerical constant: usual representations for (signed) integers and floating point numbers
Examples: 14 3.14

(term) constant: an arbitrary atom
Examples: 'Tim Burton' apple fish

compound term: of the form $f(t_1, \ldots, t_n)$ where f is an atom (**function symbol**, **term constructor**) and t_1, ..., t_n are terms
Examples: father('Tim Burton')
cons(3,cons(2,cons(5,nil))) (example of a **list**, see later)

3.3: Atomic Formulas

Atomic formula expression of the form $p(t_1, \ldots, t_n)$
where p is an atom called **predicate** or **relation**, and t_1, ..., t_n are terms.

It can be *true* of *false*.

Example: $\mathsf{release}(\text{'}\mathsf{The\ Blues\ Brothers}\text{'}, \mathsf{france}, 1980)$.

3.3: Atomic Formulas

Atomic formula expression of the form $p(t_1, \ldots, t_n)$
where p is an atom called **predicate** or **relation**, and t_1, ..., t_n are terms.

It can be *true* of *false*.

Example: release('The Blues Brothers', france, 1980).

Caution: do not confuse “atom” (Prolog terminology) with “atomic formula” (traditional logic terminology)!

3.3: Atomic Formulas

Atomic formula expression of the form $p(t_1, \ldots, t_n)$
where p is an atom called **predicate** or **relation**, and t_1, ..., t_n are terms.

It can be *true* of *false*.

Example: release('The Blues Brothers', france, 1980).

Caution: do not confuse "atom" (Prolog terminology) with "atomic formula" (traditional logic terminology)!

Question: how to distinguish a term from an atomic formula?

3.3: Atomic Formulas

Atomic formula expression of the form $p(t_1, \ldots, t_n)$
where p is an atom called **predicate** or **relation**, and t_1, ..., t_n are terms.

It can be *true* of *false*.

Example: $\mathsf{release('The\ Blues\ Brothers', france, 1980)}$.

Caution: do not confuse "atom" (Prolog terminology) with "atomic formula" (traditional logic terminology)!

Question: how to distinguish a term from an atomic formula?

Both term constructors and predicates are arbitrary atoms chosen by the programmer, no distinction possible. It depends on the context!

3.4: Logical constructs

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

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

where H is an atomic formula and φ is a formula.

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

Definite clause: rule of the form $H \leftarrow A_1 \wedge \ldots \wedge A_m$
where $A_1, \ldots A_m$ are *atomic formulas* (no disjunction).

General clause: rule of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$
where $L_1, \ldots L_m$ are *literals*, that is, atomic formulas and negated (discussed later) atomic formulas.
In Prolog, "," is used instead of $\wedge$.

3.5: Predicates again

A predicate is an arbitrary atom defined by the programmer.

The programmer would always want an *arity* (number of arguments) to be associated with a predicate.

In compiler messages, manuals etc., a predicate p with arity n is often denoted as p/n.

Within a program, a predicate p/n "exists" thanks to the fact that it is *defined*, by a set of *rules / facts* of the form:
$p(t_1, \ldots, t_n) \leftarrow \varphi$ or $p(t_1, \ldots, t_n)$.

3.5: Predicates again

A predicate is an arbitrary atom defined by the programmer.

The programmer would always want an *arity* (number of arguments) to be associated with a predicate.

In compiler messages, manuals etc., a predicate p with arity n is often denoted as p/n.

Within a program, a predicate p/n "exists" thanks to the fact that it is *defined*, by a set of *rules / facts* of the form:
$p(t_1, \ldots, t_n) \leftarrow \varphi$ or $p(t_1, \ldots, t_n)$.

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 $\left\{ \begin{array}{c} \psi \leftarrow \varphi_1 \\ \psi \leftarrow \varphi_2 \end{array} \right\}$ and
negated formulas in bodies can also be compiled away (see later).

Predicates are to Prolog what functions / procedures are to functional (or imperative) programming languages.

3.6: 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/goal: a formula of the form $L_1, \ldots, L_m$.

3.7: Exercise

Exercise 3

1. Write a database of the UK royal family (at least 10 relatives of King Charles III) using the predicates `isChildOf`, `male`, and `female`.
2. Write clauses for defining the predicates `siblings`, `isSisterOf`, `isNephewOf(X,Y)`, `isGrandchildOf`, `cousins`.
3. Test your program with queries of the kind `isSisterOf(X,Y)` ...

4: Semantics of Prolog programms

4.1: Semantics of a clause (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:
$\forall D, A(\mathsf{directed}(D, A) \leftarrow \exists M, R(\mathsf{director}(D, M) \wedge \mathsf{cast}(M, A, R)))$

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

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

4.1: Semantics of a clause (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

4.1: Semantics of a clause (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!)

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

is a-priori quantified universally:

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

but

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

The second $\equiv$-step is subtle but correct.

4.2: Declarative semantics

Declarative semantics of pure logic programming:
What does the logic program $\mathbf{P}$ make true?

4.2: Declarative semantics

Digression: What does “making true” mean?

4.2: Declarative semantics

Digression: What does “making true” mean?

In logic, we call a formula C a **logical consequence** of a set of formulas Γ, written $\Gamma \models C$, if every valuation v such that $v(H) = 1$ (“H is true”) for every $H \in \Gamma$, we have $v(C) = 1$ as well.

Digression: What does "making true" mean?

In logic, we call a formula C a **logical consequence** of a set of formulas Γ, written $\Gamma \models C$, if every valuation v such that $v(H) = 1$ ("H is true") for every $H \in \Gamma$, we have $v(C) = 1$ as well.

Example: $\{snow, snow \rightarrow cold\} \models cold$.

