Prolog

PODSTAWY

semestr zimowy 2018/2019

wersja z dnia: 19 listopada 2018
Literatura

- W.F. Clocksin, C.S. Mellish, Prolog. Programowanie, Helion
  (dostępne np. w druku na żądanie, ok. 40 zł)
- D. Diaz, GNU Prolog

Poziom podstawowy
październik: 01, 15, 29
listopad: 19
grudzień: 03, 17
styczeń: 14, 28
luty: 04
Computer and programming

Computer is a machine that can be programmed to perform certain operations:

- computer doesn’t “know” anything
- computer cannot “know” what you want it to do
- computer isn’t “intelligent”

Computer can

- store data
- perform operations in certain order
Programming language defines

- set of words with precise interpretation
- rules how to use these words ("build sentences")

There are few hundreds of programming languages

- they offer different capabilities
- some are better suited for certain tasks
- some are more difficult to learn
- some are faster (in coding and execution)
- different programming paradigms

In principle, one should be able to code every algorithm in every language.
Basic notions used in programming:

- **Algorithm**: method of achieving something by using basic operations, which are directly available in a programming language.

- **(Source) code**: the program. Algorithm coded in some programming language.

- **Compilation**: translation of the code to a form usable for the machine.

- **Variable**: data representation, can be modified.

- **Constant**: data representation, no modifications.

- **String**: data representation in the form of a set of characters interpreted as text.

- **List**: data representation in the form of an ordered set of variables and constants gathered in one object.
Example: sort two numbers in a descending order.
Algorithm: introduce numerical variables \(a, b\). If \(a \geq b\) then
Output\((a, b)\), otherwise Output\((b, a)\).

Code (fortran 77):

```fortran
function sort2(a,b,output)
  real a, b, output(2)
  if (a.ge.b) then
    output(1)=a
    output(2)=b
  else
    output(1)=b
    output(2)=a
  end if
end function
```

Homework: sorting algorithm for three numbers.
Code (prolog):

1   | ?- N=[6,3,1,7,0,4,5,2], sort(N,X).
2
3   N = [6,3,1,7,0,4,5,2]
4   X = [0,1,2,3,4,5,6,7]
5
6   yes

- the sort/2 is a built-in predicate
- we will write our own implementation later
There are two main implementations of prolog:

- GNU prolog (our choice)
- SWI prolog

Two ways of working with prolog (GNU version)

- interactive programming (gprolog)
- compiling the code (gplc)
The GNU prolog interactive console, interrupted with C-c e

$ gprolog
GNU Prolog 1.4.3 (32 bits)
Compiled Apr 11 2013, 10:35:44 with gcc
By Daniel Diaz
Copyright (C) 1999-2012 Daniel Diaz
| ?-
Prolog interruption (h for help) ? h
    a abort          b break
    c continue       e exit
    d debug          t trace
h/? help
Writing and loading a program:

- one-liners
- loading external code from a file

```prolog
| ?- consult('add.pl').
compiling /home/marek/add.pl for byte code...
/home/marek/add.pl compiled,
    2 lines read – 558 bytes written, 4 ms
```

(1 ms) yes

- writing the code directly in the console (C-d to finish)

```prolog
| ?- consult(user).
compiling user for byte code...
add(A,B,C) :- C is A+B.
```

user compiled, 3 lines read – 407 bytes written, 37736 ms

(2 ms) yes
Prolog stands for Programming in Logic. It is a declarative language of programming using logic as its basis. It was described in the 1972 by Alain Colmerauer (ISO standards in 1995 and 2000) and used since then in the areas of

- relational databases
- mathematical logic
- abstract problems
- natural languages
- automation
- symbolic algebra
- biochemical structures
- artificial intelligence
Prolog uses **terms** which are bound by **ordered relations**.

Example:

- the terms may be: jas, malgosia
- the relations may be: brat, siostra
- the relations are ordered, i.e., jas is in relation brat with malgosia, but in general not the opposite way round

A prolog program consists of 3 parts:

1. declaration of **facts** (database of objects and relations)
2. definition of **rules** that allow to manipulate the facts
3. declaration, what is the problem (question)
Example of a set of facts:

```
rodzenstwo(jas, malgosia).
rodzenstwo(malgosia, jas).
brat(jas, malgosia).
siostra(jas, malgosia).
```

Notice:
- there is no need to specify, what are jas, brat etc.
- definition through properties
- `predicate(argument1, argument2)`. is a relation (statement)
- all constants are written in small letters
- definition of a relation has a dot . at the end

We have here definition of 3 predicates (rodzenstwo/2, brat/2, siostra/2) and two constants (jas, malgosia).
• predicate defines a relation
• argument is a **simple term** (i.e., a constant or a variable)
• a **compound term** consists of simple terms connected with **functors** (and, or)

The relations are defined in an abstract way: \( x(a, b) \), but names usually suggest certain interpretation:

```prolog
1  x(a, b).
2  kolor(czerwony, papryka).
3  ojciec(zdzich, eustachy).
4  wieksze(trzy, cztery).
```
each constant represents a stand-alone object

- different constants represent always different objects...
- ... even if the property of the object (value stored under this constant) is the same!

Example in fortran

```
1 x=2
2 y=2
3 if (x.eq.y) then ...
```

Example in prolog

```
| ?- x=2, y=2, x=y.

no
```

In fact, even x=2 is a no in prolog.
A **variable** in prolog works not exactly the same way as in other languages

- it has no type in the moment of initialization
- the first statement that successfully sets the variable to a certain value fixes this value once and for all
- for example, a statement of the form `if X = 2 then...` will not set `X` to 2 in fortran, but its analog in prolog will
- variables are written in capital letters `X`, `Kuba`...

*) **Encountering `X=2`, prolog sees it as a logical statement which has to be true to complete the program successfully. Therefore, if `X` is not set yet, this statement can be satisfied by substituting 2 for `X`.**
Variables in prolog are used to

- ask questions, because the answer must be placed somewhere
- act as accumulator (temporary storage place) in recurences

```
kolor(czerwony, auto).
kolor(zielony, ufoludek).

| ?- kolor(X, auto).
X = czerwony
yes
```
Basic logical functors

,: AND
;; OR
;— IF

Series of conditions are read and interpreted from left to right. Ordering matters.
Example:

```
lata(samolot).
lata(mucha).
kolor(czarny,mucha).
kolor(czarny,samolot).

| ?- lata(X), kolor(czarny,X).
X = samolot  a
X = mucha  no

| ?- kolor(czarny,X), lata(X).
X = mucha  a
X = samolot  yes
```
Previous example showed, that if **more than one solution exists**, prolog will find all of them (one can explicitly turn this off; we will come back to this later).
The final *yes* and *no* answers come from the order in which `lata/1` and `kolor/2` predicates are defined (we will not discuss this issue here).
To find more than one solution, prolog has to do **backtracking**, i.e., it goes back and tries to find another solution(s).
SUMMARY

- **constants** (atom or number)
- **variable** (assigned [once and for all] or unassigned)
- **term**
  - simple: constant, variable
  - compound: simple terms connected by functors or operators
- **predicate** (relation)
- **functor, operator** (defined or built-in)
- **clauses**
  - facts (statements)
  - rules (IF statements)
  - queries (questions)
Functors connect simple terms to build compound ones. They can have the form of

- predicates
- operators

Notation

- infix: 3*4, 3 pomnoz 4
- prefix: * 3 4, *(3,4), pomnoz(3,4)

Operators have different priorities of execution, and can be left- or right-associative, so that maths can work.
Unification

Unification is realized by the functor =.

Properties of unification:

- two terms can be unified if they are the same (direct matching)

```
| ?- A=A.  
yes

| ?- a=a.  
yes

| ?- a=b.  
no
```
two terms can be unified if they can be made the same by instantiation of variables

```
| 1 | ?- A=a.          |
|   | A = a            |
|   | yes              |

| 4 | ?- A=a, A=B.     |
|   | A = a            |
|   | B = a            |
|   | yes              |
```

unification of two uninstantiated variables is always true

```
| 1 | ?- A=B.          |
|   | A = B            |
|   | yes              |
```
Additionally:

- unification is a special case of comparison of terms
- atoms and numbers can be easily compared by standard rules
- unification is not an instantiation of variables (we do not assign a value to the variable *directly*) but...
- ... it may result in instantiation as a condition to make the unification possible
- its result is true or false
Example 1:

<table>
<thead>
<tr>
<th>?- data(1,kwie,2014) = data(1,kwie,2014).</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
</tr>
</tbody>
</table>

- terms identical

Example 2a:

<table>
<thead>
<tr>
<th>?- data(1,kwie,2014) = data(D,kwie,2014).</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes, because 1 can be assigned to D</td>
</tr>
</tbody>
</table>

Example 2b:

<table>
<thead>
<tr>
<th>?- D=1, data(1,kwie,2014) = data(D,kwie,2014).</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes, because D=1</td>
</tr>
</tbody>
</table>
Example 3:

```prolog
(... somewhere in the code ...)
data(\text{Day1}, \text{kwie}, 2014) = \text{data(\text{D2}, \text{kwie}, 2014)}.\]
```

will be yes if:

- Day1 and D2 instantiated and have the same value
- one of them not instantiated and can be unified with the other (becomes instantiated after this operation)
- both are uninstantiated (remain that way but bound with each other)
Example 4: nested unification of (abstract) terms

| 1 | ?- b(X,a)=b(f(Y),Y), |
| 2 | d(f(f(a)))=d(U), |
| 3 | c(X)=c(f(Z)). |

U = f(f(a))
X = f(a)
Y = a
Z = a
yes

More about unification and comparing terms (manual sec. 8.2, 8.3)
# Functors of unification and comparison of terms

