LISTS IN PROLOG

The concept of lists in Prolog

- The Prolog list
  - The empty list is represented by the atom `[]`. The non-empty list is a compound `'.'(Head,Tail)` where
    - `Head` is the head (first element) of the list, while
    - `Tail` is the tail of the list, that is the remaining elements of the list.
  - Lists can be written in simplified forms („syntactic sweetening”).
  - The implementation of the lists is optimised: it is more space- and time-efficient than for other compound structures.

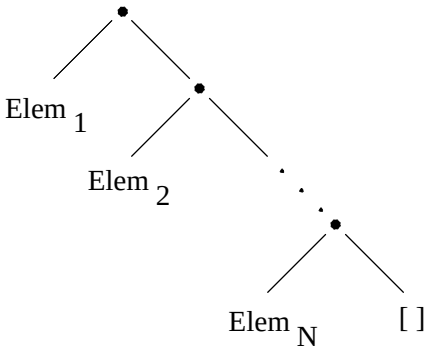
```
list_of_numbers(.(E,L)) :-
    number(E), list_of_numbers(L).
list_of_numbers([]).
```

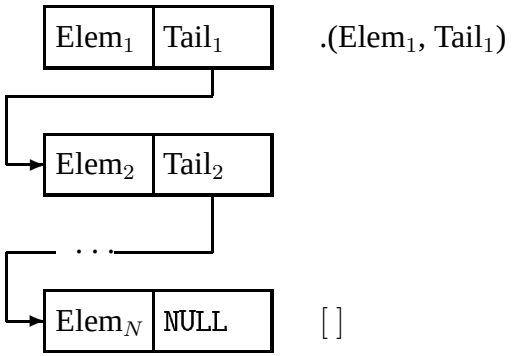
```
| ?- listing(list_of_numbers).
list_of_numbers([A|B]) :-
    number(A),
    list_of_numbers(B).
list_of_numbers([]).
```

```
| ?- list_of_numbers([1,2]).    %  [1,2] == .(1,.(2,[])) == [1|[2|[]]]
     yes
| ?- list_of_numbers([1,a,f(2)]).
     no
```

Various list representations

- Options for writing a lists of N elements:
  - canonical form : `.(`$\texttt{Elem}_1$`,.(`$\texttt{Elem}_2$`,...,.(`$\texttt{Elem}_N$`,[])...))`
  - equivalent list notation: `[`$\texttt{Elem}_1$`,`$\texttt{Elem}_2$`,...,`$\texttt{Elem}_N$`]`
  - less convenient list notation: `[`$\texttt{Elem}_1$`|[`$\texttt{Elem}_2$`,...,[ `$\texttt{Elem}_N$`|[] ] ...]]`
- The tree structure of lists and their implementation





The notation of lists — syntactic sweeteners

- the base sweetener: `[Head|Tail]` $\equiv$ `.(Head, Tail)`
- the N-fold application of the base sweetener without nested brackets:
  `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`|Tail]` $\equiv$ `[Elem`$_1$`|[Elem`$_2$`,...,Elem`$_N$`|Tail]]`
- when the tail is `[]`: `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`]` $\equiv$ `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`|[]]`

```
| ?- [1,2] = [X|Y].                  ⇒ X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].                  ⇒ X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].                ⇒ X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].                ⇒ no
| ?- [1,2,3,4] = [X,Y|Z].            ⇒ X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_].         ⇒ L = [1,2|_A] ? % open ended
| ?- L = .(1,[2,3|[]]).              ⇒ L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].              ⇒ L = [1,2,3] ?
| ?- [X|[3-Y/X|Y]]= .(A, [A-B,6]).  ⇒ A=3, B=[6]/3, X=3, Y=[6] ?
```

Ground and pattern-terms, list-patterns and open ended lists

- (Revision:) Ground term: a term which does not contain any variable
- Pattern: a (usually non-ground) term, which „represents” all the terms which can be derived from it by variable substitution.
- List-pattern: a pattern which represents a list (but possibly other terms as well).
- Open ended list: a list-pattern which represents lists of any length.
- Closed list: a list(-pattern) which represents lists of a given length.

Closed	Lists represented
`[X]`	one element lists
`[X,Y]`	two element lists
`[X,X]`	lists with two identical elements
`[X,1,Y]`	3 element lists, where element 2 is `1`

Open	Lists represented
`X`	any
`[X|Y]`	non-empty lists (with at least one element)
`[X,Y|Z]`	lists with at least 2 elements
`[a,b|Z]`	lists with at least 2 elements: `a, b, ...`

The logic variable

- The concept of logic variable:
  - can appear as a term, or in terms, cf. variables in (list) patterns
  - variables can be made identical (ie. unified): e.g. two identical variables in a term.
  - the variable is a „first class citizen” in the world of (sub)terms
- SML has pattern matching as well, but the pattern can only be used for decomposition, and not for construction of terms; the variables in patterns always obtain ground values.
- (Some new functional languages, e.g. the Oz language supports the logic variable.)
- Example: the goal below creates — in variable `L` — a list of two **identical** elements. The values of these elements will be **identical** to variable `X` in the goal.

```
first_elem([E|_], E).
second_elem([_,E|_], E).

| ?- first_elem(L, X), second_elem(L, X). ⟹ L = [X,X|_A] ? ; no
```

- If any of the three variables gets instantiated, all others will be substituted with the same value:

```
| ?- first_elem(L, X), second_elem(L, X), X = apple.
                              ⟹ X = apple, L = [apple,apple|_A] ? ; no
| ?- first_elem(L, X), second_elem(L, X), second_elem(L, wine)
                              ⟹ X = wine, L = [wine,wine|_A] ? ; no
```

Concatenating lists: the append/3 procedure

- `append(L1, L2, L3)`: List `L3` is composed of the elements of `L1` and `L2` put after each other (notation: `L3` = `L1`⊕`L2`) — two solutions:

```
append0([], L2, L) :- L = L2.
append0([X|L1], L2, L) :-
    append0(L1, L2, L3), L = [X|L3].
```