"If it is snows and if snow implies cold, then inevitably it is cold".

4.2: Declarative semantics

Digression: What does "making true" mean?

In logic, we call a formula C a **logical consequence** of a set of formulas Γ, written $\Gamma \models C$, if every valuation v such that $v(H) = 1$ ("H is true") for every $H \in \Gamma$, we have $v(C) = 1$ as well.

Example: $\{snow, snow \rightarrow cold\} \models cold$.

"If it is snows and if snow implies cold, then inevitably it is cold".

This notion applies to logic programs, here an example:

$\{\mathsf{mammal}(\mathsf{dog}).,\forall X\ \mathsf{animal}(X) \leftarrow \mathsf{mammal}(X)\} \models \mathsf{animal}(\mathsf{dog})$.

Digression: What does “making true” mean?

In logic, we call a formula C a **logical consequence** of a set of formulas Γ, written $\Gamma \models C$, if every valuation v such that $v(H) = 1$ (“H is true”) for every $H \in \Gamma$, we have $v(C) = 1$ as well.

Example: $\{snow, snow \to cold\} \models cold$.

“If it is snows and if snow implies cold, then inevitably it is cold”.

This notion applies to logic programs, here an example:

$\{\mathsf{mammal}(\mathsf{dog}).,\forall X\ \mathsf{animal}(X) \leftarrow \mathsf{mammal}(X)\} \models \mathsf{animal}(\mathsf{dog})$.

When you are asking a query $\mathsf{animal}(X)$, you are searching for the answer $\mathsf{animal}(\mathsf{dog})$. $\mathsf{animal}(\mathsf{dog})$ is (part of) the semantics of the program.

Now more abstractly ...

4.2: Declarative semantics

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 ideal.)

4.2: Declarative semantics

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 ideal.)

A notion from semantic theory: A set M of ground (not containing variables) atomic formulas such that $M \models \varphi$ for every $\varphi \in \mathbf{P}$ is called a *model* of $\mathbf{P}$. Execution of a logic programming is about extracting atomic formulas from a model of $\mathbf{P}$.

To be a bit more precise, it is about extracting atomic formulas that are true in *every* model of $\mathbf{P}$. This is equivalent to saying that for such an atomic formula, $\mathbf{P} \models A$.

We will have a closer look at this when we look at negation ...

4.3: 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."

4.3: 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 ?

4.3: 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

4.3: 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 clause? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$

4.3: 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 clause? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$
A: try to *unify* (match) the u_i's and the t_i's, and make the appropriate substitutions in φ_p

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

Exercise 4 Use Prolog to test unification for:

- $p(X, g(X, Y, X))$ and $p(2, L)$;
- $p(X, g(X, Y, X))$ and $p(2, g(X, 3, X))$;
- $p(X, g(X, Y, X))$ and $p(2, g(3, Y, 3))$;
- $p(X1, X2, X3, X4, X5, X6, X7, X8)$ and $p(f(X2, X2), f(X3, X3), f(X4, X4), f(X5, X5), f(X6, X6), f(X7, X7), f(X8, X8), f(c, c))$
- $p(g(X))$ and $p(X)$.

4.3: Unification

Exercise 4 Use Prolog to test unification for:

- $p(X, g(X, Y, X))$ and $p(2, L)$;
- $p(X, g(X, Y, X))$ and $p(2, g(X, 3, X))$;
- $p(X, g(X, Y, X))$ and $p(2, g(3, Y, 3))$;
- $p(X1, X2, X3, X4, X5, X6, X7, X8)$ and $p(f(X2, X2), f(X3, X3), f(X4, X4), f(X5, X5), f(X6, X6), f(X7, X7), f(X8, X8), f(c, c))$
- $p(g(X))$ and $p(X)$.

What do you observe for the last example?

4.4: Example search tree

directed('Fatih Akin',A) ?

(14) 'Fatih Akin'$\to D$

director('Fatih Akin',M), cast(M,A,R) ?

(7) 'Soul Kitchen'$\to M$

cast('Soul Kitchen',A,R) ?

(10) 'Adam Bousdoukos'$\to A$
'Zinos Kazantsakis'$\to R$

Answ. 1: A='Adam Bousdoukos'

(11) 'Moritz Bleibtreu'$\to A$
'Illias Kazantsakis'$\to R$

Answ. 2: A='Moritz Bleibtreu'

(12) 'Anna Bederke'$\to A$
'Lucia Faust'$\to R$

Answ. 3: A='Anna Bederke'

4.4: Example search tree

- Backward chaining ; atomic formulas in a conjunction are *proved / executed* from left to right.
- Executing an atomic formula means:
  - replacing it with a condition that makes it true, i.e., unifying it with a clause head and replacing the atomic formula with the appropriate instance of the clause body, which then in turn has to be made 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 / clauses of the program are tried in the order in which they appear in the program.
- Prolog systems use a depth-first search strategy.

4.4: Example search tree

- Backward chaining ; atomic formulas in a conjunction are *proved / executed* from left to right.
- Executing an atomic formula means:
  * replacing it with a condition that makes it true, i.e., unifying it with a clause head and replacing the atomic formula with the appropriate instance of the clause body, which then in turn has to be made 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 / clauses of the program are tried in the order in which they appear in the program.
- Prolog systems use a depth-first search strategy.

Exercise 5 How many leaves have the trees for the queries

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

5: Lists and Recursion

Observations:

- The introduction to the syntax above suggests that atoms used as term constants, term constructors and predicate symbols are completely user-defined.
- In the example programs so far, 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') . . .)