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>true, if unification possible</td>
</tr>
<tr>
<td>=</td>
<td>true, if unification not possible</td>
</tr>
<tr>
<td>==</td>
<td>true, if comparison positive</td>
</tr>
<tr>
<td>==</td>
<td>true, if comparison negative</td>
</tr>
<tr>
<td>@&lt;</td>
<td>true, if term smaller</td>
</tr>
<tr>
<td>@=&lt;</td>
<td>true, if term smaller equal</td>
</tr>
<tr>
<td>@&gt;</td>
<td>true, if term greater</td>
</tr>
<tr>
<td>@&gt;=</td>
<td>true, if term greater equal</td>
</tr>
</tbody>
</table>

- comparison of terms of the same type
- lexical order used
- the result is true or false
Difference between = and ==

- = is unification: true if terms can be unified (instantiation of variables may occur)
- == is identity: true if terms (unaltered) are identical

```
| ?- X='Reksio'.
X = Reksio
yes

| ?- X=='Reksio'.
no

| ?- X='Reksio', X=='Reksio'.
X = Reksio
yes
```
Comparison of strings based on ASCII codes:

1  | ?- X="Reksio", X=="Reksio".
   X = [82,101,107,115,105,111]
   yes

5  | ?- X=="Reksio".
   no

and lexical order:

1  | ?- "Reksio" @> "reksio".
   no

4  | ?- "reksio" @> "Reksio".
   yes
Numerical versus lexical comparison of terms

```
| ?- X=55, Y=6, X<Y.  
no

| ?- X="55", Y="6", X<Y. 
uncaught exception: 
error(type_error(evaluable,’,’/2),(<)/2)

| ?- X="55", Y="6", X@<Y. 
X = [53,53]  
Y = [54]  
yes
```
Term types (for more: manual, sec. 6.3.4, 8.1)

- variable
- constant – abstract
- constant – number (integer, float)
- constant – string

An atom term is a constant or a string.
An atomic term is an atom or a number.
Predicates which check term types:

- **var(T)**: true, if $T$ a variable
- **nonvar(T)**: true, if $T$ not variable
- **atom(T)**: true, if $T$ an atom
- **atomic(T)**: true, if $T$ an atom or number
- **number(T)**: true, if $T$ a number
- **compound(T)**: true, if $T$ not simple

**Examples:** variables

1. | ?- var(A). |
   2. yes

1. | ?- var(2). |
   2. no

1. | ?- var(a). |
   2. no

1. | ?- var('tekst'). |
   2. no
**Examples:** atoms

1. `| ?- atom(A).`
   - no
2. `| ?- atom(a).`
   - yes
3. `| ?- atom(2).`
   - no
4. `| ?- atom('tekst').`
   - yes

**Examples:** atomic terms

1. `| ?- atomic(A).`
   - no
2. `| ?- atomic(a).`
   - yes
3. `| ?- atomic(2).`
   - yes
4. `| ?- atomic('tekst').`
   - yes
Arithmetics (for more see manual, sec. 8.6)

=:= equal
=\= not equal
< less than
=< not greater than
> greater than
>= not smaller than

+ plus
- minus
* times
/ divided by
// integer division
mod modulo division

The infix functor is substitutes under its left argument the arithmetic expression from its right

1 srednia(A,B,S) :- S is (A+B)/2.
2 1 is mod(7,2) /* true */
3 3 is 7//2 /* true */
4 3.5 is 7/2 /* true */
Built-in mathematical functions and operations include

- power, square root
- bitwise and, or, xor, not
- absolute value, sign
- trigonometric, inverse trigonometric and hyperbolic functions
- exponent, logarithm

So, what is more: three large (32cm) or two XXL (40cm) pizzas?

```prolog
area(R,A) :- A is (pi * R^2).

?- area(16,A), C is 3*A, area(20,B), D is 2*B.
```

```
| A = 804.24771931898704 |
| B = 1256.6370614359173 |
| C = 2412.743157956961 |
| D = 2513.2741228718346 |
```

`yes`
Other simple examples:

1 | ?- X is sin(pi/2).

    X = 1.0

    yes

1 | ?- X is sin(3*pi/2).

    X = -1.0

    yes
How to generate the number $\pi$ using trigonometric functions:

```
| ?- X is 4.0*atan(1.0).
```

```
X = 3.1415926535897931
```

```
yes
```

The result is “exact”:

```
| ?- X is 4.0*atan(1.0) - pi.
```

```
X = 0.0
```

```
yes
```
SUMMARY

- Functors
  - unification of terms
  - comparison of terms (numbers, character strings)
- types of terms
  - simple, compound, variable, constant (number, string)
  - atom = constant or string
  - atomic = constant or string or number
- simple arithmetics (functor is)
**Prolog programs**

During the execution of a program prolog:

- consults the definitions of rules,
- jumps between the query, the definitions, and the statements,
- sets choice points in the statements, to resume the search from the previous point.

It is difficult to track exactly what is prolog doing, but you can use the tracer, which is built-in in the gprolog console.
Example 1.1. Linear chain

pocket → suitcase → backpack → box → wardrobe

1. smaller(pocket, suitcase).
2. smaller(suitcase, backpack).
3. smaller(backpack, box).
4. smaller(box, wardrobe).

5. is_smaller(X,Y) :- smaller(X,Y).

Consider the queries:

1. | ?- is_smaller(backpack, box).

1. | ?- is_smaller(backpack, X).
| ?- is_smaller(backpack, box).
1  1  Call: is_smaller(backpack, box) ?
2  2  Call: smaller(backpack, box) ?
2  2  Exit: smaller(backpack, box) ?
1  1  Exit: is_smaller(backpack, box) ?
yes

{trace}

| ?- is_smaller(backpack, X).
1  1  Call: is_smaller(backpack, _23) ?
2  2  Call: smaller(backpack, _23) ?
2  2  Exit: smaller(backpack, box) ?
1  1  Exit: is_smaller(backpack, box) ?
X = box
yes

{trace}
Example 1.2. Linear chain

pocket → suitcase → backpack → box → wardrobe

1 smaller(pocket, suitcase).
2 smaller(suitcase, backpack).
3 smaller(backpack, box).
4 smaller(box, wardrobe).

5 is_smaller(X,Y) :- smaller(X,Y);
6 smaller(X,Z), smaller(Z,Y).

Consider the queries:

| ?- is_smaller(backpack, box).
| ?- is_smaller(backpack, X).
Trace:

?- is_smaller(backpack, box).
1 1 Call: is_smaller(backpack, box) ?
2 2 Call: smaller(backpack, box) ?
2 2 Exit: smaller(backpack, box) ?
1 1 Exit: is_smaller(backpack, box) ?
true ? ;

1 1 Redo: is_smaller(backpack, box) ?
2 2 Call: smaller(backpack, _87) ?
2 2 Exit: smaller(backpack, box) ?
3 2 Call: smaller(box, box) ?
3 2 Fail: smaller(box, box) ?
1 1 Fail: is_smaller(backpack, box) ?
(1 ms) no
{trace}
Lines 2–6

- definition of `is_smaller` is consulted
- the first part of the definition is considered
- the database is searched from the top
- search successful on database’s line 3
- `true ?` – this is not the end, because there is still the second part of the definition of `is_smaller`
Lines 8–14

- definition of is_smaller is consulted again (hence Redo)
- the second part of the definition is considered
- Z variable is here represented internally as _87
- finding solution for
  \[ \text{smaller(backpack,Z), smaller(Z,box)} \]
- again, at database’s line 3 the first part can be solved:
  \[ \text{smaller(backpack,box), so Z=box} \]
- the second part yields: \[ \text{smaller(box,box)} \]
- this cannot be solved
- returning to \[ \text{smaller(backpack,Z)} \], but no such a Z can be found in the database
- no – program terminates
Trace:

?- is_smaller(backpack,X).
1 1 Call: is_smaller(backpack,_23) ?
2 2 Call: smaller(backpack,_23) ?
2 2 Exit: smaller(backpack,box) ?
1 1 Exit: is_smaller(backpack,box) ?
X = box ? ;

1 1 Redo: is_smaller(backpack,box) ?
2 2 Call: smaller(backpack,_92) ?
2 2 Exit: smaller(backpack,box) ?
3 2 Call: smaller(box,_23) ?
3 2 Exit: smaller(box,wardrobe) ?
1 1 Exit: is_smaller(backpack,wardrobe) ?
X = wardrobe
yes
{trace}
Lines 2–6

- the variable X is represented here by _23
- definition of is_smaller is consulted
- the first part of the definition is considered
- the database is searched from the top
- search successful on database’s line 3: X=box

Lines 8–14

- return to the last call and resume the search
- definition of is_smaller is consulted
- the second part of the definition is considered
- looking for Z (here _92) such that
  - smaller(backpack,Z), smaller(Z,X)
- database’s line 3: Z=box
- looking for X such that is_smaller(box,X)
- database’s line 4: X=wardrobe
- yes: no more possibilities, the definition of is_smaller finishes, program terminates
Example 1.3. Linear chain

pocket → suitcase → backpack → box → wardrobe

Let’s do it the proper way:

smaller(pocket, suitcase).
smaller(suitcase, backpack).
smaller(backpack, box).
smaller(box, wardrobe).

is_smaller(X, Y) :- smaller(X, Y);
smaller(X, Z), is_smaller(Z, Y).

Consider the queries:

?- is_smaller(backpack, box).

?- is_smaller(backpack, X).