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

```
    > append0([1,2,3],[4],A)
(2) > append0([2,3],[4],B), A=[1|B]
(2) > append0([3],[4],C), B=[2|C], A=[1|B]
(2) > append0([],[4],D),C=[3|D],B=[2|C],A=[1|B]
(1) > D=[4], C=[3|D], B=[2|C], A=[1|B]
BIP > C=[3,4], B=[2|C], A=[1|B]
BIP > B=[2,3,4], A=[1|B]
BIP > A=[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

```
    > append([1,2,3],[4],A), write(A)
(2) > append([2,3],[4],B), write([1|B])
(2) > append([3],[4],C), write([1,2|C])
(2) > append([],[4],D), write([1,2,3|D])
(1) > write([1,2,3,4])
[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

- The complexity of `append0/append(L1, ...)`: run time is proportional to the length of list `L1`.
- Why is `append/3` better than `append0/3`?
  - `append/3` is **right recursive**, equivalent to a cycle (does not use the stack)
  - `append([1,...,1000],[0],[2,...])` fails immediately, `append0(...)` fails only after 1000 steps
  - `append/3` can be used for splitting lists as well (see later), while `append0/3` can not.

Building lists *from the beginning* — using open ended lists

- The append predicate creates — at the very first reduction — the head of the resulting list! (The output parameter is set to a list pattern with a yet unknown tail, cf. logic variables.)

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-     append(L1, L2, L3).
| ?- append([1,2,3], [4], Result) ⟹  Result = [1|A], append([2,3], [4], A)
```

- Advanced tracing options for demonstrating this
  - `library(debugger_examples)` —programming the tracer, defining new debugger commands
  - new command: '`N` ⟨ name ⟩' —labels the argument at focus
  - standard command: '`^` ⟨ arg number ⟩' —focuses on a given argument
  - new command: '`P` [⟨ name ⟩]' —prints out the named terms (the one specified or all)

```
| ?- use_module(library(debugger_examples)).
| ?- trace, append([1,2,3],[4,5,6],A).
        1      1 Call: append([1,2,3],[4,5,6],_543) ? ^ 3
        1      1 Call: ^3 _543 ? N Result
        1      1 Call: ^3 _543 ? P                      ⇒  Result = _543
        2      2 Call: append([2,3],[4,5,6],_2700) ? P  ⇒  Result = [1|_2700]
        3      3 Call: append([3],[4,5,6],_3625) ? P    ⇒  Result = [1,2|_3625]
        4      4 Call: append([],[4,5,6],_4550) ? P     ⇒  Result = [1,2,3|_4550]
        4      4 Exit: append([],[4,5,6],[4,5,6]) ? P   ⇒  Result = [1,2,3,4,5,6]
        3      3 Exit: append([3],[4,5,6],[3,4,5,6]) ?
        2      2 Exit: append([2,3],[4,5,6],[2,3,4,5,6]) ?
        1      1 Exit: append([1,2,3],[4,5,6],[1,2,3,4,5,6]) ?
A = [1,2,3,4,5,6] ? ; no
```

Reversing lists

- Naive solution (quadratic in the length of the list)

```
% nrev(L, R): List R is the reverse of list L.
nrev([], []).
nrev([X|L], R) :-
    nrev(L, RL),
    append(RL, [X], R).
```

- A solution which is linear in the length of the list

```
% reverse(R, L): List R is the reverse of list L.
reverse(R, L) :-  revapp(L, [], R).

% revapp(L1, L2, R): The reverse of L1 prepended to  L2 gives R.
revapp([], R, R).
revapp([X|L1], L2, R) :-
    revapp(L1, [X|L2], R).
```

- The `lists` library contains the definitions of procedures `append/3` and `reverse/2`.
- Loading the library:

```
:- use_module(library(lists)).
```

append and revapp — building lists in two directions

- Prolog implementation

```
append([], L, L).
append([X|L1], L2, [X/L3]) :-
    append(L1, L2, L3).
```

```
revapp([], L, L).
revapp([X|L1], L2, L3) :-
    revapp(L1, [X/L2], L3).
```

- C++ implementation

```
struct link   { link *next;
                char elem;
                link(char e): elem(e) {}
              };
typedef link *list;
```

```
list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}
```

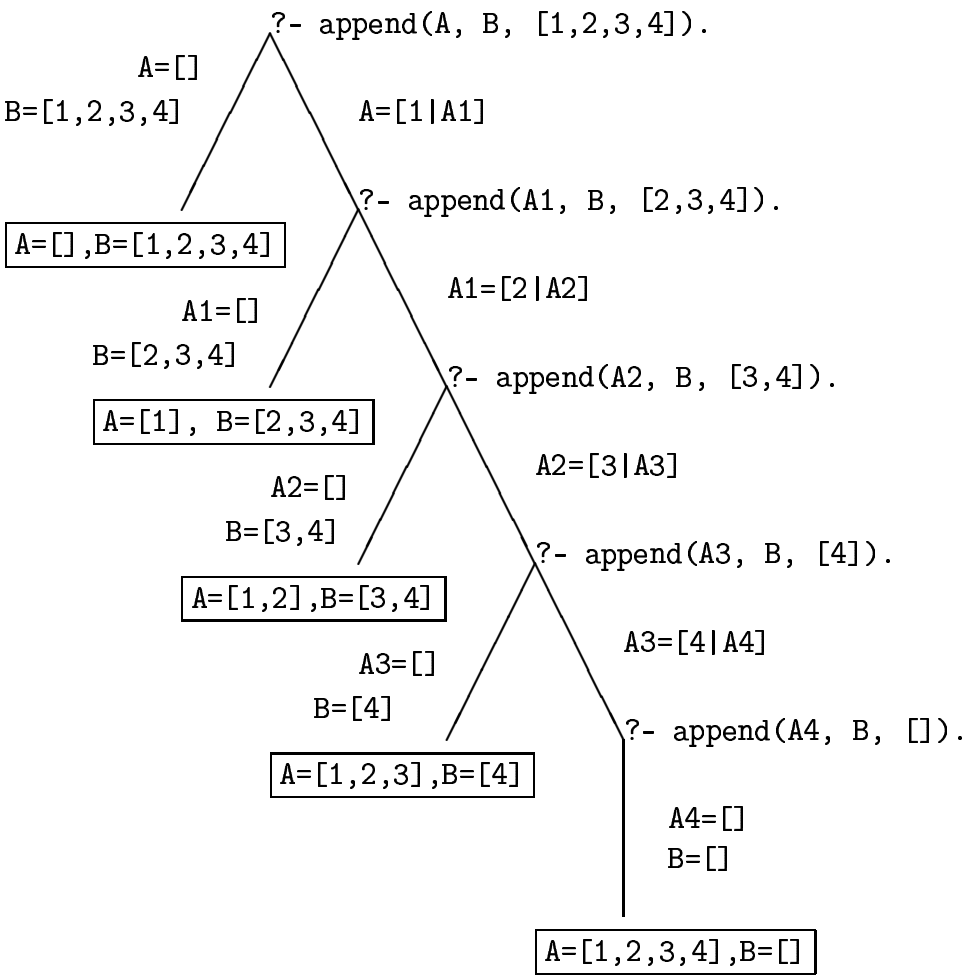
```
list revapp(list list1, list list2)
{ list l = list2;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }

