Declarative programming

Péter Hanák
hanak@inf.bme.hu

Department of Control Engineering and Information Technology

Péter Szeredi, Dávid Hanák, András György Békés
{szeredi,dhanak,bekesa}@cs.bme.hu

Department of Computer Science and Information Theory

REQUIREMENTS, INFORMATION

Declarative Programming: Information

Homepage, Mailing-list

- Homepage: <http://dp.iit.bme.hu>
- Mailing-list: <http://www.iit.bme.hu/mailman/listinfo/dp-l>.
  Mails to the list members have to be sent to <dp-l@www.iit.bme.hu>.
  Only list members' mail arrives to others without moderator approval.

Lecture Notes

- Szeredi, Péter and Benkő, Tamás: Declarative Programming. Introduction to logic programming (in Hungarian)
- Hanák, D. Péter: Declarative Programming. Introduction to functional programming (in Hungarian)
- Electronic version is available on the homepage (ps, pdf)

Declarative Programming: Information (cont.)

Compiler and Interpreter

- SICStus Prolog — version 3.12 (license may be requested through the ETS)
- Moscow SML (2.0, freeware)
- Both of them are installed on kempelen.inf.bme.hu.
- Both of them can be downloaded from the homepage (linux, Win95/98/NT)
- Exercising/tutoring through ETS on the Web (see homepage)
- System manuals in HTML and PDF format
- Other programs: swiProlog, gnuProlog, poly/ML, smlnj
- emacs-wordprocessor has SML and Prolog mode (linux, Win95/98/NT)

Declarative Programming: Requirements during the Semester

Big HomeWork (BHW)

- In both programming language (Prolog, SML)
- Work independently!
- Programs should be efficient (time limit!), well documented (with comments)
- Developer documentation: 5–10 pages, for both programming languages (TXT, TEX/LATEX, HTML, PDF, PS; BUT NOT DOC or RTF)
- Announced in the 6th week, on the homepage, with downloadable framework
- Deadline in the 12th week; submission in electronic format (see homepage)
- The test-cases handed out and the test cases used at scoring are not the same, but of similar difficulty
- The programs which perfectly solve all the test cases, participate in a *ladder competition* (winners get additional points)

Declarative Programming: Requirements during the Semester (cont.)

Big HomeWork (cont.)

- optional, but *very much* recommended!
- Can also be handed in if solved only in one programming language
- Until the deadline homeworks can be handed in several times, only the last one is scored
- Scoring (for both languages):
  - Each of the 10 test cases, which run correctly and within the time limit earns 0.5 points/test case, max 5. points in total, if at least 4 cases are correct
  - for the documentation, the readability of the code and comments max. 2,5 points
  - That means max. 7,5 total points/language
- The weight of the BHW in the final mark: 15% (15 points from 100 points)

Declarative Programming: Requirements during the Semester (cont.)

Small HomeWork (SHW)

- 2-3 exercises from both Prolog and from SML
- Handing in: electronically (see homepage)
- Optional, but *very much* recommended
- Every good solution earns 1 additional point

Using the Web Exercising system

- Optional, but *indispensable* for the successful midterm-test and exam!
- Embedded in the ETS system (see homepage)

Declarative Programming: Requirements during the Semester (cont.)

Midterm-test, Supplementary Midterm-test (MTT, SMTT, SSMTT)

- The midterm-test is mandatory, closed book test!
- Rule of 40% (for the pass, minimum 40%/language has to be obtained).
  Exception: those students who have already obtained a signature.
- The MTT is in the 7th-10th week, the SMTT is in the last week of the semester
- A single opportunity for SSMTT (in reasonable case) will be given in the first three weeks of the exam-period
- The material covered by the MTT is the first two blocks (1th-7th week)
- The material covered by the SMTT and. the SSMTT is the same as that of the MTT
- The test weights 15% (15 points from 100 points) in the final mark
- If more tests are written the *highest* score is valid

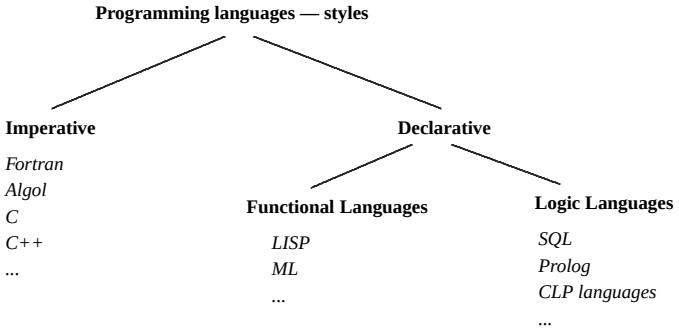
Declarative Programming: Exam

Exam

- Those students can sign in to the exam, who have already got a signature in the given semester, or up to 4 semesters before
- The exam is oral, with preparation in writing
- Prolog, SML: Several smaller tests (program-coding, -analyzing) for 2x35 points
- The final points obtained are the sum of the following: the max. 70 points got in the exam, plus the points got in the **present** semester: for MTT: max. 15 points, for BHW: max. 15 points, plus the additional points (SHW, ladder-competition)
- We do *not* accept points from *earlier* semesters!
- The exam is closed-book exam, but it is possible to ask for some help
- We check the “authenticity” of the BHW and MTT
- Rule of 40% (for the pass minimum 40%/language have to be obtained)
- Earlier exam questions are available on homepage

DECLARATIVE AND IMPERATIVE PROGRAMMING

Classifi cation of Programming Languages



Imperative and Declarative Programming Languages

- Imperative Program
  - Imperative style, using commands
  - Variables: the value of a variable can be modified
  - example in C:

```
int pow(int a, int n) {  // pow(a,n) = a ^ n
  int p = 1;             //  Let p be  1!
  while (n > 0) {        // Repeat until n>0 :
    n = n-1;             //  Decrease n by 1!
    p = p*a;    }        // Multiply p by a!
  return p;           }  // Return the value of p
```

- Declarative Program
  - Declarative style, equations and statements
  - Variable: has a single value, unknown at program writing time
  - SML example :

```
fun pow(a, n) =
    if n > 0            (* If n > 0 *)
      then a*pow(a,n-1)  (* then a^n = a*a^(n-1) *)
      else 1             (* else a^n = 1 *)
```

Declarative Programming in Imperative Language

- It is possible to program in C in a declarative way
  - If we do not use: assignments, loops, jumps, etc.,
  - One can use: (recursive) functions, if-then-else
- The `powd` is a declarative version of the `pow` function:

```
/* powd(a,n) = a^n */ int powd(int a, int n) {
  if (n > 0)                  /* If n > 0 */
    return a*powd(a,n-1);     /* then a^n = a*a^(n-1) */
  else
    return 1;                 /* else a^n = 1 */
}
```

- The (above type of) recursion is expensive, requires non-constant memory `:-(`.