?- is_smaller(X, backpack).
Trace (1st part):

<table>
<thead>
<tr>
<th>Line</th>
<th>Call/Exit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Call</td>
<td>is_smaller(backpack, box) ?</td>
</tr>
<tr>
<td>2</td>
<td>Call</td>
<td>smaller(backpack, box) ?</td>
</tr>
<tr>
<td>2</td>
<td>Exit</td>
<td>smaller(backpack, box) ?</td>
</tr>
<tr>
<td>1</td>
<td>Exit</td>
<td>is_smaller(backpack, box) ?</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td>? ;</td>
</tr>
</tbody>
</table>

- the first part is standard and follows directly from database’s line 3
- the execution continues with the second part of the definition of is_smaller
Trace (2nd part):

1  1  Redo: is_smaller(backpack, box) ?
2  2  Call: smaller(backpack, _86) ?
3  2  Exit: smaller(backpack, box) ?
4  2  Call: is_smaller(box, box) ?
5  3  Call: smaller(box, box) ?
6  3  Call: smaller(box, _135) ?
7  3  Exit: smaller(box, wardrobe) ?
8  3  Call: is_smaller(wardrobe, box) ?
9  4  Call: smaller(wardrobe, box) ?
10  4  Fail: smaller(wardrobe, box) ?
11  4  Call: smaller(wardrobe, _184) ?
12  4  Fail: smaller(wardrobe, _172) ?
13  5  Fail: is_smaller(wardrobe, box) ?
14  2  Fail: is_smaller(box, box) ?
15  1  Fail: is_smaller(backpack, box) ?
16  no
17  {trace}
(l.1) resume from the last point

try to solve
smaller(backpack,Z), is_smaller(Z,box)

(l.3) first choice Z=box, satisfies smaller(backpack,box), jump to definition of is_smaller

(l.4) is_smaller(box,box): first check smaller(box,box) (fails, l.6), then look for Z0 such that:
smaller(box,Z0), is_smaller(Z0,box)

(l.8) first choice for Z0 is wardrobe, so check
is_smaller(wardrobe,box), jump to definition of is_smaller (l.9,10)

is_smaller(wardrobe,box) ultimately fails (l.14), so
is_smaller(box,box) fails as well (l.15), and so does the initial query

no – no more solutions available
Trace (1st part): two solutions found

?- is_smaller(backpack,X).
1 1  Call: is_smaller(backpack,_24) ?
2 2  Call: smaller(backpack,_24) ?
2 2  Exit: smaller(backpack,box) ?
1 1  Exit: is_smaller(backpack,box) ?
X = box ? a

1 1  Redo: is_smaller(backpack,box) ?
2 2  Call: smaller(backpack,_93) ?
2 2  Exit: smaller(backpack,box) ?
3 2  Call: is_smaller(box,_24) ?
4 3  Call: smaller(box,_24) ?
4 3  Exit: smaller(box,wardrobe) ?
3 2  Exit: is_smaller(box,wardrobe) ?
1 1  Exit: is_smaller(backpack,wardrobe) ?
X = wardrobe
Trace (2nd part): no more solutions available

1  1  Redo: is_smaller(backpack, wardrobe) ?
3  2  Redo: is_smaller(box, wardrobe) ?
4  3  Call: smaller(box, _142) ?
4  3  Exit: smaller(box, wardrobe) ?
5  3  Call: is_smaller(wardrobe, _24) ?
6  4  Call: smaller(wardrobe, _24) ?
6  4  Fail: smaller(wardrobe, _24) ?
6  4  Call: smaller(wardrobe, _191) ?
6  4  Fail: smaller(wardrobe, _179) ?
5  3  Fail: is_smaller(wardrobe, _24) ?
3  2  Fail: is_smaller(box, _24) ?
1  1  Fail: is_smaller(backpack, _24) ?
(3 ms) no
{trace}
Trace (1st part): two solutions available

1  |     is_smaller(X,backpack).
2  1  Call: is_smaller(_23,backpack) ?
3  2  Call: smaller(_23,backpack) ?
4  2  Exit: smaller(suitcase,backpack) ?
5  1  1  Exit: is_smaller(suitcase,backpack) ?
6  X = suitcase ? a

1  1  Redo: is_smaller(suitcase,backpack) ?
2  2  Redo: smaller(suitcase,backpack) ?
2  2  Fail: smaller(_23,backpack) ?
2  2  Call: smaller(_23,_92) ?
2  2  Exit: smaller(pocket,suitcase) ?
3  2  Call: is_smaller(suitcase,backpack) ?
4  3  Call: smaller(suitcase,backpack) ?
4  3  Exit: smaller(suitcase,backpack) ?
4  3  Exit: is_smaller(suitcase,backpack) ?
1  1  Exit: is_smaller(pocket,backpack) ?
18 X = pocket
Trace (2nd part, truncated): no more solutions available

1 1 Redo: is_smaller(pocket,backpack) ?
2 3 2 Redo: is_smaller(suitcase,backpack) ?
3 4 3 Call: smaller(suitcase,_141) ?
4 4 3 Exit: smaller(suitcase,backpack) ?
5 5 3 Call: is_smaller(backpack,backpack) ?
6 6 4 Call: smaller(backpack,backpack) ?
7 6 4 Fail: smaller(backpack,backpack) ?
8 6 4 Call: smaller(backpack,_190) ?
9 6 4 Exit: smaller(backpack,box) ?
10 7 4 Call: is_smaller(box,backpack) ?
11 8 5 Call: smaller(box,backpack) ?
12 8 5 Fail: smaller(box,backpack) ?
13 8 5 Call: smaller(box,_239) ?
14 8 5 Exit: smaller(box,wardrobe) ?
15 9 5 Call: is_smaller(wardrobe,backpack) ?
16 10 6 Call: smaller(wardrobe,backpack) ?
17 10 6 Fail: smaller(wardrobe,backpack) ?
18 10 6 Call: smaller(wardrobe,_288) ?
19 10 6 Fail: smaller(wardrobe,_276) ?
20 9 5 Fail: is_smaller(wardrobe,backpack) ?
21 7 4 Fail: is_smaller(box,backpack) ?
22 5 3 Fail: is_smaller(backpack,backpack) ?
23 3 2 Fail: is_smaller(suitcase,backpack) ?
24 2 2 Redo: smaller(pocket,suitcase) ?
25 2 2 Exit: smaller(suitcase,backpack) ?
26 (...) 4 3 Fail: smaller(wardrobe,backpack) ?
27 4 3 Call: smaller(wardrobe,_141) ?
28 4 3 Fail: smaller(wardrobe,_129) ?
29 3 2 Fail: is_smaller(wardrobe,backpack) ?
30 1 1 Fail: is_smaller(_23,backpack) ?
31 (7 ms) no
32 {trace}
**Homework:** Consider a slightly modified version of this program

```
smaller(pocket, suitcase).
smaller(suitcase, backpack).
smaller(backpack, box).
smaller(box, wardrobe).

is_smaller(X,Y) :- smaller(X,Y);
               is_smaller(X,Z), smaller(Z,Y).
```

The simple query breaks the prolog console:

```
| ?- is_smaller(backpack, box).
  true ? ;

Fatal Error: local stack overflow
(size: 16384 Kb, reached: 16384 Kb,
environment variable used: LOCALSZ)
```

Why?
Example 2.1 Graph (tree), categories in nodes

- the root is CHARACTER
- the branches (nodes) are FUN and SERIOUS
- the 2nd rank nodes are SMART, SILLY, LUCKY
- the leaves are morris, marlena...
The same information is contained in this graph, if \textsc{fun/serious} and \textsc{smart/silly/lucky} are on the same footing, so the order doesn’t matter.

\begin{center}
\begin{tikzpicture}
\node at (0,0) {\textbf{SMART}};
\node at (2,0) {\textbf{SILLY}};
\node at (4,0) {\textbf{LUCKY}};
\node at (0,-1) {\textbf{FUN}};
\node at (0,-2) {\textbf{SERIOUS}};
\node at (2,-1) {\textbf{FUN}};
\node at (2,-2) {\textbf{SERIOUS}};
\node at (4,-1) {\textbf{FUN}};
\node at (4,-2) {\textbf{SERIOUS}};
\node at (0,-3) {\textbf{morris}};
\node at (0,-4) {\textbf{bulgot}};
\node at (0,-5) {\textbf{mort}};
\node at (2,-3) {\textbf{rico}};
\node at (2,-4) {\textbf{julian}};
\node at (2,-5) {\textbf{skipper}};
\node at (4,-3) {\textbf{marlena}};
\node at (4,-4) {\textbf{kowalski}};
\node at (4,-5) {\textbf{szeregowy}};
\end{tikzpicture}
\end{center}
The root definition for graph 1

```prolog
character(X) :- fun(X); serious(X).
```

The root definition for graph 2

```prolog
character(X) :- smart(X); silly(X); lucky(X).
```

Other information will be inferred directly from the database. This is not the most efficient way of using a graph.
Both graphs will be described in the same way, directly in the database:

smart(marlena).
smart(morris).
...
silly(mort).
...
fun(marlena).
fun(morris).
fun(mort).
...

Example of queries:

| ?- fun(X).
| ?- fun(X), lucky(X).
Example 2.2 Graph, categories in the structure of the graph
Basic information required in the database is of the form

```
male(abraham).
male(clancy).
...
female(mona).
female(jackie).
...
parents(abraham,mona,herb).
parents(abraham,mona,homer).
parents(clancy,jackie,marge).
...
```

so we are coding certain properties of the entries (male/1, female/1) and the structure of the graph (parents/3). This is enough to describe the whole structure of the graph.
Now, one can define further properties (i.e. family relations in this example) using the basic definitions.