  return l;
}
```

Splitting lists using append/3

```
% append(L1, L2, L3):
% The L3 list is made by
% concatenating elements
% of lists L1 and L2.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

| ?- append(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```



Variations on append 1. — Appending three lists

- The search space of `append/3` is **finite**, if the first **or** the third argument is a closed list (or both).
- `append(L1,L2,L3,L123)`: `L1` $\oplus$ `L2` $\oplus$ `L3` = `L123`

```
append(L1, L2, L3, L123) :-
        append(L1, L2, L12), append(L12, L3, L123).
```

  - Not efficient, eg.: `append([1,...,100],[1,2,3],[1], L)` uses 203 steps instead of 103!
  - Not suitable for splitting lists — creates infinite choice point
- An efficient version, suitable for splitting a given list to three parts:

```
% L1 ⊕ L2 ⊕ L3 = L123, where either L1 and L2, or L123 is given (is a closed list).
append(L1, L2, L3, L123) :-
        append(L1, L23, L123), append(L2, L3, L23).
```

  - The first `append/3` call produces an open ended list:

    `| ?- append([1,2], L23, L).` $\Rightarrow$ `L = [1,2|L23] ?`
  - The instantiation of `L3`, i.e. whether it is open or closed, does not matter.

Pattern search in lists using append/3

- Elements occuring in pairs

```
% in_pair(List, Elem):  Elem is an element of List
% which has  an identical right neighbour in the list.
in_pair(L, E) :-
    append(_, [E,E|_], L).

| ?- in_pair([1,8,8,3,4,4], E).
        E = 8 ? ; E = 4 ? ; no
```

- Stuttering sublists

```
% stuttering(L, D): D is a nonempty sublist of L,
% which is followed by a sublist identical to it.
stuttering(L, D) :-
        append(_, Tail, L),
        D = [_|_],
        append(D, End, Tail),
        append(D, _, End).

| ?- stuttering([2,2,1,2,2,1], D).
        D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

Search in lists

- `member(E, L)`: `E` is the element of list `L`

```
member(Elem, [Elem|_]).
member(Elem, [_|Tail]) :-
        member(Elem, Tail).
```

```
member(Elem, [Head|Tail]) :-
        (   Elem = Head
        ;   member(Elem, Tail)
        ).
```

- Possible uses of `member/2`
  - A Yes-No question:

```
| ?- member(2, [1,2,3]).          ⇒   yes
```

  - Enumerating list elements:

```
| ?- member(X, [1,2,3]).          ⇒   X = 1 ? ; X = 2 ? ; X = 3 ?  ; no
| ?- member(X, [1,2,1]).          ⇒   X = 1 ? ; X = 2 ? ; X = 1 ?  ; no
```

  - Enumerating the common elements of lists — uses both above call-patterns:

```
| ?- member(X, [1,2,3]),
     member(X, [5,4,3,2,3]).      ⇒   X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```

  - Making a term an element of a list — creates an infinite choice!

```
| ?- member(1, L).                ⇒   L = [1|_A] ? ; L = [_A,1|_B] ? ;
                                      L = [_A,_B,1|_C] ? ; ...
```

- The search space of `member/2` is **finite**, if the second argument is a closed list.

Generalization of `member/2`: `select/3`

- `select(Elem, List, Rest)`: Removing `Elem` from `List` results in list `Rest`.

```
select(Elem, [Elem|Rest], Rest).     % The head is removed, the tail remains.
select(Elem, [X|Tail], [X|Rest0]) :- % The head  remains,
        select(Elem, Tail, Rest0).   %    the element is removed from the Tail.
```

- Possible uses:

```
| ?- select(1, [2,1,3], L).            % To remove a given element
        L = [2,3] ? ; no
| ?- select(X, [1,2,3], L).            % To remove an arbitrary element
        L=[2,3], X=1 ? ;  L=[1,3], X=2 ? ;   L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).              % To insert a given element!
        L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
                                       % Can 3 be inserted into [1,...]
        no                             % so, that we get [2,...]?
| ?- select(1, [X,2,X,3], L).
        L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- Library `lists` contains the definitions of procedures `member/2` and `select/3`.
- The search space of `select/3` is **finite**, if the 2nd **or** the 3rd argument is a closed list.

Permutation of lists

- `permutation(List, Perm)`: the permutation of `List` is list `Perm` (definition is directly from `library(lists)`:

```
permutation([], []).
permutation(List, [First|Perm]) :-
        select(First, List, Rest),
        permutation(Rest, Perm).
```

- Possible uses:

```
| ?- permutation([1,2], L).
        L = [1,2] ? ; L = [2,1] ? ; no

| ?- permutation([a,b,c], L).
        L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
        L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ;
        no

| ?- permutation(L, [1,2]).
        L = [1,2] ? ;
        infinite search space
```

- If the first argument in `permutation/2` is unknown: the search space of the `select` call is infinite!

TYPES IN PROLOG

Binary tree

- Different definitions of a binary tree data type:
  - Textual definition (repetition): A binary tree of integers can be
    - either a leaf (`leaf(V)`), where `V` is an integer
    - or a node (`node(L,R)`), where `L` and `R` are binary trees of integers
  - Using mathematical notation:

    `itree` $\equiv$ {`leaf(`i`)` | $i \in$ `int` } $\cup$ { `node(`l`,`r`)` | $l, r \in$ `itree` }
  - Using a type-notation:

```
:- type itree ==  {node(itree, itree)} \/ {leaf(int)}.
:- type itree --->  node(itree, itree) | leaf(int).
```

  - A Prolog predicate for checking whether a term belongs to a data type:

```
itree(leaf(V)) :-
        integer(V).
itree(node(L,R)) :-
        itree(L), itree(R).
```

- Such a datatype is called **discriminated union**, because the sets in the union are distinguished by the functors of their elements (`leaf/1`, `node/2`)

Description of types in Prolog

- Type description: a definition of a set of (ground) Prolog terms
- Basic type descriptions: `int, float, number, atom, any`
- Building new types:

  `{ str(`$\mathtt{T}_1$`, ..., `$\mathtt{T}_n$`) }` $\equiv$ `{ str(`e_1`, ..., `e_n`) | `$e_1 \in \mathtt{T}_1$`, ..., `$e_n \in \mathtt{T}_n$ `}`, $n \geq 0$

  Example: `{person(atom,atom,int)}` is the set of structures with the functor `person/3`, whose first two arguments are atoms, the third is an integer.
- Union of types (viewed as sets) can be created using the `\/` operator.

  ```
  {person(atom,atom,int)} \/ {atom-atom} \/ atom
  ```
- A type declaration can be named (in a comment): `:- type` *tname* `==` *tdescription*`.`

  ```
  :- type t1 == {atom-atom} \/ atom.,
  :- type man == {man-atom} \/ {nothing}.
  ```
- Discriminated union: a union of such complex types which have different functors. If S_1,..., S_n have different functors, the simplified (Mercury) notation can be used:

  `:- type` T `==` `{` S_1 `} \/ ...\/ {` S_n `}.` $\Rightarrow$ `:- type` T `--->` S_1 `;` `...;` S_n`.` Examples:

  ```
  :- type man ---> man-atom; nothing.
  :- type tree ---> leaf(int) ; node(tree,tree).
  ```

Type description in Prolog — continued

- Parametric types — examples

```
:- type pair(T1, T2) ---> T1 - T2.          % a structure '-' with two arguments,
                                            %     first arg. T1, the second T2 type.
:- type tree(T) ---> leaf(T)                % Binary tree made up of elements with
          ; node(tree(T),tree(T)).          %     type T
:- type assoc_tree(KeyT, ValueT)            % Tree of pairs made of
          == tree(pair(KeyT, ValueT)).      %     KeyT and ValueT types
:- type dictionary == assoc_tree(word, word).
:- type word == atom.
```

- Syntax of type declarations

⟨type declaration⟩ ::= ⟨named type⟩ | ⟨type construction⟩
⟨named type⟩ ::= `:- type` ⟨type id⟩ `==` ⟨type description⟩ .
⟨type construction⟩ ::= `:- type` ⟨type id⟩ `--->` ⟨discriminated union⟩ .
⟨discriminated union⟩ ::= ⟨constructor⟩ ; ...
⟨constructor⟩ ::= ⟨name constant⟩ | ⟨structure name⟩`(`⟨type description⟩, ...`)`
⟨type description⟩ ::= ⟨type id⟩ | ⟨type variable⟩ | `{` ⟨constructor⟩ `}` |
⟨type description⟩ `\/` ⟨type description⟩
⟨type id⟩ ::= ⟨type name⟩ | ⟨type name⟩`(`⟨type variable⟩, ...`)`
⟨type name⟩ ::= ⟨name constant⟩
⟨type variable⟩ ::= ⟨variable⟩

Declaration of predicate-type

- Declaring the argument types of a predicate
  `:-` `pred` ⟨procedure name⟩`(`⟨type id⟩, ...`)`
- Example:

```
:- pred sum_tree(tree(int), int).
```

- Declaring the modes of predicate arguments (Optional, multiple declarations allowed.)
  `:-` `mode` ⟨procedure name⟩`(`⟨mode id⟩, ...`)` ahol ⟨mode id⟩ ::= `in` | `out` | `inout`.
- Examples:

```
:- mode sum_tree(in, in).   % checking the sum of a tree
:- mode sum_tree(in, out).  % calculating the sum of a tree
:- mode sum_tree(out,in).   % building a tree with a given sum
```

- Mixed type- and mode declarations
  `:-` `pred` ⟨procedure name⟩`(`⟨type id⟩`::`⟨mode id⟩, ...`)`
- Example:

```
:- pred between(int::in, int::in, int::out).
```

Mode declarations: the notation used in the SICStus manual

- The SICStus manual uses a different notation for marking the in/out arguments, such as:

```
sum_tree(+T, ?Sum).
```

- Mode notation:
  - `+` input argument (ground)
  - `-` out argument (non-ground)
  - `:` procedure argument (in meta-procedures)
  - `?` any

THE PROLOG SYNTAX

The summary of Prolog syntax

- The principles of Prolog syntax
  - All program elements are terms!
  - The necessary connectives (`','`, `;`, `:-` `-->`) are standard operators.
  - We classify the program elements according to their functor:
    - *query:* `?- Goal.`
      Goal is run, and the variable substitutions are displayed (this is the default in the so called top-level interactive shell).
    - *command:* `:- Goal.`
      The *Goal* is run silently. Use: eg. for placing declarations (operator, ...).
    - *rule:* `Head :- Body.`
      The rule is added to the program.
    - *grammar rule:* `Head --> Body.`
      The grammar rule is transformed to a Prolog clause and is added to the program (see DCG grammars).
    - *fact:* `All other terms.`
      Is added to the program as a rule with an empty body.

Variants of the Prolog language

- Two modes of execution in the SICStus system
  - `iso` — Corresponds to the ISO Prolog standard.
  - `sicstus` — Compatible with earlier versions.
  - Switching between execution modes: `set_prolog_flag(language, `*`Mode`*`)`.
  - Differences:
    - Minor syntactic details, like the `0x1ff` hex format for numbers is available only in ISO mode,
    - Minor differences in the behaviour of in-built predicates.
  - No differences in case of the predicates described so far.

Syntactic sweeteners — summary, practical advises

- Canonical form of expressions involving operators:
  - Enclose the subterms in parentheses according to operator priority and kind, e.g. `-a+b*2` ⇒ `((-a)+(b*2))`.
  - Transform the term to canonical form:
    `(A Inf B)` ⇒ `Inf(A,B)`, `(Pref A)` ⇒ `Pref(A)`, `(A Postf)` ⇒ `Postf(A)`
    Example: `((-a)+(b*2))` ⇒ `(-(a)+ *(b,2))` ⇒ `+(-(a),*(b,2))`.
  - Tricky cases:
    - The comma, when used as an atom, should be quoted: eg. `(pp,(qq;rr))` ⇒ `','(pp,;(qq,rr))`.
    - `-` *Number* ⇒ negative number constant, but `-` *Other* ⇒ prefix form.
      Example: `-1+2` ⇒ `+(-1,2)`, but `-a+b` ⇒ `+(-(a),b)`.
    - *Name*`(...)` ⇒ compound term;
      Name `(...)` ⇒ a term with a prefix operator. Examples:
      `-(1,2)` ⇒ `-(1,2)` (unchanged), but
      `- (1,2)` ⇒ `-(',',(1,2))`.

Syntactic sweeteners — lists, others

- Transforming lists to their canonical form.
  - Insert an empty list as a tail, where needed:
    `[1,2]` ⇒ `[1,2|[]]`. `[[X|Y]]` ⇒ `[[X|Y]|[]]`
  - (Repeatedly) eliminate the commas: `[Elem1,Elem2...]` ⇒ `[Elem1|[Elem2...]]`.
    `[1,2|[]]` ⇒ `[1|[2|[]]]`
    `[1,2,3|[]]` ⇒ `[1|[2,3|[]]]` ⇒ `[1|[2|[3|[]]]]`
  - Transform to canonical form: `[Head|Tail]` ⇒ `.(Head,Tail)`.
    `[1|[2|[]]]` ⇒ `.(1,.(2,[]))`, `[[X|Y]|[]]` ⇒ `.(.(X,Y),[])`
- Other syntactic sweeteners:
  - Character-code notation: `0'Char`.
    `0'a` ⇒ `97`, `0'b` ⇒ `98`, `0'c` ⇒ `99`, `0'd` ⇒ `100`, `0'e` ⇒ `101`
  - String: `"xyz..."` ⇒ is the list containing the character codes of `xyz...`
    `"abc"` ⇒ `[97,98,99]`, `""` ⇒ `[]`, `"e"` ⇒ `[101]`
  - Curly brackets: `{Expr}` ⇒ `{}(Expr)` (a structure with name `{}` and one argument — the `{}` pair of characters is a lexical element on its own, namely a name constant).
  - Binary, hexa etc. notation (only in `iso` mode), eg. `0b101010`, `0x1a`.

Syntax of terms — two level grammars

- An excerpt from the syntactic description of terms, in a „traditional” language:

⟨ term ⟩ ::= ⟨ member ⟩
| ⟨ term ⟩ ⟨ additive operator ⟩ ⟨ member ⟩

⟨ member ⟩ ::= ⟨ factor ⟩
| ⟨ member ⟩ ⟨ multiplicative operator ⟩ ⟨ factor ⟩

⟨ factor ⟩ ::= ⟨ number ⟩ | ⟨ identifier ⟩ | (⟨ term ⟩)

- The same with a two level grammar:

⟨ term ⟩ ::= ⟨ term *2* ⟩

⟨ term *N* ⟩ ::= ⟨ term *N-1* ⟩
| ⟨ term *N* ⟩ ⟨ operator of priority *N* ⟩ ⟨ term *N-1* ⟩

⟨ term *0* ⟩ ::= ⟨ number ⟩ | ⟨ id ⟩ | (⟨ term *2* ⟩)

{the priority of additive and. multiplicative operators is *2* and *1*, resp.}

Syntax of Prolog terms

⟨ program element ⟩ ::= ⟨ term 1200 ⟩ ⟨ full stop ⟩

⟨ term *N* ⟩ ::=
⟨ op *N* fx ⟩ ⟨ layout ⟩ ⟨ term *N-1* ⟩
| ⟨ op *N* fy ⟩ ⟨ layout ⟩ ⟨ term *N* ⟩
| ⟨ term *N-1* ⟩ ⟨ op *N* xfx ⟩ ⟨ term *N-1* ⟩
| ⟨ term *N-1* ⟩ ⟨ op *N* xfy ⟩ ⟨ term *N* ⟩
| ⟨ term *N* ⟩ ⟨ op *N* yfx ⟩ ⟨ term *N-1* ⟩
| ⟨ term *N-1* ⟩ ⟨ op *N* xf ⟩
| ⟨ term *N* ⟩ ⟨ op *N* yf ⟩
| ⟨ term *N-1* ⟩

⟨ term 1000 ⟩ ::= ⟨ term 999 ⟩ , ⟨ term 1000 ⟩

⟨ term 0 ⟩ ::=
⟨ name ⟩ (⟨ arguments ⟩)
{ The (immediately follows the ⟨ name ⟩ !}
| (⟨ term 1200 ⟩) | { ⟨ term 1200 ⟩ }
| ⟨ list ⟩ | ⟨ string ⟩
| ⟨ name ⟩ | ⟨ number ⟩ | ⟨ variable ⟩

Syntax of terms — continued

⟨ op *N K* ⟩ ::= ⟨ name ⟩ {if ⟨ name ⟩ was previously declared an operator with priority *N* and kind *K* }

⟨ arguments ⟩ ::= ⟨ term 999 ⟩
| ⟨ term 999 ⟩ , ⟨ arguments ⟩

⟨ list ⟩ ::= []
| [⟨ listexpr ⟩]

⟨ listexpr ⟩ ::= ⟨ term 999 ⟩
| ⟨ term 999 ⟩ , ⟨ listexpr ⟩
| ⟨ term 999 ⟩ | ⟨ term 999 ⟩

⟨ number ⟩ ::= ⟨ unsigned number ⟩
| + ⟨ unsigned number ⟩
| - ⟨ unsigned number ⟩

⟨ unsigned number ⟩ ::= ⟨ natural number ⟩
| ⟨ float number ⟩

Syntax of terms — comments

- In ⟨ term N ⟩ the ⟨ layout ⟩ is needed only if the term following it starts with an opening bracket.