5.1: Lists

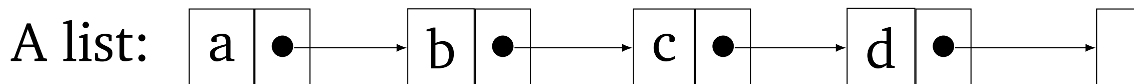
Observations:

- The introduction to the syntax above suggests that atoms used as term constants, term constructors and predicate symbols are completely user-defined.
- In the example programs so far, 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.

5.1: Lists

A list: 

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

5.1: 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: $[1, 2, 3, [-1, \mathsf{a}, [\,]\,], \text{'movie_bd'}]$ $\quad [\,]$ $\quad [_, X, Y, 1]$

We have $[1, 2, 3, 4] = [1, 2, 3|\,[4]\,] = [1|\,[2|\,[3|\,[4|\,[\,]\,]\,]\,]\,]$.

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

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: $[1, 2, 3, [-1, \mathsf{a}, [\,]\,], \mathsf{'movie_bd'}]$ $[\,]$ $[_, X, Y, 1]$

We have $[1, 2, 3, 4] = [1, 2, 3| [4]\,] = [1| [2| [3| [4| [\,]\,]\,]\,]\,]$.

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

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

Exercise 6 Type the query

```
| ?- X = "abcd".
```

How do you interpret the result?

5.2: Lists and Filtering

Having clause heads of the form $p(t_1, \ldots, t_n)$, with arbitrary terms $t_1, \ldots, t_n$ rather than the generic $X_1, \ldots, X_n$, implies that for a given query $p(\ldots)$, some clause may be applicable (unifiable arguments) while another one may not be applicable. This is called **filtering** and has its equivalent in functional programming.

5.2: Lists and Filtering

Having clause heads of the form $p(t_1, \ldots, t_n)$, with arbitrary terms $t_1, \ldots, t_n$ rather than the generic $X_1, \ldots, X_n$, implies that for a given query $p(\ldots)$, some clause may be applicable (unifiable arguments) while another one may not be applicable. This is called **filtering** and has its equivalent in functional programming.

Filtering is often used for lists, e.g. to have a an argument that only matches a non-empty list. Example:

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

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

or simply

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

Exercise 7 Define the predicate $\mathsf{isSecondElmtOf}(X, L)$.

5.3: Recursive programming

Example We wish to retrieve the last element of a list: 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 $[X]$

```
isLastElmtOf(X,[X]).
isLastElmtOf(X,[_|R]) :- isLastElmtOf(X,R).
```

Exercise 8 Write definitions for the following predicates to manipulate lists:

memb/2: such that $\mathsf{memb}(X, L)$ is true if X is element of list L.

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

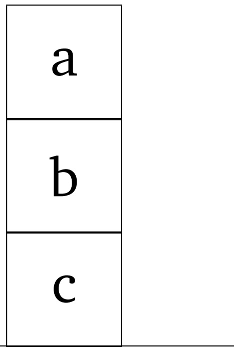
app/3: $\mathsf{app}(L, M, R)$ is true if R is the list that contain the elements of L followed by those of M.

rev/2: $\mathsf{rev}(L, R)$ is true if R is the list that contain the elements of L in reverse order.

6: Negation

6.1: Semantics of negation

Suppose we have a world consisting of three blocks a, b, c, that can be piled upon another, and we are interested in the relation of “being above”. E.g.:



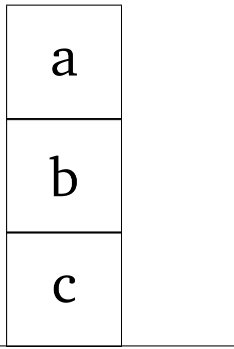
The following Prolog program models this situation:

```
on(a,b).
on(b,c).
above(X,Y) :- on(X,Y).
above(X,Z) :- on(X,Y), above(Y,Z).
```

Exercise 9 Use Prolog execution to calculate the “most natural” *model* of this program.

6.1: Semantics of negation

Suppose we have a world consisting of three blocks a, b, c, that can be piled upon another, and we are interested in the relation of "being above". E.g.:



The following Prolog program models this situation:

```
on(a,b).
on(b,c).
above(X,Y) :- on(X,Y).
above(X,Z) :- on(X,Y), above(Y,Z).
```

Exercise 9 Use Prolog execution to calculate the "most natural" *model* of this program.

The model that seems to describe exactly the picture is this:

```
{ on(a,b), on(b,c), above(a,b), above(b,c), above(a,c) }
```

6.1: Semantics of negation

But there are also other models of this program, e.g., the model in which "everything is true":

```
{ on(a,a), on(a,b), on(a,c),
  on(b,a), on(b,b), on(c,c),
  on(c,a), on(c,b), on(c,c),
  above(a,a), above(a,b), above(a,c),
  above(b,a), above(b,b), above(c,c),
  above(c,a), above(c,b), above(c,c) }
```

Two conclusions

- Definite programs, taken literally, cannot model *negative* information.
- In this example, and elsewhere, humans very often work with a *closed world assumption*: everything that is not explicitly true, is false. How to formalise this?

6.1: Semantics of negation

The *completion* of a program is supposed to have only the "most natural" model:

$\mathsf{on}(X, Y) \leftrightarrow (X = \mathsf{a} \wedge Y = \mathsf{b}) \vee (X = \mathsf{b} \wedge Y = \mathsf{c}).$

$\mathsf{above}(X, Z) \leftrightarrow \mathsf{on}(X, Z) \vee (\mathsf{on}(X, Y) \wedge \mathsf{above}(Y, Z)).$

- This "program" only exists in theory!
- Use $\leftrightarrow$ instead of $\leftarrow$, only one "clause" per predicate, disjunction, no filtering.
- If we use *negation as failure* for negated atomic formulas in queries, with the original program, then the completion gives a semantical reference . . .