**Example 2.2.1.** X is the sister of Y:

```prolog
sister(X,Y) :- parents(F,M,X), parents(F,M,Y), female(X).
```

**Example 2.2.2.** X is the mother of Y:

```prolog
mother(X,Y) :- parents(_,X,Y).
```

**Example 2.2.3.** X is the grandmother of Y:

```prolog
gmother(X,Y) :- mother(X,Z),
               (parents(_,Z,Y); parents(Z,_,Y)).
```
SUMMARY

- prolog reads the program from left to right, from top to bottom
- when jumps required, prolog remembers the point, from which it resumes
- this applies also to the choice points, i.e., the last choices (from the database) for variables
- sometimes the choice points are erased, sometimes moved during the execution of the program
- the structure of the data can be linear (chain)
- the structure of the data can have the form of a graph (tree)
  - properties encoded in the nodes
  - properties encoded in the structure
A prolog program that does something
Conversion between Celsius and Fahrenheit degrees.

run :- write('Podaj temp. w st. Celsjusza '),
        read(C), convert(C,F),
        write('Temp. to '), write(F),
        write('st. Fahrenheita'), nl,
        warning(F,Komunikat), write(Komunikat).

convert(C,F) :- F is 9.0/5.0*C + 32.

warning(T,'Ale upal!') :- T > 90.
warning(T,'Ale ziab!') :- T < 30.
warning(T,'') :- T >= 30, T =< 90.
New predicates (more on I/O during the extended course):

- `write(T), write('text')`
- `read(T)`
- `nl`

Notice:

- the predicate `run` has no argument (`run/0`) and contains the “main function” of the program
- `convert` and `warning` are called from within the `run` predicate like subroutines
- the program runs once and terminates after the first execution
Examples:

1 | ?- T=4, write(T).
2   4
3   T = 4
4   yes
5
6 | ?- write(T), T=4.
7   _23
8   %order of execution!
9   T = 4
10  yes

The order of predicates is important, write/1 acts immediately, printing on the screen as a side-effect.
Example of a simple interaction with the user:

```
| :- write('Podaj T: '), read(T), number(T),
  W is T+1, write('W='), write(W).

Podaj T: 5.
W=6

T = 5
W = 6
(1 ms) yes
```
Problem: make the program more interactive; let it keep working, until we close it.

```prolog
run :- data(C), loop(C).

data(C) :- write('Temp. w st. Celsiusza? '), read(C).

loop(C) :- number(C), convert(C,F),
write('Temp. to '), write(F),
write('st. Fahrenheit!'), nl,
warning(F,K), write(K), nl,
data(C1), loop(C1).

loop(quit).
```

with the previous definitions of convert/2 and warning/2.
**Question:** Will the program work in the same way, if one interchanges the loop statements, i.e.

```
loop(quit).
loop(C) :- ...
```

and why?
Program: general schemes

I Questions based

1. database
2. definitions
3. queries

II One-timer

1. main function (with zero or more arguments), containing sequence of predicates
2. definitions

III Interactive

1. main function (with zero or more arguments), containing sequence of predicates
2. one of it calls itself (main loop of the program)
3. definitions
Lists

A generic **list** is a data structure:

- one-dimensional ("chain", "vector", one parameter needed to point to the element of the list)
- ordered sequence of values
- always finite, with length statically declared or dynamically changing
- the same value may appear more than once
- values must (or not – depending on the implementation) be of the same type (*i.e.* only numbers, only strings..., or mixed types)
In Prolog: two equivalent notations to describe a list

- \([a, b, c]\) is the user-friendly notation
- prefix functor \(\cdot\) (dot) takes two arguments — a term and a list — and adds the term on the beginning of the list

\[
\text{\texttt{| ?- L = \texttt{[a,b,c]}, M = \texttt{\cdot(a,\cdot(b,\cdot(c,\texttt{[]}))}), L=M.}}
\]

\[
\text{L = \texttt{[a,b,c]}}
\]
\[
\text{M = \texttt{[a,b,c]}}
\]

\[
\text{yes} \quad /* \text{L and M can be unified} */
\]

- \([\texttt{[]}\) is an empty list
The first element of the list is its **head**, the rest is its **tail**. This is written as \([H \mid T]\).

Example (notice the alphabetical order H L T in prolog’s response):

```
| ?- L = [a,b,c], L = [H|T].

H = a
L = [a,b,c]
T = [b,c]

yes
```

The following notations are equivalent:

- \([a,b,c]\)
- \([a\mid[b,c]]\)
- \([a,b\mid[c]]\)
- \([a,b,c\mid[]]\)
More examples:

- an empty list has no head/tail

```prolog
1 | ?- []=[H|T].
2  No
```

- one-element list has the head only, tail is an empty list

```prolog
1 | ?- [1]=[H|T].
2  H = 1
3  T = []
```

- head is a single term, but the tail is a list

```prolog
1 | ?- [1, 2]=[H|T].
2  H = 1
3  T = [2]
```
tail is a list of lists

```prolog
| ?- [1, [2, 3]] = [H | T].
  H = 1
  T = [[2, 3]]
```

head is a list (1)

```prolog
| ?- [[1, 2], 3] = [H | T].
  H = [1, 2]
  T = [3]
```

head is a list (2)

```prolog
| ?- [[1, 2, 3], 4] = [[H1 | T1] | T2].
  H1 = 1
  T1 = [2, 3]
  T2 = [4]
```
• head has a specific structure

```
| ?- [1,2,3,4]=[Ha,Hb|T].
Ha = 1
Hb = 2
T = [3,4]
```

In Prolog lists may contain mixed elements of different types:

```
| ?- L = [[X,a],bolek,[],['Celina',[]]],
   L = [H|T].
H = [X,a]
L = [[X,a],bolek,[],['Celina',[]]]
T = [bolek,[],['Celina',[]]]
```
To summarize:

```
| ?- L = [A,B|C], L = [ala, bolek, [1,2,3]].

A = ala
B = bolek
C = [[1,2,3]]
L = [ala, bolek, [1,2,3]]
```

- L is a list with 3 elements: (1) a constant ala, (2) a constant bolek, (3) a list [1,2,3]
- the head of L is ala
- the tail C is a list containing the single element [1,2,3]
Basic operations on the list. In order to use lists, we need to have means at least to create, read, extended and delete elements from the list. What we know so far:

- **creation**: start from an empty list and add elements with the dot predicate → (b, .(a, [])) = [b, a]
- **creation**: declare the whole list → L = [b, a]
- **read**: element by element by dividing the list on its head (single element to read) and the tail, which can be treated in the same way
- **extend**: add an element on the beginning using the dot predicate
- **delete**: we don’t know yet
Problem: write a predicate that will scan a given list looking for a given term, i.e., it will check if the term is on the list.
The main idea is the following:

1. we know how to take the first element of the list (its head)
2. compare the head with the given term; if fail – see next step, if success – finish
3. discard the head, creating a new (shorter) list
4. repeat from the beginning

```
list_element(X,[X|T]).
list_element(X,[H|T]) :- list_element(X,T).
```

This code declares two dummy variables (T in first line, H in the second line). Prolog will issue a warning about singleton variables.
Solution:

```prolog
1  list_element(X,[X|_]).
2  list_element(X,[_|T]) :- list_element(X,T).
```

- underscore represents “something which is not important at the moment”
- this predicate uses recurrence (kind of loop but faster and less obvious)
- this predicate is defined by itself
- notice the order of conditions (will it work if you interchange line 1 and 2?)
- notice the condition terminating the recurrence (line 1) – this rule is always true
list_element(X, [X | _]).
list_element(X, [_|T]) :- list_element(X, T).

The definition can be summarized by two principles:

- the element $X$ is at the beginning of the list – then it is a member of the list
- the element $X$ is not at the beginning of the list – then remove the first element and repeat everything for what has remained

Notice:

- the order of the rules is crucial
- the $X$ from the first line has nothing to do with the $X$ from the second line (definitions are enclosed between the start and the finishing dot)
- the rule on the R-hand side of line 2 calls the definition of list_element, which starts again at line 1
SUMMARY

• prolog programs may run without the need of asking questions
• prolog programs may be interactive
• lists in prolog: dynamical length, multi-typed
• a non-empty list has always **always**
  ▶ a head (**single element**)  
  ▶ a tail (**list**)  
• an empty list has no head nor tail
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append/3</code></td>
<td>Concatenates</td>
</tr>
<tr>
<td><code>member/2</code></td>
<td>Seeks for element</td>
</tr>
<tr>
<td><code>reverse/2</code></td>
<td>Reverses</td>
</tr>
<tr>
<td><code>delete/3</code></td>
<td>Removes all occurrences</td>
</tr>
<tr>
<td><code>select/3</code></td>
<td>Removes first occurrence</td>
</tr>
<tr>
<td><code>subtract/3</code></td>
<td>Subtracts two lists</td>
</tr>
<tr>
<td><code>permutation/2</code></td>
<td>New list from the same elements</td>
</tr>
<tr>
<td><code>sublist/2</code></td>
<td>Seeks for sublist</td>
</tr>
<tr>
<td><code>last/2</code></td>
<td>Last element</td>
</tr>
<tr>
<td><code>length/2</code></td>
<td>Number of elements</td>
</tr>
<tr>
<td><code>nth/3</code></td>
<td>N-th element</td>
</tr>
<tr>
<td><code>min_list/2</code></td>
<td>Smallest of the elements</td>
</tr>
<tr>
<td><code>max_list/2</code></td>
<td>Largest of the elements</td>
</tr>
<tr>
<td><code>sum_list/2</code></td>
<td>Sum of all elements</td>
</tr>
<tr>
<td><code>sort/2</code>, <code>msort/2</code>, <code>keysort/2</code></td>
<td>Sorted lists</td>
</tr>
</tbody>
</table>
append/3

append(L1,L2,M) creates a new list M=L1+L2

?- L1=[a,b,c], L2=[d,e,f], append(L1,L2,M).