```
| ?- op(500, fx, succ).
yes
| ?- write_canonical(succ (1,2)), nl, write_canonical(succ(1,2)).
succ(',')(1,2))
succ(1,2)
```

- The { ⟨ term ⟩ } is equivalent with the {}(⟨ term ⟩) structure, this is important, e.g. for the DCG grammar notation.

```
| ?- write_canonical({a}).
{}(a)
```

- ⟨ string ⟩ is a sequence of characters enclosed in double quotes (" characters) — by default this is equivalent to the list of codes of these characters.

```
| ?- write("baba").
[98,97,98,97]
```

The lexical elements of Prolog 1. (repetition)

- ⟨name⟩
  - a sequence of alphanumeric characters starting with a lower case letter (lower and upper case letters, digits and underline characters are allowed as alphanumeric characters);
  - a sequence of one or more graphic characters (`+-*/\$^<>=‘~:.?@#&`);
  - the `!` or `;` characters on their own;
  - the `[]` `{}` character pairs;
  - any sequence of characters in between single quotes (`’`), in which escape-sequences starting with a backslash (`\`) can be placed.
- ⟨variable⟩
  - a sequence of alphanumeric characters starting with a capital letter or an underline.
  - Variables having the same character sequence are considered the same, if they occur in the same clause, otherwiaw they are considered different;
  - exception: all occurrences the void variable (`_`) are different.

The lexical elements of Prolog 2.

- ⟨ natural number ⟩
  - a sequence of (decimal) digits ;
  - a sequence of binary, octal or hexadeximal digits, denoting an integer in the appropriate base, in such cases the number should be prefixed with the characters `0b`, `0o`, `0x` (only available in `iso` mode)
  - character code constant of the form `0'`*c* where *c* is a single character (or an escape-sequence denoting a single character)
- ⟨ floating point number ⟩
  - compulsorily has to contain a decimal point
  - at least one (decimal) digit on both sides of the point
  - an optional expontent indicated by the letter `e` or `E`

Comments and formatting characters

- Comments
  - From the % percentage sign until the end of the line
  - or from the /* pair of characters up until the nearest */ sequence of characters.
- Layout
  - space, new line, tabulator etc. (non-visible characters)
  - comment
- Formatting of the program text
  - Layout (space, new line etc.) can be placed freely;
  - exception: no layout character should be placed between the name of a structure and the subsequent open parenthesis;
  - it is compulsory to place layout between a prefix operator and a (;
  - ⟨ full stop ⟩: a . character followed by layout.

PROLOG EXAMPLES

The introductory example of the old lecture book: path search

- The task:
  - Let's consider a set of (bus)lines.
  - The two endpoints and the lengths of all the lines are given.
  - Let's write a Prolog procedure which determines whether two points can be connected with exactly N joining lines.
- Rewriting: we search for a path between two points in a weighted and undirected. graph Edges:

```
% line(A, B, H): There is a line between towns A and B  and its length is H km.
line('Budapest', 'Prague', 515).
line('Budapest', 'Vienna', 245).
line('Vienna', 'Berlin', 635).
line('Vienna', 'Paris', 1265).
```

- Directed edges

```
% way(A, B, H): We can get from A to B with a line length of H.
way(A, B, H) :-
     (   line(A, B, H)
     ;   line(B, A, H)
     ).
```

Path search task — continuation

- Path with given number of steps (edge-sequence) and its length:

```
% path(N, A, B, H): There exists a path cosisting of (exactly)
% N sections which has a length of H.
path(0, To, To, 0).
path(N, From, To, H) :-
    N > 0,
    N1 is N-1,
    way(From, Between, H1),
    path(N1, Between, To, H2),
    H is H1+H2.
```

- An example:

```
| ?- path(2, 'Paris', To, H).
        H = 1900, To = 'Berlin' ? ;
        H = 2530, To = 'Paris' ? ;
        H = 1510, To = 'Budapest' ? ;
        no
```

Acyclical path search

- Loading the library to import procedures with given functors.

```
:- use_module(library(lists), [member/2]).
```

- Help-argument: the list of the concerned towns in reverse order

```
% path_2(N, A, B, H):There exists an  acyclical path cosisting of
% (exactly) N sections which has a length of H.
path_2(N, From, To, H) :-
    path_2(N, From, To, [From], H).