Effi cient Declarative Programming

- The recursion can be efficiently implemented under certain conditions
  - Example: Decide, if an a natural number is a power of a number `b`:

```
/* ispow(a,b) = 1 <=> exits i, such that b^i = a. Precondition: a,b > 0 */
int ispow(int a, int b) {
                                                  /* again: */
  if (a == 1)         return 1;
  else if (a%b == 0)  return ispow(a/b, b);    /* a = a/b; goto again; */
  else                return 0;
}
```

  - Here the recursive call can be implemented as the assignment and jump shown in the comment!
  - This can be done, because after the return from the recursive call, we *immediately* exit the function call.
- This kind of function invocation is called **tail recursion**, **right recursion** or **terminal recursion**
- The Gnu C compiler with a sufficient optimization level (`gcc -O2`) generates the same code from the recursive definition as from the non-recursive one!

Tail Recursive Functions

- Is it possible to write a tail recursive code for the exponentiation (`pow(a,n)`) task?
  - The problem is that a tail recursive function cannot modify the result of the returning recursive call, in other words, the final result has to be available inside the last call.
  - The solution: define an auxiliary function, which has an additional argument, a so called *accumulator*.
- Tail recursive implementation of `pow(a,n)`:

```
/* Auxiliary function: powa(a, n, p) = p*a^n */
int powa(int a,int n, int p) {
  if (n > 0)
    return powa(a, n-1, p*a);
  else
    return p;
}

int powr(int a, int n){
  return powa(a, n, 1);
}
```


Cékla: A Declarative part of the C programming language

- Limitations:
  - Types: only `int`
  - Commands: `if-then-else`, `return`, block
  - Condition part: `(` ⟨exp⟩ ⟨compare-op⟩ ⟨exp⟩ `)`
    - ⟨compare-op⟩: `< | > | == | \= | >= | <=`
  - Expressions: built from variables and integers using binary operators and function calls
    - ⟨arithmetical-op⟩: `+ | - | * | / | % |`
- The Cékla compiler is available on the homepage

The Syntax of the Cékla Language

- the syntax uses the so called DCG (Definite Clause Grammar) notation:
  - terminal symbol: `[terminal]`
  - non-terminal symbol: `non_terminal`
  - repetition (0, 1, or more repetitions, not in DCG): `(to be repeated)...`
- The syntax of program

```
program -->               function_definition... .
function_definition -->   head, block.
head -->                  type, identifier, ['('], formal_args, [')'].
type -->                  [int].
formal_args -->           formal_arg, ([","], formal_arg)... ; [].
formal_arg -->            type, identifier.
block -->                 ['{'], declaration..., statement..., ['}'].
declaration -->           type, declaration_elem, declaration_elem..., [';'].
declaration_elem -->      identifier, ['='], expression.
```


Syntax of Cékla, Continued

- Syntax of Commands

```
statement -->             [if], test, statement, optional_else_part
                        ; block
                        ; [return], expression, [';']
                        ; [';'].
optional_else_part -->    [else], statement ; [].
test -->                  ['('], expression, comparison_op, expression, [')'].
```

- Syntax of Expressions

```
expression -->            term, (additive_op, term)... .
term -->                  factor, (multiplicative_op, factor)... .
factor -->                identifier
                        ; identifier, ['('], actual_args, [')']
                        ; constant
                        ; ['('], expression, [')'].
constant -->              integer.
actual_args -->           expression, ([','], expression)... ; [].
comparison_op -->         ['<'] ; ['>'] ; ['=='] ; ['\='] ; ['>='] ; ['<='].
additive_op -->           ['+'] ; ['-'].
multiplicative_op -->     ['*'] ; ['/'] ; ['%'].
```


1st Small Homework

- Considering an integer as a sequence of digits, its *reverse* can be defined as the integer consisting of the same sequence of digits in reverse order.
  - If the original ineger has n digits, then its reverse has n digits, too.
  - For example: the reverse of 86345 is 54368.
- A program is to be written in the Cékla language which calculates the reverse of a given integer.
  - The main function should be: `reverse(a) = b`, meaning: `b` is the reverse of `a`.
  - Examples:
    - `reverse(534) = 435`
    - `reverse(9026) = 6209`
    - `reverse(86345) = 54368`

A Somewhat More Complicated Cékla Program

- The task: Convert a decimal number *num* — which is between 0 and 1023 — to a 10 digit decimal number containing only digits 0 and 1, so that when this sequence of digits is interpreted as a binary number, its value is *num*. Eg. `bin(5) = 101, bin(37) = 100101`.
- Solution in (imperative) C and in Cékla:

```
int bin(int num) {
  int bp = 512;
  int dp = 1000000000;
  int bin = 0;
  while (bp > 0) {
    if (num >= bp) {
      num = num-bp;
      bin = bin+dp;
    }
    bp = bp / 2;
    dp = dp / 10;
  }
  if (num > 0)
    return -1;
  else
    return bin;
}
```

```
int bina(int num,
         int bp,
         int dp,
         int bin) {
       if (bp > 0) {
         if (num >= bp)
           return bina(num-bp,bp/2,dp/10,bin+dp);
         else
           return bina(num,   bp/2,dp/10,bin);
         }
         if (num > 0)
           return -1;
         else
           return bin;
     }
int bind(int num) {
  return bina(num, 512, 1000000000,0); }
```

Declarative Programming Languages —Lessons Learned from Cékla

- What have we lost?
  - the mutable variables (variables whose value can be changed),
  - the assignment, loop, etc. statements
  - in general: a changeable state
- How can we handle state in a declarative way?
  - the state can be stored in the parameters of the (auxiliary) functions,
  - the change of the state (or keeping the state unchanged) has to be explicit!
- What have we won?
  - Stateless Semantics: the meaning of a language element does not depend on a state
    - Referential transparency — eg. if $f(x) = x^2$, then $f(a)$ **substitutable** with a^2.
    - Single assignment — parallel execution made easy.
  - The declarative programs are **decomposable**:
    - The parts of the program can be written, tested and verified **independently**
    - It is easy to make deductions regarding the program eg. proving its correctness.