L1 = [a,b,c]
L2 = [d,e,f]
M = [a,b,c,d,e,f]

yes

- concatenation done in order L1 L2
- append(L1,L2,M) is (in general) not the same as append(L2,L1,M)
append may be used to extend the list on its beginning or its end

```
| ?- L=[1,2,3], append([0],L,M).
L = [1,2,3]
M = [0,1,2,3]
yes
```

```
| ?- L=[1,2,3], append(L,[4],M).
L = [1,2,3]
M = [1,2,3,4]
(2 ms) yes
```

The first case can be equivalently written as:

```
| ?- L=[1,2,3], M=[0|L].
L = [1,2,3]
M = [0,1,2,3]
yes
```

The second case has no equivalent in the known constructions.
Examples:

(1) Empty list is neutral for concatenation:

```
| ?- append([],L1,M).
M = L1
yes
```

(2) Check if the lists sum up to a known result:

```
| ?- append([a,b],[c,d],[a,b,c,d]).
yes
```

```
| ?- append([a,b],[c,d],[a,b,c]).
no
```

(3) Find the missing part:

```
| ?- append([a,b,c],X,[a,b,c,d,e,f,g]).
X = [d,e,f,g]
yes
```
(4) When is concatenation symmetric?

- there is infinitely many solutions
- either one or both \( L_1, L_2 \) are empty
- or \( L_1 = L_2 \)
(5) Find all decompositions of a given list:

```prolog
?- append(L1, L2, [a, b, c, d]).
L1 = []
L2 = [a, b, c, d] ? a

L1 = [a]
L2 = [b, c, d]

L1 = [a, b]
L2 = [c, d]

L1 = [a, b, c]
L2 = [d]

L1 = [a, b, c, d]
L2 = []
yes
```
(6) List as a storage place for user input:

```prolog
run(L) :- read(X), append(L,[X],M),
          write(M), nl, run(M).
|
?- run([]).
ala.
[ala]
ma.
[ala,ma]
kota.
[ala,ma,kota]
a.
[ala,ma,kota,a]
kot.
[ala,ma,kota,a,kot]
ma.
[ala,ma,kota,a,kot,ma]
wszy.
[ala,ma,kota,a,kot,ma,wszy]
(...)
```
**member/2**

`member(T,L)` checks, if term `T` is on the list `L`

```
|   | ?- member(a,[a,b,c]).
|---|---------------------
| 1 | yes                 
```

- term `T` can be anything that can be included in the list
- some caution required if `T` is a variable (see below)
(1) Some basic examples (sublists and such):

1 | ?- member([a],[a,b,c]).
   no

1 | ?- member([a],[[a],b,c]).
   true  ? ;
   no

1 | ?- member(a,[[a],b,c]).
   no

1 | ?- member(a,[]).
   no

1 | ?- member([],[],a,b,c]).
   no
(2) Some basic examples (variables here and there):

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>?- member(X,[a, X, b]).</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X = a ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>true</code> ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X = b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>?- member(X,[Y,a,X,b]).</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Y = X ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X = a ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>true</code> ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>X = b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>?- member(X,[a,_,b]).</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X = a ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>true</code> ? ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X = b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(3) Find elements of the list (list scan):

```
| ?- L=[a,b,c,d], member(X,L).
L = [a,b,c,d]
X = a ? a
L = [a,b,c,d]
X = b ?
L = [a,b,c,d]
X = c ?
L = [a,b,c,d]
X = d
```

(4) Discriminating between two lists:

```
| ?- (L=[a,b,c,d]; L=[1,2,3]), member(2,L).
L = [1,2,3] ? ;
no
```
(5) List scan in practice. Assume you have to perform certain operation on many input data. Let it by “add 10” applied to three numbers. You can do it in one go:

```
| ?- member(X,[2,3,4]), Y is X+10.
X = 2  
Y = 12  ? ;  
X = 3  
Y = 13  ? ;  
X = 4  
Y = 14  
yes
```
**Problem:** our own implementation of member/2.

**Strategy:** define predicate `naliscie(T,L)` according to the following scheme:

- check if T is the first element of L
- if not, discard the first element and repeat

**Solution 1**

1. `naliscie(_,[]) :- fail.`
2. `naliscie(T,L) :- L=[G|O], (T=G ; naliscie(T,O)).`

**Solution 2** (notice the order of clauses!)

1. `naliscie(X,[X|_]).`
2. `naliscie(X,[_|Y]) :- naliscie(X,Y).`

**Question:** Why in solution 2 we don’t have to include explicitly the case of an empty list?
reverse/2

reverse(LM,ML) is true if ML is the reversed list LM

1 | ?- reverse([a,b,c],[c,b,a]).  
yes

(1) Finding the reversed list:

1 | ?- reverse([a,b,c,d],X).  
X = [d,c,b,a]  
yes

1 | ?- reverse(X,[a,b,c,d]).  
X = [d,c,b,a]  
yes
(2) User input revisited:

```prolog
run(L) :- read(X), append([X], L, M),
    reverse(M, N),
    write(N), nl, run(M).

| ?- run([]).
kobyla.
[kobyla]
ma.
[kobyla,ma]
maly.
[kobyla,ma,maly]
bok.
[kobyla,ma,maly,bok]
(...)
```
(3) Some other examples:

```
| ?- L=[kobyla,ma,maly,bok], reverse(L,M).
L = [kobyla,ma,maly,bok]
M = [bok,maly,ma,kobyla]
yes
```

```
| ?- L=[k,o,b,y,l,a,m,a,m,a,l,y,b,o,k], reverse(L,M).
L = [k,o,b,y,l,a,m,a,m,a,l,y,b,o,k]
M = [k,o,b,y,l,a,m,a,m,a,l,y,b,o,k]
yes
```
(4) Find all patterns for “palindromes”:

```prolog
|  ?- reverse(L,L).
|  L = [] ? ;
|  L = [_] ? ;
|  L = [A,A] ? ;
|  L = [A,_,A] ? ;
|  L = [A,B,B,A] ? ;
|  L = [A,B,_,B,A] ? ;
(....)
```
SUMMARY

- lists are extremely universal
- append/3
- member/2
- reverse/2
delete/3

delete(L,T,M) removes all occurrences of T from L, creating M

- L the initial list (usually input)
- T term to be removed from L (usually input)
- M resultant list (usually output)

Basic usage:

```
1  |  ?- L=[a,b,a,c,b,a,d], delete(L,a,M).
2  L = [a,b,a,c,b,a,d]
3  M = [b,c,b,d]
4  yes
```
**A fancy example**: remove certain characters from a “dirty” list. Here, remove all numbers from the input, leaving letters untouched.

```prolog
1 clean(L,[]) :- write(L).
2 clean(L,M) :- M=[X|_], delete(L,X,K),
3                     delete(M,X,N), clean(K,N).

| ?- L=[a,1,l,a,4,5,7,m,a,0,k,6,o,4,t,a],
  M=[0,1,2,3,4,5,6,7,8,9], clean(L,M).

[a,l,a,m,a,k,o,t,a]
L = [a,1,l,a,4,5,7,m,a,0,k,6,o,4,t,a]
M = [0,1,2,3,4,5,6,7,8,9] ? ;
no
```

**Notice**: lines 10, 11 – the code has some issue with termination.
The correct version (compare line 1)

```prolog
clean(L,[]) :- write(L), !.
clean(L,M) :- M=[X|_], delete(L,X,K),
            delete(M,X,N), clean(K,N).
```

| ?- L=[a,1,l,a,4,5,7,m,a,0,k,6,o,4,t,a],
   M=[0,1,2,3,4,5,6,7,8,9], clean(L,M). |

[a,l,a,m,a,k,o,t,a]
L = [a,1,l,a,4,5,7,m,a,0,k,6,o,4,t,a]
M = [0,1,2,3,4,5,6,7,8,9] ?
yes

The !/0 predicate is always true and allows to control the backtracking process.
The operator `!` represents the **cut** or **break-point** during recurrence. Definition of positive integers may use recurrence:

```prolog
lcalk(0).
lcalk(X) :- lcalk(Y), X is Y+1.

| ?- lcalk(X).
X = 0 ? ;
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
X = 4 ? ;
Prolog interruption (h for help) ? a
execution aborted
```

Starting from 0 the following integer can be generated by applying +1 to the previous one.
The operator \(!\) stops the recurrence:

\begin{verbatim}
lcalk(0).
lcalk(X) :- lcalk(Y), X \textbf{is} Y+1, !.
| ?- lcalk(X).
X = 0 ? ;
X = 1
\end{verbatim}

(1 ms) yes

It indicates that during backtracking prolog will not alter the previously found correct solution. Technically speaking, it will not go beyond the \(!\) mark during the second and subsequent scans of the predicate. The cut operator is often used together with the \texttt{fail} predicate to terminate the sequence.
**Side remark:** notice the order of predicates in the definition

```
lcalk(0).
lcalk(X) :- lcalk(Y), X is Y+1.
```

The following is logically equivalent, but incorrect for our purpose:

```
lcalk(0).
lcalk(X) :- X is Y+1, lcalk(Y).

?- lcalk(X).
X = 0 ? ;
uncaught exception:
    error(instantiation_error, (is)/2)
```

Do you know why?
select/3

select(T,L,M) creates from list L a new list M removing one occurrence of the term T

Basic usage:

```prolog
?- L=[a,b,a,c,b,a,d], select(a,L,M).

L = [a,b,a,c,b,a,d]
M = [b,a,c,b,a,d] ;

L = [a,b,a,c,b,a,d]
M = [a,b,c,b,a,d] ;

L = [a,b,a,c,b,a,d]
M = [a,b,a,c,b,d] ;

no
```
**subtract/3** (new in gprolog 1.4.4)

`subtract(L, M, N)` removes from `L` all occurrences of members of `M`, creating a new list `N`.