6.2: Negation as failure

Procedural meaning given to negation:

> In order to prove $\neg\varphi$, try to prove that φ. If it fails, then $\neg\varphi$ succeeds.

Remark: negation is written \+ is Prolog, and negated goals consisting of several atomic formulas should be enclosed in (()).

Exercise 10 Use Prolog to check whether block a is on itself, and whether block c is on block a.

6.2: Negation as failure

Procedural meaning given to negation:

> In order to prove $\neg\varphi$, try to prove that φ. If it fails, then $\neg\varphi$ succeeds.

Remark: negation is written \+ is Prolog, and negated goals consisting of several atomic formulas should be enclosed in (()).

Exercise 10 Use Prolog to check whether block a is on itself, and whether block c is on block a.

There is actually a clean semantics of this, paraphrased as follows:

> For a ground atomic formula A, if A finitely fails for program $\mathbf{P}$, then $\neg A$ is a logical consequence of the completion of $\mathbf{P}$.

6.2: Negation as failure

Procedural meaning given to negation:

> In order to prove $\neg\varphi$, try to prove that φ. If it fails, then $\neg\varphi$ succeeds.

Remark: negation is written \+ is Prolog, and negated goals consisting of several atomic formulas should be enclosed in (()).

Exercise 10 Use Prolog to check whether block a is on itself, and whether block c is on block a.

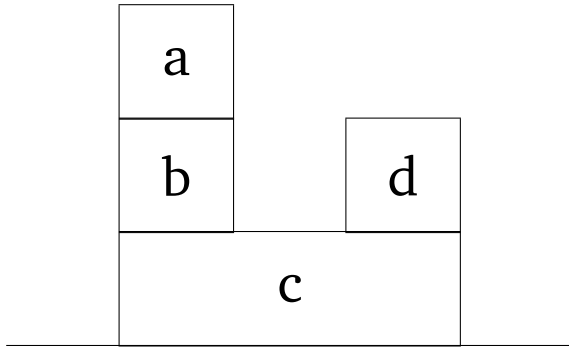
There is actually a clean semantics of this, paraphrased as follows:

> For a ground atomic formula A, if A finitely fails for program **P**, then $\neg A$ is a logical consequence of the completion of **P**.

This even works for non-ground atomic formulas. For example, the query `\+ on(c,X)` succeeds, since the query `on(c,X)` fails. This says: $\forall X.\neg\texttt{on}(c, X)$ is a logical consequence of the completion of the program, since there is no block on block a.

However, this universal quantification is not always what one wants, at least logically ...

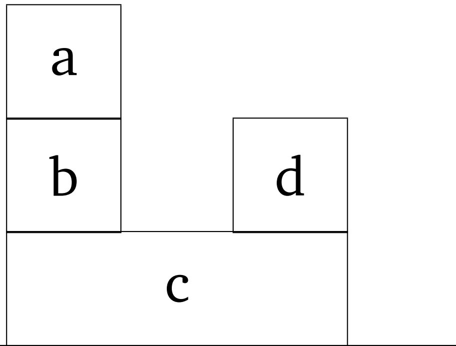
Exercise 11 Consider the following situation:



Modify your program to model this situation and write a query to search for a block on block c, that does not have block a on it.

6.2: Negation as failure

Exercise 11 Consider the following situation:



Modify your program to model this situation and write a query to search for a block on block c, that does not have block a on it.

We observe: The query `on(X,c), \+ on(a,X).` gives the expected answer but `\+ on(a,X), on(X,c).` does not.

Lesson: place calls with negation so that they are *ground* at the time of execution!

Otherwise, you will have no logical reading at all or at least not the one you expect.

Less strict: be sure that when a negated atomic query $\neg A(\bar{X})$ is called, $\forall \bar{X}. \neg A$ is intended.

6.3: Negation in clause bodies

Negation would be very limited if we only allowed it in queries but not in programs, clause bodies to be more precise. But it turns out that allowing for negation in clause bodies again endangers the logical reading of programs:

```
p(X) :- \+ p(X)
```

How can we hope to give a semantics to this program, whose completion would read as follows: $\mathsf{p}(X) \leftrightarrow \neg\mathsf{p}(X)$?

We do not go into the details as to how to avoid this, but at least one simple **lesson**: do not use negation in recursive calls!

6.3: Negation in clause bodies

Negation would be very limited if we only allowed it in queries but not in programs, clause bodies to be more precise. But it turns out that allowing for negation in clause bodies again endangers the logical reading of programs:

```
p(X) :- \+ p(X)
```

How can we hope to give a semantics to this program, whose completion would read as follows: $\mathsf{p}(X) \leftrightarrow \neg\mathsf{p}(X)$?

We do not go into the details as to how to avoid this, but at least one simple **lesson**: do not use negation in recursive calls!

(Although it sometimes works:

```
even(0).
even(s(X)) :-
  \+ even(X).
```

)

6.4: Clean exercises

We start with some exercises where negation is meant to have a clear logical meaning.

Exercise 12 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 Tim Burton.

Exercise 13 Add clauses to the database to define the following:

1. the actors who played in at least two films by Francis Ford Coppola.
2. movies in which played actors who were never directed by Francis Ford Coppola (note that *plausibly*, this would be true for all movies not directed by Francis Ford Coppola; otherwise, there would have to be a movie, not directed by Francis Ford Coppola, such that *all* actors of the movie have already been directed by Francis Ford Coppola).

6.5: Dirty Exercise

Now an exercise that makes some clever use of negation, but without clear logical meaning.

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!"