Declarative Programming Languages —Motto

- WHAT rather than HOW: The program describes the *task to be solved* (WHAT to solve), rather than the *exact steps of solution process* (HOW to solve).
- In practice both aspects have to be taken care of – dual semantics:
  - Declarative semantics — What (what kind of task) does the program solve;
  - Procedural semantics — How does the program solve it.

Declarative Programming —Why do We Teach it?

- New, high-level programming elements
  - recursion
  - pattern matching
  - backtracking
- New way of thinking
  - decomposable programs: parts of a program (relations, functions) have independent meaning
  - verifiable programs: the code and the meaning of a program can be compared.
- New application areas
  - symbolic application
  - tasks requiring deduction
  - high reliability software systems

An Example dialog with a 50-line Prolog program

(Translation from Hungarian.)

```
| ?- dialog.
|: I am a Hungarian lad.
Understood.
|: Who am I?
Hungarian lad
|:Who is Péter?
I do not know.
|: Péter is student.
Understood.
|: Péter is smart student.
Understood.
|: Who is Péter?
student
smart student
|: I am happy.
Understood.
|: You are a Prolog program.
 Understood.
|: Who am I?
Hungarian lad
happy
|: You are clever.
Understood.
|: You are the center of the world.
Understood.
|: Who are You?
a Prolog program
clever
the center of the world
|:Really?
I do not understand.
|: I am fed up with You.
So am I.
```

INTRODUCTION TO LOGIC PROGRAMMING

The Basic Idea of Logic Programming

- Logic Programming (LP):
  - Programming using mathematical logic
    - a logic program is a **set of logic statements**
    - the **execution of a logic program** is a **deductive process**
  - But: the deduction in (first order) logic requires the traversal of a huge search space
    - Let us restrict the language of the logic
    - Select a simple deduction algorithm, which can be followed by humans
  - The most widespread implementation of LP is the **Prolog** language: **Pro**gramming in **log**ic
    - a severely restricted sublanguage of the first order predicate logic, the so called **definite** or **Horn-clause** language
    - Execution mechanism: **pattern matching** directed procedure invocation with **backtracking** search.

The Outline of the LP Part of the Course

- **Block 1:** The basics of Prolog programming langauge (6 lectures)
  - Logic background
  - Syntax
  - Execution mechanism
- **Block 2:** Prolog programming methods (6 lectures)
  - The most important built-in procedures
  - Advanced language and system elements
- Outlook: New directions in logic programming (1 lecture)

Short Historical Overview of Prolog/LP

60s	Early theorem proving programs
1970-72	The theoretical basis of logic programming (R A Kowalski)
1972	The first Prolog interpreter (A Colmerauer)
1975	The second Prolog interpreter (P Szeredi)
1977	The first Prolog compiler (D H D Warren)
1977–79	Several experimental Prolog applications in Hungary
1981	The Japanese 5th Generation Project chooses logic programming
1982	The Hungarian MProlog is one of the first commercial Prolog implementations
1983	A new compiler model and abstract Prolog machine (WAM) appears (D H D Warren)
1986	The beginning of the Prolog standardization
1987–89	New logic programming languages (CLP, Gödel etc.)
1990–...	Prolog appears on parallel computers Highly-efficient Prolog compilers .....

Information about Logic Programming

- Prolog implementations:
  - SWI Prolog: `http://www.swi-prolog.org/`
  - SICStus Prolog: `http://www.sics.se/sicstus`
  - GNU Prolog: `http://pauillac.inria.fr/~diaz/gnu-prolog/`
- Network information sources:
  - The WWW Virtual Library: Logic Programming: `http://www.afm.sbu.ac.uk/logic-prog`
  - CMU Prolog Repository: (within `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/`)
    - Main page: `0.html`
    - Prolog FAQ: `faq/prolog.faq`
    - Prolog Resource Guide: `faq/prg_1.faq`, `faq/prg_2.faq`

EMPTY

This page is intentionally left blank.

English Textbooks on Prolog

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)
  Downloadable as a pdf file from `http://www.ida.liu.se/~ulfni/lpp`
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

Our first Prolog program: checking if a number is a power of another

- A simple example in Cékla and Prolog:

```
/* ispow(a,b) = 1 <=> exits i, such that b^i = a. Precondition: a,b > 0 */
```

```
int ispow(int num, int base) {
  if (num == 1)
    return 1;
  else if (num%base == 0)
    return ispow(num/base, base);
  else
    return 0;
}
```

```
ispow(Num, Base) :-
    (   Num =:= 1
    ->  true
    ;   Num rem Base =:= 0,
        Num1 is Num//Base,
        ispow(Num1, Base)
    ).
```

- `ispow` is a Prolog **predicate**, that is a procedure (function) returning a Boolean value.
- The procedure consists of a single clause, of form *Head* `:-` *Body*.
- The head contains the parameters `Num` and `Base` which are variables (**written in capitals**!)
- The body consists of a single goal which is a **conditional structure**:
  `if Cond then ThenCode else ElseCode` ≡ `( Cond -> ThenCode ; ElseCode )`
- The "`true`", "`A =:= B`" and "`A is B`" structures are calls of built-in predicates.

Some Built-In Predicates

- Unification: X = Y: The `X` and `Y` **symbolic** expressions can be brought to the same form, by instantiating variables (and carries out these instantiations).
- Arithmetic predicates
  - `X is Exp`: The **arithmetic** expression `Exp` is evaluated and its **value** is unified with `X`.
  - `Exp1<Exp2, Exp1=<Exp2, Exp1>Exp2, Exp1>=Exp2, Exp1=:=Exp2, Exp1=\=Exp2`: The values of arithmetic expressions `Exp1` and `Exp2` are in the given relation with each other (`=:=` means arithmetic equality, `=\=` means arithmetic inequality).
  - If any of `Exp, Exp1 or Exp2` is not a **ground** (variable-free) arithmetic expressions ⇒error.
  - the most important arithmetic operators `+, -, *, /, rem, //` (integer-div)
- Output predicates
  - `write(X)`: The Prolog expression `X` is written out (displayed on the screen).
  - `nl`: A new line is written out.
- Other predicates
  - `true, fail`: Always succeeds vs. always fails.
  - `trace, notrace`: Turns (exhaustive) tracing on/off.