**Basic usage:**

```
1 | ?- subtract([1,2,3,a,b,c,4,5,6],[a,b,c],X).
   X = [1,2,3,4,5,6]
   yes

1 | ?- subtract([a,b,c],[b,z,a,d],X).
   X = [c]
   yes
```
Examples:

```prolog
| ?- subtract([1,2,3,4,5],X,M).
M = []
X = [1,2,3,4,5|_]
yes
```

- only one solution proposed
- the solution is “maximal”, i.e., contains the whole list \([1,2,3,4,5]\)
  plus something more, which is not important
- other solutions exist, like \(X=[a]\), \(M=[1,2,3,4,5]\) etc. but not taken into account!
Built-in predicates for list manipulation

?- subtract(X,[1,2,3,4,5],M).
M = []
X = []  ;

M = []
X = [1]  ;

M = []
X = [1,1]  ;

M = []
X = [1,1,1]  ;
Prolog interruption (h for help) ? a
execution aborted

- not useful, simplest solutions
- infinite series
To sum up, `subtract`

- is a new predicate
- doesn’t seem to “work properly” (my personal feeling!), *ie.*, it is not as “stable” as other predicates against different input parameters.
## Quick comparison:

<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>DESCRIPTION</th>
<th>ARG.1</th>
<th>ARG.2</th>
<th>ARG.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete(L,T,M)</td>
<td>removes all occurrences of term T in list L creating list M</td>
<td>list</td>
<td>term</td>
<td>list</td>
</tr>
<tr>
<td>select(T,L,M)</td>
<td>removes one occurrence of term T in list L creating list M</td>
<td>term</td>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>subtract(L,T,M)</td>
<td>removes all occurrences of members of list T in list L creating list M</td>
<td>list</td>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

- notice the order of arguments!
1 | ?- L=[a,b,a,c,d], T=a, delete(L,T,M).
   
   L = [a,b,a,c,d]
   M = [b,c,d]
   T = a
   yes

1 | ?- L=[a,b,a,c,d], T=a, select(T,L,M).
   
   L = [a,b,a,c,d]
   M = [b,a,c,d]
   T = a ? ;

   L = [a,b,a,c,d]
   M = [a,b,c,d]
   T = a ? ;
   no

1 | ?- L=[a,b,a,c,d], T=[a,c], subtract(L,T,M).
   
   L = [a,b,a,c,d]
   M = [b,d]
   T = [a,c]
   yes
permutation/2

permutation(L,M) changes the order of elements of L, producing M

Basic usage:

```prolog
| ?- permutation ([a,b,c],X).
X = [a,b,c]  \ ?  a
X = [a,c,b]
X = [b,a,c]
X = [b,c,a]
X = [c,a,b]
X = [c,b,a]
no
```
Be careful – it doesn’t work the other way round:

```
| ?- permutation(X,[1,2,3]).
X = [1,2,3] ? ;
Fatal Error: global stack overflow
```

Example:

```
| ?- permutation([1,1,1],X).
X = [1,1,1] ? a
X = [1,1,1]
X = [1,1,1]
X = [1,1,1]
X = [1,1,1]
X = [1,1,1]
X = [1,1,1]
no
```
**sublist/2**

`sublist(L,M)` checks if members of `L` can be found in `M` in the same order of appearance as in `L`.

**Basic usage:**

1. `?- sublist([a,b,c],[1,2,3,a,b,c,4,5,6]).`
   - true
   - no

2. `?- sublist([a,b,c],[1,2,a,6,b,4,c]).`
   - true
   - no

- the list `L` is not a sublist of `M` (strictly speaking)
- the elements of `L` may disjointly appear in `M` provided that the order is correct.
Example 1:

```
| ?- sublist(X,[a,b,c]).
X = [a,b,c] ? a
X = [b,c]
X = [c]
X = []
X = [b]
X = [a,c]
X = [a]
X = [a,b]
(1 ms) no
```
Example 2:

<table>
<thead>
<tr>
<th></th>
<th>?- sublist([a,b,c],X).</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X = [a,b,c] ? ;</td>
</tr>
<tr>
<td>2</td>
<td>X = [_ ,a,b,c] ? ;</td>
</tr>
<tr>
<td>3</td>
<td>X = [_ ,_ ,a,b,c] ? ;</td>
</tr>
<tr>
<td>4</td>
<td>X = [_ ,_ ,_ ,a,b,c] ? ;</td>
</tr>
<tr>
<td>5</td>
<td>X = [_ ,_ ,_ ,_ ,a,b,c] ?</td>
</tr>
<tr>
<td>6</td>
<td>Prolog interruption (h for help) ? a</td>
</tr>
<tr>
<td>7</td>
<td>execution aborted</td>
</tr>
</tbody>
</table>
**last/2**

last(L, X) checks if X is the last element of L

Basic usage:

```
|  ?- L=[1,2,3,4], last(L,X).
  L = [1,2,3,4]
  X = 4
  yes
```
length/2

length(L,X) checks if X is the number of elements of L

Basic usage:

1 | ?- length([1,2,3,4],X).
  X = 4
  yes

2 | ?- length([a,[a,b,c]],X).
  X = 2
  yes

- an empty list has length zero
Example 1:

<table>
<thead>
<tr>
<th>?- length([a,X],Y).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = 2</td>
</tr>
<tr>
<td>yes</td>
</tr>
</tbody>
</table>

- X is a single object (also possibly a list)
- as a single object, it will change the length of the list as +1
- therefore X is not instantiated during the resolution of this call (it is simply not important for the solution, what is the nature of X)
Example 2.1:

```
| ?- length([a|X],Y).
X = []
Y = 1 ? ;

X = [_]
Y = 2 ? ;

X = [_,_]
Y = 3 ?
Prolog interruption (h for help) ? a
execution aborted
```

- The `|` operator has a list on its right hand side (the tail is always a list).
- The construction `[a|X]` is a list of total length `1 + (length of X)`.
### Example 2.2:

<table>
<thead>
<tr>
<th>?- length((a,X),Y).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = []</td>
</tr>
<tr>
<td>Y = 1 ? ;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>X = [_]</td>
</tr>
<tr>
<td>Y = 2 ? ;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>X = [<em>,</em>]</td>
</tr>
<tr>
<td>Y = 3 ?</td>
</tr>
<tr>
<td>Prolog interruption (h for help) ? a</td>
</tr>
<tr>
<td>execution aborted</td>
</tr>
</tbody>
</table>

- the . operator appends a term to a list
- the construction (a,X) is a list of total length $1 + (\text{length of } X)$
nth/3

nth(N,L,X) checks if term X can be unified with the N-th element of list L.

Basic usage:

```
| ?- nth(3,[a,b,c,d],X).
  X = c
  yes
```

- the elements are counted from 1 (not from zero, as in some languages); this is consistent with the fact that an empty list has length zero
SUMMARY

- shrinking lists:
  - delete/3
  - select/3
  - subtract/3 (new in prolog 1.4.4)

- changing the order of elements: permutation/2

- checking the content of the list: sublist/2

- inspecting elements:
  - last/2
  - nth/3

- counting elements: length/2

- the cut predicate !/0
Prolog
Built-in predicates for list manipulation

Marek Góźdź (2018/2019)
**min – max – sum**

min_list(L,X) checks if X is the smallest of elements of L (numerical)

max_list(L,X) checks if X is the largest of elements of L (numerical)

sum_list(L,X) checks if X is the sum of all elements of L (numerical)

```
| ?- min_list([1,2,3,4],X).
X = 1
yes

| ?- max_list([1,2,3,4],X).
X = 4
yes

| ?- sum_list([1,2,3,4],X).
X = 10
yes
```
sort/2

sort(L1,L2) unifies L2 with L1 after sorting

Sorting order:

- variables *in the order of appearance*
- some special characters
- numbers
- other special characters (see ASCII table)
- strings in strong parentheses ’TEXT’
- constants
- strings in weak parentheses "TEXT"
- repeating elements are merged
Examples:
Sorting texts I

1 | ?- sort (["Lech","Czech","Rus","Anka"], L).
2 L = [[65,110,107,97],[67,122,101,99,104],
3 [76,101,99,104],[82,117,115]]

Sorting texts II

1 | ?- sort ([’Lech’,’Czech’,’Rus’,’Anka’], L).
2 L = [’Anka’,’Czech’,’Lech’,’Rus’]

Variables are not sorted

1 | ?- sort ([Lech,Czech,Rus,Anka], L).
2 L = [Lech,Czech,Rus,Anka]

Merging duplicates

1 | ?- sort ([Lech,Czech,Anka,Rus,Anka], L).
2 L = [Lech,Czech,Anka,Rus]
Prolog

Built-in predicates for list manipulation

Sorting mixed entries I

\[
\begin{align*}
| \text{?- sort([1,2,4,3,a,Z,z,A],X).} & \text{X = [Z,A,1,2,3,4,a,z]} \\
\end{align*}
\]

Sorting mixed entries II

\[
\begin{align*}
| \text{?- sort([AA,Z,a,b,Y,A,B,1,2,_,’X’,c,’Z’],X).} & \text{X = [AA,Z,Y,A,B,_,1,2,’X’,’Z’,a,b,c]} \\
\end{align*}
\]

First argument cannot be uninstantiated

\[
\begin{align*}
| \text{?- sort(L,[1,2,3]).} & \text{uncaught exception:} \\
& \text{error(instantiation_error,sort/2)} \\
\end{align*}
\]
Example: How to use lists to solve simple school problems?

Five friends took part in a run. Willy did not win. Gregory was 3rd, after Daniel. Daniel was not 2nd. Andrew did not win, nor was he the last. Bill finished right after Willy.