6.5: Dirty Exercise

How did he know the color of his hat and what color was it?[1]

(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.)

[1] Excerpt from mathsisfun.com

6.5: Dirty Exercise

```
whiteorblack(white).
whiteorblack(black).

possible(A,B,C) :-
  whiteorblack(A), whiteorblack(B), whiteorblack(C),
  \+(( A = white , B = white , C = white )).

%Rear guy can guess his colour C if C is possible and no
%other colour is possible
guess_rear(A,B,C) :-
  possible(A,B,C) , \+((possible(A,B,X) , X \= C )).

%Middle guy can guess the colour B of his hat knowing the
%colour A of the guy before him and knowing that the guy
%behind him could not guess his colour.
guess_middle(A,B) :-
  possible(A,B,_) ,
  \+(( possible(A,X,_) , \+(guess_rear(A,X,_)) , X \= B)).
```

6.5: Dirty Exercise

```
solution(A,B,C) :-
  possible(A,B,C) ,
  \+(guess_rear(A,B,C)) ,
  \+(guess_middle(A,B)).
```

Exercise 14 Calculate some (all) answers for possible, guess_rear, guess_middle, solution using Prolog.

7: More details on running Prolog

7.1: Syntax reminder

Summary (reminder) of syntax:

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.

7.2: 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(cast). 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.

Exercise 15 Using the small movie database, trace the calls to the predicates `cast` and `director` during the execution of the queries:

```
cast(M,A,R), directed('John Landis',A).
cast(M,A,R), \+ directed('John Landis',A).
```

7.3: Printing messages

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

7.4: 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) :-
  q(X), r(X,Z), s(Z,U).
```

should be written

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

Advice: From now on, use anonymous variables in your programs to avoid the "singleton variable..." message.

7.5: Arithmetic

- 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** $\underline{\mathsf{is}}$

- $*$ 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: $X = 4.53\ldots$

7.5: Arithmetic

- 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** <u>is</u>

  - 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: $X = 4.53\ldots$

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`).

Exercise 16 Define a predicate that can compute the value of $n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$.

Exercise 17 Define a predicate `quadEq/4` that computes the solutions of a quadratic equation: `quadEq`(A, B, C, X) should be true if $AX^2 + BX + C = 0$.

Exercise 18

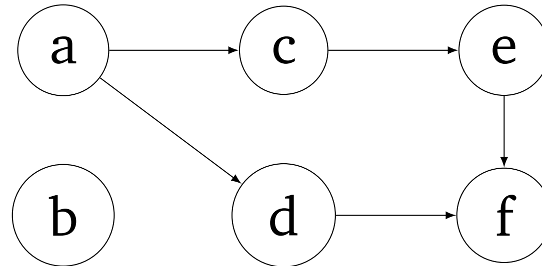
- Define the predicate `len/2` for computing the length of a list.
- Define the predicate `quicksort/2` for sorting a list of numbers.
- Define the predicate `mergesort/2` for sorting a list of numbers.

8: About Prolog search strategy

8.1: Non terminating queries: Retrieval in a graph

Directed graph = binary relation ⇒ predicate edge/2:
$\mathsf{edge}(X, Y)$ is true if there is an edge between from vertex X to Y.

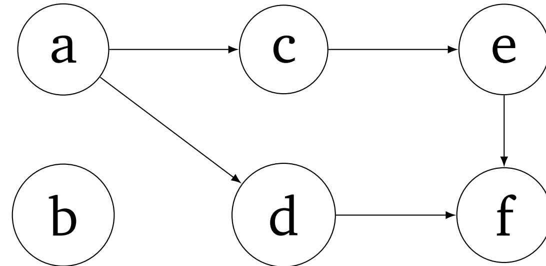
$\mathsf{edge}(a, c).$ $\mathsf{edge}(a, d).$
$\mathsf{edge}(c, e).$ $\mathsf{edge}(e, f).$
$\mathsf{edge}(d, f).$



8.1: Non terminating queries: Retrieval in a graph

Directed graph = binary relation ⇒ predicate edge/2:
$\mathsf{edge}(X, Y)$ is true if there is an edge between from vertex X to Y.

$\mathsf{edge}(a, c)$. $\mathsf{edge}(a, d)$.
$\mathsf{edge}(c, e)$. $\mathsf{edge}(e, f)$.
$\mathsf{edge}(d, f)$.



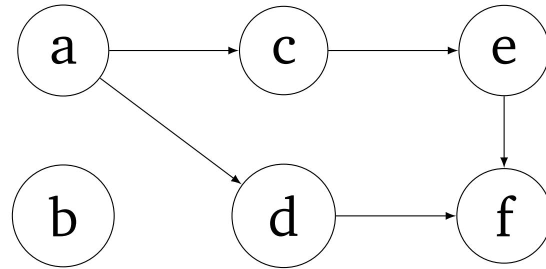
Definition of a predicate path/2, such that $\mathsf{path}(X, Y)$ is true if there is a path, of any length, from X to Y:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Exercise 19 Draw the search trees for the following queries:
$\mathsf{path}(a, e)$? $\mathsf{path}(e, a)$? $\mathsf{path}(a, U)$?

8.1: Non terminating queries: Retrieval in a graph

$\mathsf{edge}(a, c).$ $\mathsf{edge}(a, d).$
$\mathsf{edge}(c, e).$ $\mathsf{edge}(e, f).$
$\mathsf{edge}(d, f).$



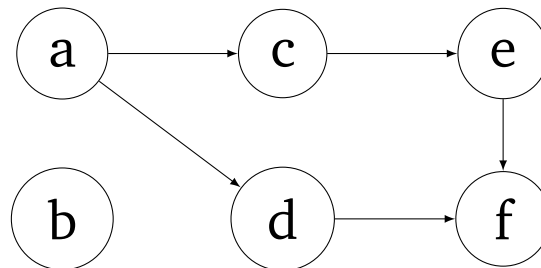
Exercise 20 Now, draw the search trees for the query $\mathsf{path}(a, e)$?

If we re-define path/2 as follows:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(Z,Y), edge(X,Z).
```

8.1: Non terminating queries: Retrieval in a graph

$\mathsf{edge}(a, c)$. $\mathsf{edge}(a, d)$.
$\mathsf{edge}(c, e)$. $\mathsf{edge}(e, f)$.
$\mathsf{edge}(d, f)$.