Built-In Predicates for Program Development

- `consult(File)` or `[File]`: Reads the program from the `File` and stores it in interpreted format. (if `File = user` ⇒ read from the terminal)
- `listing` or `listing(Predicate)`: Lists all interpreted predicates, or all interpreted predicates with the given name.
- `compile(File)`: Reads the program from the `File` and compiles it.
- The compiled format is faster, but cannot be listed, and tracing is **slightly less** accurate.
- `halt`: Exit the Prolog system.

```
> sicstus
SICStus 3.11.0 (x86-linux-glibc2.3): Mon Oct 20 15:59:37 CEST 2003
| ?- consult(ispow).
% consulted /home/user/ispow.pl in module user, 0 msec 376 bytes
yes
| ?- ispow(8, 3).
no
| ?- ispow(8, 2).
yes
| ?- listing(ispow).
(...)
yes
| ?- halt.
>
```


Writing General (non Boole-valued) Functions in Prolog

- Example: Calculating the power of a natural number in Cékla and Prolog:

```
/* powd(a,n) = a^n */
int powd(int a, int n) {
  if (n > 0)
    return a*powd(a,n-1);


  else
    return 1;
}
```

```
/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
        (   N > 0
        ->  N1 is N-1,
            powd(A, N1, P1),
            P is A*P1
        ;   P = 1
        ).

| ?- powd(2, 8, P).
P = 256 ?
```

- The predicate `powd` with 3 arguments corresponds to the`powd` function with 2 arguments.
- The two arguments of the function correspond to the first two arguments of the predicate, which are **input** i.e. instantiated arguments.
- The result of the function is the last, **output** argument of the predicate, which is usually an uninstantiated variable.

Predicates with Multiple Clauses

- The conditional structure is not a basic element of the Prolog language (it was not there in the first Prologs)
- Instead a conditional a predicate with two, **mutually exclusive** clauses can be used:

```
/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
        (   N > 0
        ->  N1 is N-1,
            powd(A, N1, P1),
            P is A*P1
        ;   P = 1
        ).
```

```
powd2(A, N, P) :-
        N > 0,
        N1 is N-1,
        powd2(A, N1, P1),
        P is A*P1.
powd2(A, N, 1) :-
        N =< 0.
```

- If a predicate has multiple clauses, Prolog tries **all of them** :
  - If the 2nd parameter of `powd2` (`N`) is positive, then the first clause is used,
  - otherwise (i.e. if `N =< 0`) the second one.
- If the second clause of `powd2` is: `powd(A,0,1)`, then a call with a negative exponent fails.
- In general the clauses need not be exclusive: a single question can lead to multiple answers:

```
equation_root(A, B, C, X) :- X is (-B + sqrt(B*B-4*A*C))/(2*A).
equation_root(A, B, C, X) :- X is (-B - sqrt(B*B-4*A*C))/(2*A).
```

Predicates with Multiple Answers —Family Relationships

- Data

  A child–parent relation, eg. family relations in the family of King Stephen I, the first king of Hungary:

child	parent
Imre	István
Imre	Gizella
István	Géza
István	Sarolta
Gizella	Civakodó Henrik
Gizella	Burgundi Gizella

- The task:

  We have to define the grandchild–grandparent relation, i.e. write a program which finds the grandparents of a given person.

The Grandparent Problem —Prolog Solution

```
% parent(C, P):P is a parent of C.
parent('Imre', 'István').
parent('Imre', 'Gizella').
parent('István', 'Géza').
parent('István', 'Sarolt').
parent('Gizella', 'Civakodó Henrik').
parent('Gizella', 'Burgundi Gizella').

% Grandparent is a grandparent of  Child.
grandparent(Child, Grandparent) :-
    parent(Child, Parent),
    parent(Parent, Grandparent).
```

```
% Who are Imre's grandparents?
| ?- grandparent('Imre', GP).
GP = 'Géza' ? ;
GP = 'Sarolt' ? ;
GP = 'Civakodó Henrik' ? ;
GP = 'Burgundi Gizella' ? ; no
% Who are Géza's grandchildren?
| ?- grandparent(GC, 'Géza').
GC = 'Imre' ? ; no
```


Data Structures in Declarative Languages —Example

- The binary tree data structure is
  - either a node (`node`) which contains two subtrees (`left,right`)
  - or a leaf (`leaf`) which contains an integer
- Let us define binary tree structures in different languages:

```
% Declaration of a structure in C
enum treetype Node, Leaf; struct tree {
  enum treetype type;
  union {
    struct { struct tree *left;
             struct tree *right;
           } node;
    struct { int value;
           } leaf;
  } u;
};
```

```
% Data type declaration in SML
 datatype Tree =
         Node of Tree * Tree
       | Leaf of int

% Data type description in Prolog

 :- type tree --->
        node(tree, tree)
      | leaf(int).
```


Calculating the Sum of a Binary Tree

- To calculate the sum of the leaves of a binary tree:
  - if the tree is a node, add the sums of the two subtrees
  - if the tree is a leaf, return the integer in the leaf

```
% C function (declarative)
int sum_tree(struct tree *tree) {
  switch(tree->type) {
  case Leaf:
   return tree->u.leaf.value;
  case Node:
   return
    sum_tree(tree->u.node.left) +
    sum_tree(tree->u.node.right);
   }
}
```

```
% Prolog procedure (predicate)
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1),
        sum_tree(Right, S2),
        S is S1+S2.
```

Sum of Binary Trees

- Prolog sample run:

```
% sicstus -f
SICStus 3.10.0 (x86-linux-glibc2.1): Tue Dec 17 15:12:52 CET 2002
Licensed to BUTE DP course
| ?- consult(tree).
% consulting /home/szeredi/peldak/tree.pl...
% consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes
yes
| ?- sum_tree(node(leaf(5),
                   node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- sum_tree(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal:  10 is _73+_74
| ?- halt.
%
```

- The cause of the error: the built-in arithmetic is one-way: the `10 is S1+S2` call causes an error!