General remarks:

- we need variables, which will represent the runners
- at the end we want to assign numbers form 1 to 5 to them, to indicate the order of completing the race
- the same place cannot be assigned to two different runners
- some conditions from the puzzle must be fulfilled
Strategy:

- introduce variables: Will, Gregory, Daniel, Andrew, Bill
- introduce a list $L$ which holds the numbers from 1 to 5
- write conditions from the text of the exercise
- remove the numbers represented by Will, Gregory, ... form the list $L$ one by one
- the last point assures that no number will be assigned to any variable twice*)

*) Different approach: permute the list $[1, 2, 3, 4, 5]$ until all conditions are met.
Five friends took part in a run. Willy did not win. Gregory was 3rd, after Daniel. Daniel was not 2nd. Andrew did not win, nor was he the last. Bill finished right after Willy.

Solution:

```prolog
place(W,G,D,A,B) :- L0=[1,2,3,4,5], G=3, 
    select(W,L0,L1), W\=1, 
    select(D,L1,L2), D\=2, 
    select(A,L2,L3), A\=1, A\=5, 
    select(B,L3,[G]), B is W+1, 
    G>D. 

| ?- place(Willy,Gregory,Daniel,Andrew,Bill). 
Andrew = 2 
Bill = 5 
Daniel = 1 
Gregory = 3 
Willy = 4 ? 
yes
```
Other example:

Four cars: Ford, Toyota, Mazda, and Honda are parked next to each other, from left to right. All of them have different colors:

- one of the cars is red
- the car to the right of Ford is blue
- Toyota is second
- Mazda is white
- Honda is not fourth and is not silver

What is the order of cars and what are their colors?
Solution 1:

```prolog
cars(F,T,M,H) :-
    C1=[red,blue,white,silver], N1=[1,2,3,4],
    F=[Fc,Fn], T=[Tc,Tn], M=[Mc,Mn], H=[Hc,Hn],
    member(Fc,C1), Fc \= blue , select(Fc,C1,C2),
    member(Tc,C2),
    member(Mc,C3), Mc=white , select(Mc,C3,C4),
    member(Hc,C4), Hc \= silver ,
    member(Fn,N1), Fn \= 4 , select(Fn,N1,N2),
    member(Tn,N2), Tn=2 , select(Tn,N2,N3),
    member(Mn,N3),
    member(Hn,N4), Hn<4 ,
    X is Fn+1,
    ( T=[blue,X] ; M=[blue,X] ; H=[blue,X] ).
```

Marek Góźdź (2018/2019)
PROLOG
19 listopada 2018
138 / 174
This solution:

- uses two lists (colors and numbers)
- each car is represented as a list (color and number)
- combination of `select/3` and `member/2` plus many intermediate lists assures, that no repetition of entries show up in the solution
- has a lot of unnecessary conditions (ex., from $X=F_{n+1}$ follows that $F_n \neq 4$ and $F_c \neq \text{blue}$)

```prolog
| ?- cars(Ford, Toyota, Mazda, Honda).
Ford = [silver,1]
Honda = [red,3]
Mazda = [white,4]
Toyota = [blue,2] ? ;
no
```

**Homework:** clean and shorten the code.
Solution 2: a different approach

cars2(X,Y) :-
  C=[red, blue, white, silver], permutation(C,X),
  N=[ford, toyota, mazda, honda], permutation(N,Y),
  nth(Rn,X,red), nth(Bn,X,blue),
  nth(Wn,X,white), nth(Sn,X,silver),
  nth(Fn,Y,ford), nth(Tn,Y,toyota),
  nth(Mn,Y,mazda), nth(Hn,Y,honda),
  Bn is Fn+1, Mn=Wn, Hn\=Sn, Tn=2, Hn<4.
Here:

- two output lists store cars \((X)\) and colors \((Y)\)
- the position of cars is represented by their position on the list
- \texttt{permutation/2} assures that no repetitions occur
- all conditions from the exercise are written entirely in the language of positions \(X_n\) (line 8.)

```
| ?- cars2(X,Y).
X = [silver,blue,red,white]
Y = [ford,toyota,honda,mazda] ? ;
(2 ms) no
```

**Remark:** this solution is rather slow (because of \texttt{permutation}), hence the 2 ms time required to check all the remaining possibilities. It may not work efficiently for a greater number of cars.

**Homework:** check if the remark above is true by increasing the length of the lists.

**Homework:** can you propose a third solution to this exercise?
Example: Magic squares.
In a magic square of size $n \times n$
- entries are numbers from 1 to $n^2$
- each entry is different
- sum of elements in rows, columns and both diagonals is constant

A popular 3x3 square (sum 15)

\[
\begin{array}{ccc}
6 & 1 & 8 \\
7 & 5 & 3 \\
2 & 9 & 4 \\
\end{array}
\]

A well-known 4x4 square (sum 34)

\[
\begin{array}{cccc}
16 & 3 & 2 & 13 \\
5 & 10 & 11 & 8 \\
9 & 6 & 7 & 12 \\
4 & 15 & 14 & 1 \\
\end{array}
\]
**Side remark:** some of the magic squares have interesting properties.

A. Dürer’s 4x4 square from *Melencolia I* (1514)

The sum is 34 in all rows, columns, both diagonals, all five green subsquares, and when adding together the corners.
Code to generate a 3x3 square:

```
kwadrat3x3(S,X) :-
    S=[E11,E12,E13,E21,E22,E23,E31,E32,E33],
    L=[1,2,3,4,5,6,7,8,9], permutation(S,L),
    sum_list([E11,E12,E13],X),
    sum_list([E21,E22,E23],X),
    sum_list([E31,E32,E33],X),
    sum_list([E11,E21,E31],X),
    sum_list([E12,E22,E32],X),
    sum_list([E13,E23,E33],X),
    sum_list([E11,E22,E33],X),
    sum_list([E13,E22,E31],X).
```
| ?- kwadrat3x3(X,S).

S = 15
X = [6,1,8,7,5,3,2,9,4] ? ;
S = 15
X = [8,1,6,3,5,7,4,9,2] ? ;
S = 15
X = [6,7,2,1,5,9,8,3,4] ? ;
S = 15
X = [8,3,4,1,5,9,6,7,2] ? ;
S = 15
X = [2,7,6,9,5,1,4,3,8] ? ;
S = 15
X = [4,3,8,9,5,1,2,7,6] ? ;
S = 15
X = [2,9,4,7,5,3,6,1,8] ? ;
S = 15
X = [4,9,2,3,5,7,8,1,6] ? ;
(371 ms) no
The obtained solution clearly shows that

- there is only one magic square 3x3
- all propositions are merely modifications of the square

\[
\begin{array}{ccc}
6 & 1 & 8 \\
7 & 5 & 3 \\
2 & 9 & 4 \\
\end{array}
\]

including mirror inversion, transposition etc.

- the sum is always 15
One may extend the previous code to the 4x4 version:

```prolog
kwadrat4x4(S,X) :-
    S=[E11,E12,E13,E14,E21,E22,E23,E24,
       E31,E32,E33,E34,E41,E42,E43,E44],
    L=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],
    permutation(S,L),
    sum_list([E11,E12,E13,E14],X),
    sum_list([E21,E22,E23,E24],X),
    sum_list([E31,E32,E33,E34],X),
    sum_list([E41,E42,E43,E44],X),
    sum_list([E11,E21,E31,E41],X),
    sum_list([E12,E22,E32,E42],X),
    sum_list([E13,E23,E33,E43],X),
    sum_list([E14,E24,E34,E44],X),
    sum_list([E11,E22,E33,E44],X),
    sum_list([E14,E23,E32,E41],X).
```
- the 4x4 code is extremely ineffective
- after over an hour it didn’t find any result
- the transition from 3x3 to 4x4 is in fact a transition from 9 fields to 16 fields (almost twice as many!)
- the number of permutations of the list \([1,2,\ldots,9]\) is \(9! = 362880\)
- the number of permutations of the list \([1,2,\ldots,16]\) is \(16! \approx 2.1 \times 10^{13}\)
- prolog can do the job effectively even for 9x9 squares (see: sudoku) with the help of a special subsystem: Finite Domain Solver
Example: Towers of Hanoi

| | |
|XXX|
|XXXXX|
|XXXXXXX|
|XXXXXXXXX|

Rules of the game.
Move the stack to another rod, obeying:

- only one disk can be moved at a time
- each move consists of taking the upper disk from one of the stacks and placing it on top of another stack
- no disk may be placed on top of a smaller disk

The minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where $n$ is the number of disks.
Strategy: [See, e.g., Clocksin, Mellish, Sect. 7.4]

- the 3 rods are: source, destination, and one temporary. The solution scheme is recursive. For \( n \) disks:
  1. move \( n - 1 \) disks from source to temporary
  2. move the \( n \)-th (largest) disk from source to destination
  3. move \( n - 1 \) disks from temporary to destination

- to complete (1) you must move \( n - 2 \) disks from source to (another) temporary and so on, hence recurrence is needed

- as input to the program, the number \( n \) must be given
The code:

```prolog
hanoi(N) :- move(N,source,destination,temp).

move(0,_,_,_,_):- !.
move(N,A,B,C):- M is N-1,
             move(M,A,C,B),
             info(A,B),
             move(M,C,B,A).

info(X,Y):= write([X,-->,Y]), nl.
```

Here: left rod is “source”, center is “destination”, right is “temporary”

Notice:

- no algorithm to solve the puzzle is given
- we describe the basic possible operation (move) and the (trivial) starting point with zero disks on the rod
- prolog does the rest
For 3 disks one needs $2^3 - 1 = 7$ moves, for 4 disks the solution requires $2^4 - 1 = 15$ moves.
SUMMARY

- No summary this time.
- This was the last regular lecture.
- A comprehensive summary will be given during our last meeting.
- Q & A session will follow, so prepare your questions!
A comprehensive summary

Preliminaries, syntax, and others:

- declarative paradigm of programming
- language of logic (, ; :-)
- don’t forget about the full-stop at the end of the sentence
- constant (a, ala, 2)
- variable (A, Ala, Dwa) can be assigned or unassigned
- predicates (built-in or user-defined)
- predicate can be a fact, relation or a rule

1. piekna(ala).        %  fact
2. piekna(ala,usmiechnieta).     %  relation
3. piekna(Ona) :- usmiechnieta(Ona).       %  rule
Program in prolog:
   - set of facts (knowledge base / database)
     - fact has the form: predicate(argument).
     - the argument must be a constant
     - construction: predicate(X). is wrong!
   - my own definition(s)
   - often the main program is a single predicate, which contains subsequent calls of other predicates

How to run a program in the GNU prolog console:
   - gprolog → consult(user)
   - gprolog → consult('program-name.pro')
   - gplc program-name.pro → program-name

Running a program in prolog means
   - either asking questions,
   - or running the main predicate (with or without input data).
Terms (see Sect. 6.2 in the Manual):

- simple or compound
- type: **number** (including: **integer**, **float**)
- type: **atom** (including **constant**, **character**, **string**)
- type: **atomic** (including: **atom**, **number**)

Predicates which check term types:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var(T)</td>
<td>true, if T a variable</td>
</tr>
<tr>
<td>nonvar(T)</td>
<td>true, if T not variable</td>
</tr>
<tr>
<td>atom(T)</td>
<td>true, if T an atom</td>
</tr>
<tr>
<td>atomic(T)</td>
<td>true, if T an atom or number</td>
</tr>
<tr>
<td>number(T)</td>
<td>true, if T a number</td>
</tr>
<tr>
<td>compound(T)</td>
<td>true, if T not simple</td>
</tr>
<tr>
<td>list(T)</td>
<td>true, if T a list</td>
</tr>
</tbody>
</table>
Constants:

- label starts with small letter
- unique
- different labels label different constants
- constants having the same value are still different
- unification of different objects is never possible

```
1 | ?- a=2.  
   no
```

Here, "2" and "a" are two different constants!
Variables:

- label starts with **capital letter**
- no type declaration
- weak type control
- unique, but can be unified
- different labels label different variables
- **assigned** or **unassigned**
- once assigned a value, remains assigned
A recap on **unification**

- two terms can be unified if they are the same
  
  ```prolog
  1  A=A.  /* yes */
  2  a=a.  /* yes */
  3  a=b.  /* no */
  ```

- unassigned variable (A) will be unified with another term by setting its type and value to match that term
  
  ```prolog
  1  A=B.  /* yes, A unassigned, B assigned */
  2  A=b.  /* yes */
  ```

- unification of two unassigned variables is always true
  
  ```prolog
  1  A=B.  /* yes; A, B unassigned */
  ```

Notice:

- unification is not an assignment but...
- ... it may result in assignment of variables as a “side-effect”
- its result is true or false
Unification may be treated as a special case of comparison (ordering)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>true, if unification possible</td>
</tr>
<tr>
<td>=</td>
<td>true, if unification not possible</td>
</tr>
<tr>
<td>==</td>
<td>true, if comparison positive</td>
</tr>
<tr>
<td>==</td>
<td>true, if comparison negative</td>
</tr>
<tr>
<td>@&lt;</td>
<td>true, if term smaller</td>
</tr>
<tr>
<td>@=&lt;</td>
<td>true, if term smaller equal</td>
</tr>
<tr>
<td>@&gt;</td>
<td>true, if term greater</td>
</tr>
<tr>
<td>@&gt;=</td>
<td>true, if term greater equal</td>
</tr>
</tbody>
</table>

Order of terms:
- variables (oldest first), FD variables (oldest first)
- floating point numbers (in numerical order)
- integers (in numerical order)
- atoms (in alphabetical – character code – order)
- compound terms (ordered first by arity, then by the name of the principal functor and by the arguments in left-to-right order)
Arithmetics

- `is` is the operator for substituting the result of an arithmetic operation for a variable or constant
- Lots of operations and functions are predefined

```prolog
% A pseudo-random number generator <0,1>
| ?- random(X).
X = 0.8401877167634666

| ?- random(X).
X = 0.39438292663544416

| ?- random(X).
X = 0.78309922339394689
```
1 | ?- X=2, Y is X+3.
   X = 2
   Y = 5
   yes

1 | ?- random(X), Y is X+3.
   X = 0.91164735751226544
   Y = 3.91164735751226544
   yes

1 | ?- L=[1,2,3,4], L=[A,B,X|_], Y is X+10.
   A = 1
   B = 2
   L = [1,2,3,4]
   X = 3
   Y = 13
Unification, comparison, equals to...

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>unify L- and R-hand side</td>
</tr>
<tr>
<td>==</td>
<td>compare L- and R-hand side</td>
</tr>
<tr>
<td>is</td>
<td>unify the expression on the L-hand side with the value of the evaluated arithmetical expression on the R-hand side</td>
</tr>
</tbody>
</table>
Lists

- empty list [] has length zero
- non-empty list can be always decomposed into head and tail [H|T]
- tail is always a list

```
% check if term is a list
is_a_list([]).
is_a_list([H|T]) :- is_a_list(T).
```

- elements of mixed types
- this is the main prolog data structure, so lists are widely used to almost everything
  - represent the order of runners
  - represent row/column in magic square, sudoku, tic-tac-toe
  - ...
Some most useful pre-defined predicates describing lists manipulations

- `append(A,B,AplusB).`
- `delete(L,T,M).`
- `member(T,L).`
- `permutation(L1,L2).`
- `length(L).`
- `last(L,T).`
- `sum_list(L,T).`
Trees

- trees can be described by using previous-next relations (like parent-child)
- these relations have two or more arguments

**Example 1.** Tree with all branches disjoint.

```
    root
   /    \
  left  right
 /    /   \   
llefl   rleaf  lright  rright
```

- `next(root,left).`
- `next(root,right).`
- `next(left,lleft).`
- `next(left,rleft).`
- `next(right,lright).`
- `next(right,rright).`
Predicate `next/2` moves one level down the tree. Reverse the direction

```prolog
prev(A,B) :- next(B,A).
```

Move two levels down

```prolog
next_next(A,C) :- next(A,B), next(B,C).
```

Go back, up to the top

```prolog
back(A,C) :- prev(A,C); ( prev(A,B), back(B,C) ).
```

Up and down – horizontal traversing the tree

```prolog
same_parent_node(A,B) :- next(P,A), next(P,B), A ≠ B.
```
Example 2. Tree with intersection points. The intersections cause, that going upwards, one can take more than one path in some nodes.

How to describe this type of tree?
Method 1. General, works always.

- `parent(adrahil2, finduilas).`
- `parent(ecthelion2, denethor2).`
- `parent(finduilas, boromir).`
- `parent(denethor2, boromir).`
- `parent(finduilas, faramir).`
- `parent(denethor2, faramir).`
  
  (...)

- `male(adrahil2).`
- `male(ecthelion2).`
- `male(denethor2).`
  
  (...)

- `female(finduilas).`
  
  (...)

- everything described separately
Method 2. General, works always, shorter.

father(adrahil2, finduilas).
father(ecthelion2, denethor2).
father(denethor2, faramir).
father(denethor2, boromir).

(...) 
mother(finduilas, boromir).
mother(finduilas, faramir).

(...) 
male(elboron).

male(X) :- father(X, _).
female(X) :- mother(X, _).

- database cannot be discontinuous
- the male/female information is partially coded in the father/mother predicates
Method 3. Works for full trees, meaning that each child has both parents listed in the tree.

```
parents(adrahil2,_,finduilas).  % notice the _!
parents(ecthelion2,_,denethor2).  % notice the _!
parents(denethor2,finduilas,boromir).
parents(denethor2,finduilas,faramir).
parents(eomund,thoedwyn,eowyn).
parents(faramir,eowyn,elboron).
male(elboron).
```

- this is the full database – most compact version
- due to the lack of certain information, underscores were used
- one needs to be careful when defining new predicates: for the first two entries prolog will substitute anything in the second field (every mother is OK)
- a way out is to code these exceptions separately case by case
**Problem:** traversing the tree. How to define siblings? Common mistakes:

- siblings have a common parent

\[
\text{siblings}(X,Y) :- \text{parent}(P,X), \text{parent}(P,Y), X \neq Y.
\]

→ will double the results if both parents present

- siblings have a common mother or father

\[
\text{siblings}(X,Y) :- \text{mother}(P,X), \text{mother}(P,Y), X \neq Y.
\]
\[
\text{siblings}(X,Y) :- \text{father}(P,X), \text{father}(P,Y), X \neq Y.
\]
\[
\text{siblings}(X,Y) :- ((\text{mother}(P,X), \text{mother}(P,Y)) ;
\]
\[
(\text{father}(P,X), \text{father}(P,Y))),
\]
\[
X \neq Y.
\]

→ this is equivalent to the first case – doubles the results
check one parent only, say, the mother

\[
\text{siblings}(X,Y) :- \text{mother}(P,X), \text{mother}(P,Y), X \neq Y.
\]

→ for trees where each child has a mother; a similar definition for fathers may be given, with the same assumption about the tree structure

check one common parent

\[
\text{siblings}(X,Y) :- \text{nonvar}(X),
\text{parent}(Z,X), !, \text{parent}(Z,Y), X \neq Y.
\]

\[
\text{siblings}(X,Y) :- \text{nonvar}(Y),
\text{parent}(Z,Y), !, \text{parent}(Z,X), X \neq Y.
\]

→ this definition works fine unless you ask for all siblings in the tree:

\[
| \text{?- siblings}(X,Y).
\]
KEEP CALM AND DO YOU HAVE ANY QUESTIONS