Exercise 20 Now, draw the search trees for the query $\mathsf{path}(a, e)$?

If we re-define path/2 as follows:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(Z,Y), edge(X,Z).
```

Is it better with the original definition?

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

⇒ In general, it is better to instantiate variables <u>before</u> the recursive call(s).

8.2: 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

8.2: 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 ⇒ store it in a list
⇒ 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:

```
path(X,Y,[X,Y]) :- edge(X,Y).
path(X,Y,[X|P1]) :- edge(X,Z), path(Z,Y,P1).
```

Exercise 21 Draw the search tree for the queries $\mathsf{path}(a, e, P)$, $\mathsf{path}(a, b, P)$ and $\mathsf{path}(a, f, P)$.

8.2: Recursive predicates that construct a list

Exercise 22 On the graph example again.

Question 22.1 What happens if we add an edge from c to a?

Question 22.2 Re-define your predicate path/3, so that it gives “some” paths without looping even if the graph has cycles.
(Hint: use a list of “forbidden” nodes, and an intermediate predicate path/4.)
More difficult: give a solution that enumerates all paths (is complete) when the graph has cycles.

Exercise 23 Define a predicate to compute the length of a list, and another predicate to *generate* a list of a given length.

Exercise 24 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 *pair* of lists of the elements on both sides of the river. For instance, the initial state could be described by $river([\mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{x}], [\,])$, the final state is $river([\,], [\mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{x}])$.

Question 24.1 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.)

The main predicate of your program, plan/1, must be defined so that the query:
$? - \mathsf{plan}(L)$.
yields a list of successive states that lead from the initial state to the final state: $L = [\ \mathsf{river}([\mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{x}], [\,])\ ,\ \mathsf{river}([\mathsf{b}, \mathsf{x}], [\ldots]) \ldots]$.

Question 24.2 Define a predicate step/2 such that $\mathsf{step}(E1, E2)$ is true if it is possible to change from state $E1$ to state $E2$ with a single crossing step of the farmer, with or without one good. (Use the built-in predicate select/3.)

Question 24.3 Define a predicate plan/4 such that $\mathsf{plan}(I, F, P, N)$ 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, $[\mathsf{f}, \mathsf{x}, \mathsf{b}]$ and $[\mathsf{x}, \mathsf{f}, \mathsf{b}]$ may refer to the same state.)

Question 24.4 Finally define plan/1: plan(P) must be true if P is a sequence of states that describe a valid plan from the initial state to the final state.

8.3: Accumulators

Consider the predicate to reverse a list:

reverse$([a, b, c], R) \Rightarrow R = [c, b, a]$

First solution with append: append(L_1, L_2, L_3) is true if $L_3 = L_1.L_2$.

```
rev([],[]).
rev([H|T],L) :-
  rev(T,L2),
  app(L2,[H],L).
```

Number of operations in $O(|L|^2)$.

Can we do better?

8.3: Accumulators

Idea: Through the recursive calls, we should somewhow shuffle the elements starting from the beginning of the first argument, into the second argument, in a *single pass*:

```
rev_test([H|T],A) :-
  rev_test(T,[H|A]).
```

We need a base case, let's simply try:

```
rev_test([],_).
```

Exercise 25

- Try this in Prolog with the query `rev_test([1,2,3],[]).` Is the result informative in any way?

Idea: Through the recursive calls, we should somehow shuffle the elements starting from the beginning of the first argument, into the second argument, in a *single pass*:

```
rev_test([H|T],A) :-
  rev_test(T,[H|A]).
```

We need a base case, let's simply try:

```
rev_test([],_).
```

Exercise 25

- Try this in Prolog with the query `rev_test([1,2,3],[]).` Is the result informative in any way?
- Now try it with the tracer to see that the reversed list is indeed constructed.

The result only appears at the *leaf* of the search tree, it is invisible at the root. How can we carry this solution from the leaf to the root?

8.3: Accumulators

Using an *accumulator*:

```
rev(L,R) :-
  rev_aux(L,R,[]).

rev_aux([],A,A).
rev_aux([H|T],R,A) :-
  rev_aux(T,R,[H|A]).
```

8.3: Accumulators

Using an *accumulator*:

```
rev(L,R) :-
  rev_aux(L,R,[]).