Peano Arithmetic —Addition

- We can define the addition for natural numbers using Peano axioms if the numbers are built by repeated application of the `s(X)` „successor” function:
  `1 = s(0), 2 = s(s(0)), 3 = s(s(s(0))), ...` (Peano representation).

```
% plus(X, Y, Z): The sum of X and Y is Z (X,Y,Z are in Peano representation).
plus(0, X, X).                  % 0+X = X.
plus(s(X), Y, s(Z)) :-
        plus(X, Y, Z).          % s(X)+Y = s(X+Y).
```

- The `plus` predicate can be used in multiple directions:

```
| ?- plus(s(0), s(s(0)), Z).       Z = s(s(s(0))) ? ; no    %  1+2 = 3

| ?- plus(s(0), Y, s(s(s(0)))).    Y = s(s(0)) ? ; no       %  3-1 = 2

| ?- plus(X, Y, s(s(0))).          X = 0, Y = s(s(0)) ? ;   %  2 = 0+2
                                   X = s(0), Y = s(0) ? ;   %  2 = 1+1
                                   X = s(s(0)), Y = 0 ? ;   %  2 = 2+0
                                   no
| ?-
```


Building Trees with a Given Sum

- Building a tree with a given sum, using Peano arithmetic:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left, Right), S) :-
        plus(S1, S2, S),
        S1 \= 0, S2 \= 0,     % X \= Y built-in procedure, meaning
                              % X and Y cannot be unified
                              % 0 excluded, to avoid ∞ many solutions.
        sum_tree(Left, S1),
        sum_tree(Right, S2).
```

- the running of the procedure:

```
| ?- sum_tree(Tree, s(s(s(0)))).
Tree = leaf(s(s(s(0)))) ? ;                              % 3
Tree = node(leaf(s(0)),leaf(s(s(0)))) ? ;                % (1+2)
Tree = node(leaf(s(0)),node(leaf(s(0)),leaf(s(0)))) ? ;  % (1+(1+1))
Tree = node(leaf(s(s(0))),leaf(s(0))) ? ;                % (2+1)
Tree = node(node(leaf(s(0)),leaf(s(0))),leaf(s(0))) ? ;  % ((1+1)+1)
no
```


The Data Structure of Prolog, the Notion of Term

- constant (*atomic*)
  - number: numeric constant (*number*) — integer or float, eg. `1, -2.3, 3.0e10`
  - name: symbolic constant (*atom*), eg. `'István', ispow, +, -, <, sum_tree`
- compound or structure (*compound*)
  - so called canonical form: ⟨ name of structure ⟩(⟨ arg_1 ⟩, ...)
    - the ⟨ name of structure ⟩ is an atom, the ⟨ arg_i ⟩ arguments are arbitrary Prolog terms
    - examples: `leaf(1), person(william,smith,2003,1,22), <(X,Y), is(X, +(Y,1))`
  - syntactic sugar, ie. operators: `X is Y+1` ≡ `is(X, +(Y,1))`
- Variable (*var*)
  - eg. `X, Parent, X2, _var, _, _123`
  - The variable is initially uninstantiated, ie. it has no value, it can be instantiated to an arbitrary Prolog term (including another variable), in the process of unification (pattern matching)

THE SYNTAX OF PROLOG —FIRST APPROXIMATION

Predicates, clauses

- Example:

```
% A definition of the predicate with two clauses, the functor is: sum_tree/2
sum_tree(leaf(Val), Val).            %                    1. clause, fact
sum_tree(node(Left,Right), S) :-     %             head   \
        sum_tree(Left, S1),          % goal       \       |
        sum_tree(Right, S2),         % goal       | body  | 2. clause, rule
        S is S1+S2.                  % goal       /       /
```

- Syntax:

⟨ Prolog progran ⟩	::=	⟨ predicate ⟩ . . .	
⟨ predicate ⟩	::=	⟨ clause ⟩ . . .	{with the same functor}
⟨ clause ⟩	::=	⟨ fact ⟩.␣ |	
		⟨ rule ⟩.␣	{functor of the clause = functor of the head}
⟨ fact ⟩	::=	⟨ head ⟩	
⟨ rule ⟩	::=	⟨ head ⟩ :- ⟨ body ⟩	
⟨ body ⟩	::=	⟨ goal ⟩, . . .	
⟨ goal ⟩	::=	⟨ term ⟩	
⟨ head ⟩	::=	⟨ term ⟩	

Recommended formatting of Prolog programs

- The recommended formatting of Prolog programs:
  - Place clauses of a predicate one after the other, do not put empty lines between them. Separate predicates by empty lines and possibly comments.
  - Write the head of the clause at the beginning of a line, and prefix each goal in the body with an indentation of a few (8 recommended) spaces. Preferably write the head and each goal on separate lines.

Prolog terms

- Example — a clause head as a term:

```
%     sum_tree(node(Left,Right), S)    % compound term, functor is sum_tree/2
%     ________ ________________  _
%        |         |             |
% structure name    |           argument, variable
%                  \- argument, compound term
```

- Syntax:

⟨ term ⟩	::=	⟨ variable ⟩ |	{no functor}
		⟨ constant ⟩ |	{Functor: ⟨ constant ⟩`/0`}
		⟨ compound term ⟩ |	{Functor: ⟨ structure name ⟩`/`⟨ arity ⟩}
		`(` ⟨ term ⟩ `)`	{Because of operators, see later}
⟨ constant ⟩	::=	⟨ name constant ⟩ |	{also called ⟨ atom ⟩}
		⟨ number constant ⟩	
⟨ number constant ⟩	::=	⟨ integer ⟩ |	
		⟨ float number ⟩	
⟨ compound term ⟩	::=	⟨ structure name ⟩ (⟨ argument ⟩, . . .)	
⟨ structure name ⟩	::=	⟨ name constant ⟩	
⟨ argument ⟩	::=	⟨ term ⟩	

Lexical elements

- Examples:

```
% variable:               Fact FACT _fact X2 _2 _
% name constant (atom):   fact ≡ 'fact' 'István' [] ; ',' += ** \= ≡ '\\='
% number constant:        0 -123 10.0 -12.1e8
% not a name constant:    !=, Istvan
% not a number constant:  1e8 1.e2
```

- Syntax:

⟨variable⟩ ::= ⟨capital letter⟩⟨alphanumeric char⟩...|
_⟨alphanumeric char⟩...

⟨name constant⟩ ::= '⟨quoted character⟩...' |
⟨lower case letter⟩⟨alphanumeric char⟩...|
⟨sticky char⟩...| ! | ; | [] | {}

⟨integer⟩ ::= {signed or unsigned digit sequence }

⟨float number⟩ ::= { a sequence of digits with a compulsory decimal point in between, with an optional exponent}

⟨quoted character⟩ ::= {any non ' and non \ character} | \ ⟨escape sequence⟩

⟨alphanumeric char⟩ ::= ⟨lower case letter⟩ | ⟨upper case letter⟩ | ⟨digit⟩ | _

⟨sticky char⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Syntactic sugar: operators

- Example:

```
% S is -S1+S2 is equivalent to the term: is(S, +(-(S1),S2))
```

- Terms with operators

⟨compound term⟩ ::=
⟨structure name⟩ (⟨argument⟩, ...) {until now we had only this}
| ⟨argument⟩ ⟨operator name⟩ ⟨argument⟩ ⟨infix term⟩
| ⟨operator name⟩ ⟨argument⟩ ⟨prefix term⟩
| ⟨argument⟩ ⟨operator name⟩ ⟨postfix term⟩

⟨operator name⟩ ::= ⟨structure name⟩ {if declared as an operator}

- Built-in predicates handling operators:
  - `op(Priority, Type, OpName)` or `op(Priority, Type, [OpName1,OpName2,...])`:
    - `Priority`: integer between 0–1200
    - `Type`: `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` - one of these name constants
    - `OpName`: any symbolic constant
    - If priority is positive the operator(s) are defined, if it is 0 they are deleted.
  - `current_op(Priority, Type, OpName)`: lists the current operators.

Standard built-in operators

Standard operators

```
1200 xfx :- -->
1200  fx :- ?-
1100 xfy ;
1050 xfy ->
1000 xfy ','
 900  fy \+
 700 xfx < = \= =..
          =:= =< == \==
          =\= > >= is
          @< @=< @> @>=
 500 yfx + - /\ \/
 400 yfx * / // rem
          mod << >>
 200 xfx **
 200 xfy ^
 200  fy - \
```

Other built-in operators of SICStus Prolog

```
1150  fx dynamic multifile
         block meta_predicate
 900  fy spy nospy
 550 xfy :
 500 yfx #
 500  fx +
```


Characteristics of operators

- An operator is characterized by its type and priority.
- The type determines the operator-class (the way the operator is placed) and the associativity:

Type			Class	Interpretation
left-assoc.	right-assoc.	non-assoc.		
`yfx`	`xfy`	`xfx`	infix	`X f Y` ≡ `f(X, Y)`
	`fy`	`fx`	prefix	`f X` ≡ `f(X)`
`yf`		`xf`	postfix	`X f` ≡ `f(X)`

- If multiple operators are present the parenthesizing depends on the priority and associativity:
  - `a/b+c*d` ≡ `(a/b)+(c*d)` because the priority of `/` and `*` is 400, which is **smaller** than the priority of `+` (500) (smaller priority = **stronger** binding).
  - `a+b+c` ≡ `(a+b)+c` as the `+` operator's type is `yfx`, thus it is left-associative (letter `y` is on the left side of `yfx`) — binds to the left, parentheses are from the left to the right
  - `a^b^c` ≡ `a^(b^c)` as `^` operator's type is `xfy`, therefore it is right-associative (binds to the right, parentheses are from the right to the left)
  - `a=b=c` syntactically bad, as the `=` operator's type is `xfx`, thus it is non-associative.

Operators: use of parentheses

- Let us set off from a fully parenthesized term containing multiple operators.
- The priority of a subterm is the priority of its (outermost) operator.
- If a term with priority ap appears as an argument to an operator with priority op then the parentheses around the argument can be omitted if:
  - $ap < op$, for example `a+(b*c)` $\equiv$ `a+b*c` ($ap = 400, op = 500$)
  - $ap = op$, and the term is the right argument of a right-associative operator, for example `a^(b^c)` $\equiv$ `a^b^c` ($ap = 200, op = 200$)
  - $ap = op$, the left argument of a left-associative operator, for example `(1+2)+3` $\equiv$ `1+2+3`. Exception: if the operator of the left argument is right-associative, thus the previous condition can be applied.
- An example for the exception:
  - ```
    :- op(500, xfy, +^).
    | ?- :- write((1 +^ 2) + 3), nl.  ⇒ (1+^2)+3
    | ?- :- write(1 +^ (2 + 3)), nl.  ⇒ 1+^2+3
    ```
  - Thus: in case of conflict the associativity of the first operator „wins”.

Operators —additional comments

- It is not allowed to have operators with the same name and in the same class at the same time.
- We can define operators in the text of a program with directives, for example:
  ```
  :- op(500, xfx, --).  :- op(450, fx, @).  sum_tree(@V, V).  sum_tree(L--R, V) :- ...
  ```
- The twofold role of the “comma”
  - separates the arguments of the structure-term
  - works as an operator of priority 1000, type `xfy`, e.g. in clause bodies:
    `(p :- a,b,c) = :-(p,',’(a,',’(b,c)))`
  - the “bare” comma (`,`) is not allowed as a name constant, but as an operator it can be used without the quotes as well.
  - In an argument a term with a priority higher than 999 should be placed inside parentheses:
    ```
    | ?- write_canonical((a,b,c)).  ⇒ ',’(a,',’(b,c))
    | ?- write_canonical(a,b,c).    ⇒ ! procedure write_canonical/3 does not exist
    ```
- For the unambiguous analysis, the Prolog standard stipulates, that
  - an operator as an operand has to be placed in parentheses, for example: `Comp = (>)`
  - an infix and a postfix operator with the same name cannot exist.
- These restrictions are not compulsory in many Prolog systems.

Use of operators

- What are operators good for?
  - convenient writing of arithmetic procedures , like `X is (Y+3) mod 4`
  - symbolic processing of expressions (like symbolic derivation)
  - for writing the clauses themselves (`:-` and `',’` are both operators)
  - clauses can be handed over to meta-predicates, like `asserta( (p(X):-q(X),r(X)) )`
  - to make heads and procedure calls more readable:
    ```
    :- op(800, xfx, [grandparent_of, parent_of]).

    GP grandparent_of GC :-  P parent_of GC, GP parent_of P.
    ```
  - to make data structures more readable, like
    ```
    :- op(100, xfx, [.]).

    acid(sulphur, h.2-s-o.4).
    ```
- Why are operators bad?
  - It is a single global resource, it can cause problems in a larger project.