% path_2(N, A, B, Excluded, H): There exists an  acyclical path
% consisting of (exactly) N sections not going through the towns in
% Excluded which has a length of H.
path_2(0, To, To, Excluded, 0).
path_2(N, From, To, Excluded, H) :-
    N > 0, N1 is N-1, way(From, Between, H1),
    \+ member(Between, Excluded),
    path_2(N1, Between, To, [Between|Excluded], H2), H is H1+H2.
```

- An example run:

```
| ?- path_2(2, 'Paris', To, H).
        H = 1900, To = 'Berlin' ? ;
        H = 1510, To = 'Budapest' ? ; no
```

Further improvement: acyclical path search with route-gathering

- The basic idea: the (reversed) route is gathered in the `Excluded` list.
- A **new argument** is needed in the recursive procedure to present the route!

```
:- use_module(library(lists), [member/2, reverse/2]).

% path_3(N, A, B, Way, H): There exists an acyclical  Way route
% between A and B consisting of (exactly) N sections which has a length of H.
path_3(N, From, To, Way, H) :-
    útvonal_3(N, From, To, [From], RWay, H),
    reverse(RWay, Way).

% path_3(N, A, B, RWay0, RWay, H): There exists an acyclical route
% between A and B consisting of (exactly) N sections which has a length of H and
% not going through RWay0.
% RWay = (the A → B path reversed leads to ) ⊕ RWay0.
path_3(0, To, To, RevWay, RevWay, 0).
path_3(N, From, To, RevWay0, RevWay, H) :-
    N > 0, N1 is N-1, way(From, Between, H1),
    \+ member(Between, RevWay0),
    path_3(N1, Between, To, [Between|RevWay0], RevWay, H2), H is H1+H2.
```

```
| ?- path_3(2, 'Paris', _, Út, H).
        H = 1900, Út = ['Paris','Wiena','Berlin'] ? ;
        H = 1510, Út = ['Paris','Wiena','Budapest'] ? ; no
```

Representation of weighted graph with edgelist

- The representation of graph
  - the graph is a list of edges,
  - the edge is a structure with three arguments,
  - arguments: the two end-points and the weight.
- Type definition

```
% :- type edge ---> edge(point, point, weight).
% :- type point == atom.
% :- type weight == int.
% :- type graph == list(edge).
```

- Example

```
net([edge('Budapest','Wiena',245),
         edge('Budapest','Prague',515),
         edge('Wiena','Berlin',635),
         edge('Wiena','Paris',1265)]).
```

Searching path free from repetition in a graph represented by list

```
:- use_module(library(lists), [select/3]).

% path_4(N, G, A, B, L, H): There exists an L path consisting of N parts
% going from A to B in the G graph with H length.
ossza H.
path_4(0, _Graph, To, To, [To], 0).
path_4(N, Graph, From, To, [From|Route], H) :-
        N >0, N1 is N-1,
        select(Edge, Graph, Graph1),
        edge_endpoints_length(Edge, Honnan, Between, H1),
        path_4(N1, Graph1, Between, To, Route, H2),
        H is H1+H2.

% edge_endpoints_length(Edge, A, B, H): The endpoints of
% Edge undirected edge is A and B, legth H.
edge_endpoints_length(edge(A,B,H), A, B, H).
edge_endpoints_length(edge(A,B,H), B, A, H).

| ?- net(_Graph), path_4(2, _Graph, 'Budapest', _, Route, H).
        H = 880, Route = ['Budapest','Wiena','Berlin'] ? ;
        H = 1510, Route = ['Budapest','Wiena','Paris'] ? ;
        no
```

The treatment of binary trees — the leaf of the tree

- Let's write a predicate to decide whether a given value exists in a leaf of the tree!
- ```
  % leaf_of_tree(Tree, Value): The Tree binary tree contains a leaf with value Value.
  leaf_of_tree(leaf(V), V).      % if the tree is only a leaf and the value inside

                                 % is equal with the searched one then "true"
  leaf_of_tree(node(L,_), V) :-
         leaf_of_tree(L, V).     % if the Value is contained in the left tree, than the whole tree cont
  leaf_of_tree(node(_,R), V) :-
         leaf_of_tree(R, V).     % if the Value is contained in the right tree, than the whole tree cor
  ```
- The underbar is a void variable, all of it's appearance are different variables!
- Examples: checking (1), listing the leaves of a tree(2),
  listing give leaves of a tree, (3) (∞ searching space).

```
| ?- leaf_of_tree(node(node(leaf(1),leaf(2)),leaf(7)), 2). ⟹ yes                    (1)
| ?- leaf_of_tree(node(node(leaf(1),leaf(2)),leaf(7)), 3). ⟹ no                     (1)
| ?- leaf_of_tree(node(leaf(1),leaf(7)), E). ⟹ E = 1 ? ; E = 7 ? ; no               (2)
| ?- leaf_of_tree(Tree, 3). ⟹ Tree = leaf(3) ? ; Tree = node(leaf(3),_A) ? ;  ...  (3)
```
```

Composite data structure with conjuctive and disjuctive ingress

- A composite data structure can be built up in two ways:
  - konjunktívan: the parts are walked in with AND connection, usually give one result
    - like: the summ of tree (`sum_tree`), tree checking (`itree`), tree printing:

```
% treeout(Tree): Tree printing (always true :-). Prints out the tree as a side effect.
treeout(leaf(V)) :-
        write(@), write(V).     % The write(X) is an in-built pred., prints the X term out.
treeout(node(L,R)) :-
        write('('), faki(L), write(' -- '), treeout(R), write(')').

| ?- treeout(node(node(leaf(1),leaf(8)),leaf(7))).      ⇒ ((@1 -- @8) -- @7)
yes
```

  - disjunctive: the parts are walked in with OR connection, new result at backtracking
    - like: listing of tree leaves (`tree_leaf`)
- The disjunctive, listing walk in can be complemeted with new conditions easily
  - We search the leaves of a tree within the `(5,10)` interval:

```
| ?- _Tree = node(node(leaf(1),leaf(8)),leaf(7)), leaf_of_tree(_Tree, E), 5 < E, E < 10.
     ⟹ E = 8 ? ; E = 7 ? ; no
| ?- _Tree = (...),  leaf_of_tree(_Tree, E), 5 < E, E < 10, write(E), write(' '), fail.
     ⇒ 8 7  ⟹ no
```

- The `fail` in-built predicate always „fails” , like it is good for organising of backtracking cycle.