rev_aux([],A,A).
rev_aux([H|T],R,A) :-
  rev_aux(T,R,[H|A]).
```

Number of operations in $O(|L|)$

Remark The predicates member/2, append/3, select/3 and reverse/2 are usually predefined in Prolog.

8.4: 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.

8.4: 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:

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

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

8.4: 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:

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

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

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

8.4: 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:

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

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

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

But if we define a predicate to count backwards:

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

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

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

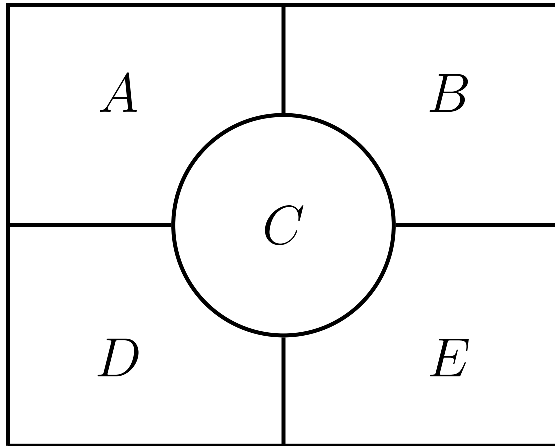
8.4: Last call optimization*

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 clause.

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

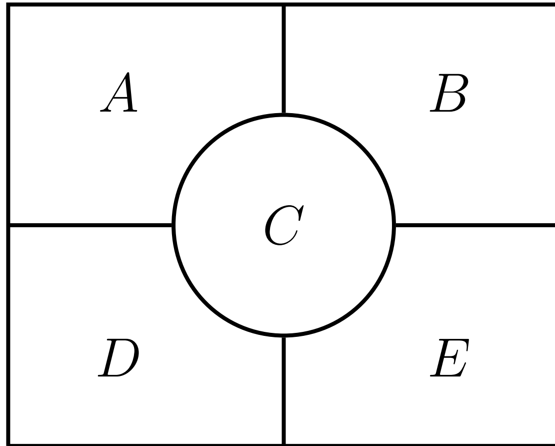
- the recursive call is the last one in the clause, and
- no other clause can be tried after the recursive call
  (the other clauses must therefore in general appear *before* the recursive one
- no more backtrack is possible with the other goals in the clause

8.5: Generate and test



"Map coloring": 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?)

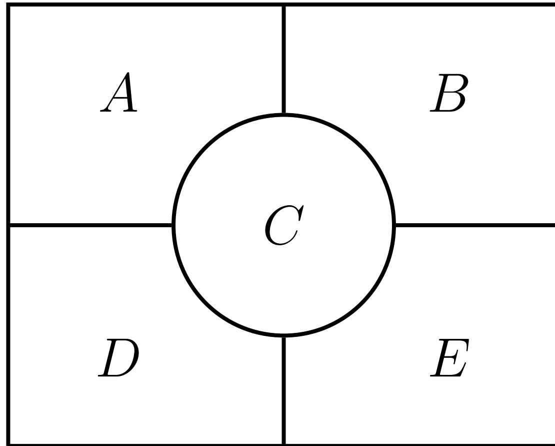
8.5: Generate and test



"Map coloring": 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?)

```
solution(A,B,C,D,E) :- generate(A,B,C,D,E), test(A,B,C,D,E).
generate(A,B,C,D,E) :-
  color(A), color(B), color(C), color(D), color(E).
color(green). color(purple). color(orange).
test(A,B,C,D,E) :- A \= B, A \= C, A \= D, A \= E,
  B \= C, B \= D, B \= E, C \= D, C \= E, D \= E.
```

8.5: Generate and test



"Map coloring": 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?)

```
solution(A,B,C,D,E) :- generate(A,B,C,D,E), test(A,B,C,D,E).
generate(A,B,C,D,E) :-
  color(A), color(B), color(C), color(D), color(E).
color(green). color(purple). color(orange).
test(A,B,C,D,E) :- A \= B, A \= C, A \= D, A \= E,
  B \= C, B \= D, B \= E, C \= D, C \= E, D \= E.
```

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

8.5: Generate and test

Exercise 26 * 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. . .

Exercise 27 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.

Hint: define and use predicates `all_member` and `all_diff`.

8.5: Generate and test

“Generate & test” for Sudoku

	2		
1			
			4
		1	

```
sudoku4(L) :- generate(L) , test(L).
generate(L) :- all_member([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], [1,2,3,4]).
test([X11, X12, X13, X14, X21, X22,..., X42, X43, X44]) :-
  all_diff([X11,X12,X13,X14]), all_diff([X21,X22,X23,X24]) ,
  all_diff([X31,X32,X33,X34]), all_diff([X41,X42,X43,X44]),
  all_diff([X11,X21,X31,X41]), all_diff([X14,X24,X34,X44]),
  all_diff([X12,X22,X32,X42]), all_diff([X13,X23,X33,X43]),
  all_diff([X11,X12,X21,X22]), all_diff([X13,X14,X23,X24]),
  all_diff([X31,X32,X41,X42]), all_diff([X33,X34,X43,X44]).
all_member([],_).
all_member([V|Vs],D) :- member(V,D) , all_member(Vs,D).
all_diff([]).
all_diff([X|L]) :- \+ member(X,L) , all_diff(L).
```

Use with `sudoko4([_,2,_,_,1,_,_,_,_,_,_,4,_,_,1,_]).`

8.6: Exercises*

Exercise 28 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.

8.6: Exercises*

Question 28.1 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 28.2 Define a predicate that can compute the degree of movie separation between two given actors or actresses.

8.6: Exercises*

Exercise 29 The "flights" Prolog database [3] contains Prolog facts with the following information:

- flying times, for instance
  vol(it, 386, blagnac, 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
  tarif(it, toulouse, paris, 500)
  indicates that a ticket to fly from Toulouse to Paris with "it" costs 500€;
- the cost of airport taxes, for instance
  taxe(toulouse,100)
  indicates that every passenger using an airport in Toulouse must pay a 100€ tax every time;
- location of airports, for instance
  aeroport(toulouse, blagnac)
  indicates that Blagnac airport is near Toulouse.

---

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

8.6: Exercises*

Question 29.1 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 29.2 Define a relation directConnection/4 such that connection(A_D, H_D, A_A, H_A) 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 29.3 Define now a relation route/4 such that route(A_D, H_D, A_A, H_A) 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 29.4 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.)

Exercise 30 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$(f, [g, e])$. compone

8.6: Exercises*

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 30.1 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 30.2 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 $U = [e, g, e, h, h, h, d, h, h, h]$.

9: Meta-programming

9.1: 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.

- All solutions to the query $\mathsf{director}(\text{'Woody Allen'}, M)$ are in different branches of Prolog's search tree.
- All branches of the search tree are independent of one another.

9.1: List of solutions

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

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

- call the goal $G(T)$;
- 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:

$\mathsf{findall}(A, \mathsf{directed}(\text{'Fatih Akin'}, A), L)? \Rightarrow$

$L = [\text{'Adam Bousdoukos'}, \text{'Moritz Bleibtreu'}, \text{'Anna Bederke'}]$

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

9.1: List of solutions

bagof*:

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$(A, \text{directed}(D, A), L)? \Rightarrow \ldots$

Exercise 31 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.

10: The GNU Prolog finite domain constraints solver

10.1: The Sudoku 4×4 in Prolog

Recall the Prolog solution:

```
sudoku4(L) :- generate(L) , test(L).
generate(L) :- all_member([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], [1,2,3,4]).
test([X11, X12, X13, X14, X21, X22,..., X42, X43, X44]) :-
  all_diff([X11,X12,X13,X14]), all_diff([X21,X22,X23,X24]) ,
  all_diff([X31,X32,X33,X34]), all_diff([X41,X42,X43,X44]),
  all_diff([X11,X21,X31,X41]), all_diff([X14,X24,X34,X44]),
  all_diff([X12,X22,X32,X42]), all_diff([X13,X23,X33,X43]),
  all_diff([X11,X12,X21,X22]), all_diff([X13,X14,X23,X24]),
  all_diff([X31,X32,X41,X42]), all_diff([X33,X34,X43,X44]).
all_member([],_).
all_member([V|Vs],D) :- member(V,D) , all_member(Vs,D).
all_diff([]).
all_diff([X|L]) :- \+ member(X,L) , all_diff(L).
```

Query:

sudoku4(L).

The search tree for sudoko4([_,2,_,_,1,_,_,_,_,_,_,4,_,_,1,_])has $4^{12} =$ 17 million leaves !!

10.1: The Sudoku 4×4 in Prolog

A more efficient version: Generate only valide lines (somehow, the “generate” and “test” parts are interleaved).

```
all_member_diff([V|Vs],D) :- select(V,D,D1) , all_member_diff(Vs,D1).
all_member_diff([],_).

sudoku4(L)
:- L=[X11,X12,X13,X14,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44]
,  all_member_diff([X11,X12,X13,X14],[1,2,3,4])
,  all_member_diff([X21,X22,X23,X24],[1,2,3,4])
,  all_member_diff([X31,X32,X33,X34],[1,2,3,4])
,  all_member_diff([X41,X42,X43,X44],[1,2,3,4]) ,  nl,  write(L)
,  all_diff([X11,X21,X31,X41]),  all_diff([X11,X21,X31,X41])
,  all_diff([X12,X22,X32,X42]),  all_diff([X13,X23,X33,X43])
,  all_diff([X11,X12,X21,X22]),  all_diff([X13,X14,X23,X24])
,  all_diff([X31,X32,X41,X42]),  all_diff([X33,X34,X43,X44]).
```

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

10.1: The Sudoku 4×4 in Prolog

With the constraint solver:

```
sudoku4_fd(L)
:- L=[X11,X12,X13,X14,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44]
,  fd_domain(L,1,4)
,  fd_all_different([X11,X12,X13,X14])
,  fd_all_different([X21,X22,X23,X24])
,  fd_all_different([X31,X32,X33,X34])
,  fd_all_different([X41,X42,X43,X44])
,  fd_all_different([X11,X21,X31,X41])
,  fd_all_different([X11,X21,X31,X41])
,  fd_all_different([X12,X22,X32,X42])
,  fd_all_different([X13,X23,X33,X43])
,  fd_all_different([X11,X12,X21,X22])
,  fd_all_different([X13,X14,X23,X24])
,  fd_all_different([X31,X32,X41,X42])
,  fd_all_different([X33,X34,X43,X44])
,  fd_labeling(L).
```

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.

10.2: An overview of the constraint solver

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)

10.2: An overview of the constraint solver

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

10.2: An overview of the constraint solver

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.

10.2: An 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
```

10.2: An 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
```

10.2: An 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
```

10.2: An 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
```

10.2: An 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
```

10.2: An 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
```

10.2: An 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
```

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 ensures *local consistency* (it is local to *one* constraint)

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
```

```
| ?- 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
```

10.2: An 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
```

10.2: An 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.

Exercise 32 Consider the following fragment of a program for the famous 8-queens problem:

```
%noattack(Q,Qs,D):
%Q is vertical position of some queen
%Qs represents the vertical positions of all queens of some lower
%fragment of the chessboard, starting from the row D rows beneath the
%row of Q.
noattack(Q,[],_).
...

safe([]).
safe(...) :- noattack(...) , safe(...).

eightqueens(Solution) :-
  Solution = [_,_,_,_,_,_,_,_],
  fd_domain(Solution,1,8),
  safe(Solution),
  fd_labeling(Solution).
```

10.2: An overview of the constraint solver

- Complete the missing parts.
- Add a predicate `nqueens` that generalizes `eightqueens` to an arbitrary number of queens.

10.2: An 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.

10.3: Exercises

Exercise 33 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.

Soleve the problem using the constraint solver.

10.3: Exercises

Exercise 34 * 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

10.3: Exercises

Exercise 35 * 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.