Arithmetics in Prolog

- Operators make it possible to write arithmetic expressions the usual way, as we do in mathematics or in other programming languages.
- The `is` built-in predicate expects an arithmetic expression on its right side (2. argument), evaluates it, and unifies the result with the argument on the left side.
- The `=:=` built-in predicate expects an arithmetic expression on both sides, evaluates them, and fails if the values are not equal.
- Examples:
  ```
  | ?- X = 1+2, write(X), write('  '), write_canonical(X), Y is X.
  ⇒           1+2                    +(1,2)    ⟹ X = 1+2, Y = 3 ? ; no
  | ?- X = 4, Y is X/2, Y =:= 2.    ⟹ X = 4, Y = 2.0 ? ; no
  | ?- X = 4, Y is X/2, Y = 2.      ⟹ no
  ```
- **Important**: the terms composed of arithmetical operators (+,-,...) are **compound Prolog terms**. Only the built-in arithmetic predicates evaluate these!
- The Prolog terms are basically symbolic, the arithmetic evaluation is the “exception”.
  ```

Classical symbolic expression processing: derivation

- Let's write a Prolog predicate which calculates the derivative of a term made up of numbers and the x name constant.

```
% deriv(Expr, D):  D is the derivative of Expr with respect to x.
deriv(x, 1).
deriv(C, 0) :-                      number(C).
deriv(U+V, DU+DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-          deriv(U, DU), deriv(V, DV).

| ?- deriv(x*x+x, D).
  ⟹  D = 1*x+x*1+1 ? ; no

| ?- deriv((x+1)*(x+1), D).
  ⟹  D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no

| ?- deriv(I, 1*x+x*1+1).
  ⟹  I = x*x+x ? ; no

| ?- deriv(I, 0).
  ⟹  no
```


THE FOUNDATIONS OF PROLOG IN LOGIC

Example with operators: substitution value of a polynomial

- Polynomial: a Prolog term built from numbers and the 'x' name constant, using the '+' and '*' operators.
- The task: calculate the value of a polynomial for a given x value.

```
% value_of(Expr, X, E): E is the value of the polynomial Expr,
% with the substitution x=X
value_of(x, X, E) :-
        E = X.
value_of(Expr, _, E) :-
        number(Expr), E = Expr.
value_of(K1+K2, X, E) :-
        value_of(K1, X, E1),
        value_of(K2, X, E2),
        E is E1+E2.
value_of(K1*K2, X, E) :-
        value_of(K1, X, E1),
        value_of(K2, X, E2),
        E is E1*E2.

| ?- value_of((x+1)*x+x+2*(x+x+3), 2, E).
E = 22 ?
```


Prolog equivalents of basic concepts in logic

- The elements of the logic language:
  - Expression (*term*): built up from variables and constants combined with functions eg. $f(a, g(X))$, where f and g are names of functions with 2 resp. 1 arguments, a is a constant name (e.g., a 0-argument function) and X is variable name.
  - Predicate: a relational symbol completed with the appropriate number of arguments where the arguments are expressions eg.: $divisor(X, X * Y)$.
  - Statement (*formula*): Predicates combined with logical operators (e.g., $\wedge$, $\vee$, $\neg$, $\rightarrow$) and quantors ($\forall$, $\exists$), eg. $\forall X(X < 0 \rightarrow \neg X < X * 2)$.
  - Prolog conventions:
    - The variable names begin with a capital letter or an underscore.
    - Functions and predicates wth two arguments can be written in infix form eg. $X + 2 * Y \equiv +(X, *(2, Y))$, $X < X * 2 \equiv <(X, *(X, 2))$
    - Function (and constant) names begin with a small letter or are written between single quotes. Symbols or symbol sequences are allowed as function, constant or statement names (eg. $+, *, <$).

Restricting the language of logic

- To make the deduction process more efficient, it is worth it to restrict the language of logic.
- We introduce the concept of *clauses*. A clause is logic statement of the following form:

$$\forall X_1 \ldots X_j((F_1 \vee \ldots \vee F_n) \leftarrow (T_1 \wedge \ldots \wedge T_m))$$

  - The left side of the implication (corollary) is the **head** of the clause
  - The right side (condition) is the **body**, members of the conjunction in the body are **goals**.
  - F_i and T_j are predicates $n, m \geq 0 \rightarrow$ both head and body can be empty.
  - $X_1 \ldots X_j$: all variables of the clause.
- An exact equivalent of the above formula (cf. $A \leftarrow B \equiv A \vee \neg B$):

$$\forall X_1, \ldots X_j(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$$

- Simplified form of clauses: $F_1, \ldots, F_n$:–$T_1, \ldots, T_m$. If $m = 0$, then `:-` is eliminated.
- Examples — warning, these are generic clauses, not all of them are allowed in Prolog!

```
male(X), female(X) :- human(X).     % A human is male or female.
:- male(X), female(X).              ≡ ∀ X ¬ (male(X) ∧ female(X))
                                    % Nothing is both a male and a female.
love(X, X) :- saint(X).             % All saints love themselves.
saint('István').                    % István is a saint.
```


THE SEMANTICS AND EXECUTION OF PROLOG PROGRAMS

Declarative Semantics —the Logic Form of Clauses

- The notion of general clause as introduced in mathematical logic:
  $F_1, \ldots, F_n$:–$T_1, \ldots, T_m$. $\forall \overline{X}(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$
- Definite clause or Horn clause: a clause whose head contains at most one element ($n \leq 1$).
- Classification of Horn clauses
  - If $n = 1, m > 0$, then the clause is called a **rule**, eg.
    `grandparent(GC,GP) :- parent(GC,P), parent(P,GP).`
    logic form: $\forall GC\ GP\ P$ (`grandparent`$(GC, GP) \leftarrow$ `parent`$(GC, P) \wedge$ `parent`(P, GP))
    equivalent form: $\forall GC\ GP$ (`grandparent`$(GC, GP) \leftarrow \exists P$(`parent`$(GC, P) \wedge$ `parent`(P, GP)))
  - in case of $n = 1, m = 0$ the clause is a **fact**, eg.
    `parent('Imre', 'István').`
    its logic form is exactly the same.
  - In case of $n = 0, m > 0$ the clause is a **query**, eg.
    `:- grandparent('Imre', X).`
    logic form: $\forall X \neg$`grandparent`$(\text{'Imre'}, X)$, equivalently $\neg \exists X$`grandparent`$(\text{'Imre'}, X)$
  - If $n = 0, m = 0$, then it is an **empty clause**, denoted by: $\square$. Logically this is an empty disjunction, which is equivalent to false.

The Role of Functions in Prolog

- The role of function symbols
  - The Prolog is based on so called equality-free logic, so we can not state that two terms (of first order logic) are equal.
  - This is the reason that the function symbols can be *only* be used as constructor-functions:
    $f(x_1, \ldots, x_n) = z \Leftrightarrow (z = f(y_1, \ldots, y_n) \wedge x_1 = y_1) \wedge \ldots \wedge (x_n = y_n))$
  - Example `leaf(X) = Z` $\Leftrightarrow$ `Z = leaf(Y)` $\wedge$ `X = Y`, in other words, `leaf(X)` is a new entity, different from all other entities.
- Example:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1), sum_tree(Right, S2), S is S1+S2.

| ?- sum_tree(node(leaf(1),leaf(2)), Sum).     ⇒    Sum = 3 ?
| ?- sum_tree(Tree, 3).                        ⇒    Tree =leaf(3) ?
```