Omitting leaves from the binary tree

- Let us write a new predicate to decide whether a given value exists in a leaf of a compound tree! The predicate hands back a new tree made of the original but with omission of the found leaf.

```
% tlr(Tree, Value, Remains): With the omission of the Value value leaf of
% the Tree compound binary tree Remains is tree what remains. (tlr = tree_leaf_remain)
tlr(node(leaf(V),T), V, T).    % if the left tree part is the wanted leaf
                               % then the right tree part is the Remaining tree
tlr(node(T,leaf(V)), V, T).    % the same for the right leaf case
tlr(node(L0,R), V, node(L,R)) :-
        tlr(L0, V, L).         % if the leaf can be ommited from the left tree
                               % then the remaining of it, completed with
                               % the right tree, will be the remains of the
                               % whole tree
tlr(node(L,R0), V, node(L,R1)) :-
        tlr(R0, V, R1).        % the same for right tree part
```

- The `tlr/3` predicate can be used for checking, but for decomposing of the tree as well:

```
| ?- tlr(node(leaf(1),node(leaf(2),leaf(3))), 2, T).  ⟹
   T = node(leaf(1),leaf(3)) ? ; no
| ?- tlr(node(leaf(1),node(leaf(2),leaf(3))), 7, T).  ⟹ no
| ?- tlr(node(leaf(1),node(leaf(2),leaf(3))), X,  T). ⟹
   T = node(leaf(2),leaf(3)), X = 1 ? ;
```

```
T = node(leaf(1),leaf(3)), X = 2 ? ;
T = node(leaf(1),leaf(2)), X = 3 ? ; no
```

Insertion of leaf in binary tree

- Let us write a predicate to insert a leaf with given value to a tree in all possible ways!
- We don't have to write it, we have already done so! The `tlr` predicate is good for this as well:

```
% tlr(Tree, Value, Remains): The leaf with Value value of Tree binary tree
% is omitted, Remains tree remains from the Tree. Short: Tree - Value = Remains.

% tlr(Tree, Value, Remains): The Tree (compound) binary tree is built of
% inserting a Value leaf into the Remains tree. Tree = Remains + Value.
tlr(node(leaf(V),T), V, T).    % A leaf inserted into T tree
(...)                          % so that the one leafed tree is put before T
```

- Examples:

```
| ?- tlr(Tree, 2, leaf(1)), treeout(Tree), write(' '), fail.
(@2 -- @1) (@1 -- @2)                          ⟹ no
| ?- tlr(Tree0, 2, leaf(1)), tlr(Tree, 3, Fa0), treeout(Tree), write(' '), fail.
(@3 -- (@2 -- @1)) ((@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)
(@2 -- (@3 -- @1)) (@2 -- (@1 -- @3)) (@3 -- (@1 -- @2)) ((@1 -- @2) -- @3)
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- (@3 -- @2)) (@1 -- (@2 -- @3))  ⟹ no

fourleafed(X, Y, Z, U, Fa) :-  % Tree is made of X, Y, Z, U leaves
     tlr(Fa0, Y, leaf(X)), tlr(Fa1, Z, Fa0), tlr(Fa, U, Fa1).

| ?- findall(Fa, fourleafed(1,3,4,6,Fa), Fak), length(Fak,Db). ⟹  Db = 120, Fak = (...)
```

Example: producing a term with give value

- The task: write a Prolog program for the following problem:
  - With the use of numbers 1, 3, 4, 6 and the four basic arithmetic operators produce the 24 numerical value!
  - All the four numbers have to be used, in any order.
  - Any operators of the four basic arithmetic operations with any parenthesis.
- We already have a predicate (`fourleafed/5`) building any tree from four numbers.
- Define a predicate, which builds arithmetical term from a tree!

```
% tree_term(Tree, Term): Term is an arithmetical term, equivalent
% in shape with the Tree, and built of the same numbers and of
% the four basic operators.
tree_term(leaf(V), V).
tree_term(node(L,R), Exp) :-
        tree_term(L, E1),
        tree_term(R, E2),
        base4(E1, E2, Exp).

% base4(X, Y, Term): Term is built of X and Y with one of
% the four basic operators.
base4(X, Y, X+Y).                       base4(X, Y, X-Y).
base4(X, Y, X*Y).                       base4(X, Y, X/Y).
```

```
| ?- tree_term(node(leaf(1),node(leaf(2),leaf(3))), Expr).
Expr = 1+(2+3) ? ; Expr = 1-(2+3) ? ; Expr = 1*(2+3) ? ; Expr = 1/(2+3) ? ;
(...)
Expr = 1+2/3 ?   ; Expr = 1-2/3 ?   ; Expr = 1*(2/3) ? ; Expr = 1/(2/3) ? ; no
```

Example: producing a term with given value (continuation)

- Earlier mentioned predicates:
  - `fourleafed/5` listing the trees made of 4 given values
  - listing the arithmetical terms equal with the a tree `tree_term/2`
- Based of these the solution of the task we can easily write the needed predicate:

```
% Term is a term built of X, Y, Z, U and the four basic
% operators, with the value of Value
fourleafed_value(X, Y, Z, U, Value, Expr) :-
        fourleafed(X, Y, Z, U, Tree),
        tree_term(Tree, Expr),
        Expr =:= Value.

| ?- fourleafed_value(1,3,4,6,24,Expr).
Expr = 6 * (1 * 3 * 4)  ? ; no
```

- Comments
  - Variables can be substituted in arithmetical procedures not only to numbers, but to ground arithmetical terms as well.

- Instead of the last call of `fourleafed_value` procedure `Value is Expr` **would not** be good! Why?