  - The term `node(leaf(1),leaf(2))` in the query is *unambiguously decomposed* by the procedure.
  - Pattern matching (unification) is bidirectional: it is able to both compose and decompose.

Declarative Semantics of Prolog

- Declarative Semantics
  - An auxiliary concept: an **instance** of a term/statement is a term/statement obtained by substituting some/all variables in it.
  - The execution of a query is **successful**, if an instance of the body of the query is a logical **consequence** of the program (i.e. the conjunction of the clauses in the program).
  - The result of the execution is the **substitution** which produces the instance.
  - A query may execute successfully in multiple ways.
  - The execution of a query **fails** if none of its instances is a consequence of the program.
- Example:

```
parent('Imre', 'István').                              (p1)
parent('Imre', 'Gizella').                             (p2)
parent('István', 'Géza').                              (p3)
parent('István', 'Sarolt').                            (p4)
parent('Gizella', 'Civakodó Henrik').                  (p5)
parent('Gizella', 'Burgundi Gizella').                 (p6)

grandparent(GC, GP) :- parent(GC, P), parent(P, GP).   (gp)

:- grandparent('Imre', GP).                            (goal)
```

  - from `(p1)` + `(p3)` + `(gp)` follows that `grandparent('Imre', 'Géza')`, so `(goal)` executes successfully using the `GP = 'Géza'` substitution.
  - Another example of a successful execution: `(p1)+(p4)+(gp)` → `GP = 'Sarolt'`.

Declarative Semantics

- Why is declarative semantics a good thing?
  - The program is **decomposable**: It is possible to assign a meaning to single predicates (even to single clauses) separately.
  - The program is **verifiable**: in view of the intended meanings of predicates it can be checked if the clauses describe true statements.
  - It is very important to formulate the intended meaning of a predicate in a **head comment**. This is a declarative sentence which describes the relation between the arguments. Examples:
    - Head comments: `% parent(C,P):` `C` has parent `P`.
      `% grandparent(GC,GP):` `GC` has grandparent `GP`.

      `grandparent(GC, GP) :- parent(GC, P), parent(P, GP).`
      The meaning of the clause: If `GC` has parent `P` ans `P` has parent `GP`, then `GC` has grandparent `GP`. This is in accordance with our expectations and it is acceptable as a **true statement**.
    - Head comments: `% sum_tree(T, Sum):` Tree `T` has leaf sum `Sum`.
      `% E is Exp:` Arithm. expr. `Exp` has value `E`. (`is` is infix!)

      `sum_tree(node(L,R), S) :- sum_tree(L, S1), sum_tree(R, S2), S is S1+S2.`
      Meaning: If tree `L` has leaf sum `S1` and tree `R` has leaf sum `S2`, and the arith. expression `S1+S2` has value `S`, then tree `node(L,R)` has leaf sum `S`. This is again a true statement.

Declarative Semantics (contd.)

- Why is declarative semantics insufficient?
  - The declarative semantics is based on general deduction.
  - There are several ways to do deduction, so this process needs search.
  - In case of an infinite search space the deduction enginge may fall into an **infinite loop**.
  - In case of finite search space the search may have very **poor efficiency**.
  - Some **built-in predicates** are able to work only under certain conditions. For instance: `S is S1+S2` signals error, if `S1` or `S2` is unknown. Because of this

    `sum_tree(node(L,R), S) :- S is S1+S2, sum_tree(L, S1), sum_tree(R, S2).`

    is logically correct, but leads to an error.
- As a consequence, it is very important that a Prolog programmer knows thoroughly the execution mechanism of Prolog, in other words, the **procedural semantics** of the language.
- Motto: **Think declaratively and check procedurally!**
  Meaning: after you have written your declarative program, think it over if the procedural execution is correct (does not fall in an infinite loop, is efficient, the built-in predicates are operational, etc.)

Procedural Semantics of Prolog

- The execution mechanism of Prolog can be described in several ways:
  - Theorem proving by SLD resolution (very briefly see below)
  - Theorem proving by goal reduction (see next slides)
  - Pattern matching based application of backtrackable procedures (details see later).
- Theorem proving by SLD used in Prolog:
  - SLD resolution: **L**inear resolution with a **S**election function for **D**efinite clauses.
  - The query **negates** the existence of the object looked for, eg. `'Imre'` has no grandparents: `:- grandparent('Imre', GP).` $\equiv \neg\exists G$ `grandparent(`$'$`Imre`$'$`,`GP`)`
  - The *resolvent* of the query and a program clause results in a new query.
  - Such resolution steps are repeated until an empty clause is reached (backtracking when reaching a dead end).
  - By this we prove **indirectly** that the body of the query is a consequence of the program: the negation of the body and the program is proven to imply "false" ($\square$).
  - This proof is constructive, ie. the variables of the query are instantiated — this is the answer we seek eg. $G =$ `'Géza'`).
  - Further answers can be produced by other proofs.

Prolog as a Goal-Reduction Theorem Prover

- The main idea: The goal to be solved is reduced to subgoals from which it follows.
- Example program:

```
parent('Imre', 'István').                              (p1)
parent('Imre', 'Gizella').                             (p2)
parent('István', 'Géza').   (...)                      (p3)

grandparent(GC, GP) :- parent(GC, P), parent(P, GG).   (gp)
```

- the initial query: `:- grandparent('Imre', GP').`
  (Now a query is considered as a statement to be proven.)
- We extend the query with one or more special goals to preserve the values of the variables:
  `:- grandparent('Imre', GP'), write(GP').`
- The query is **reduced** (see next slide) repeatedly, until only `write` goals remain:

```
[red. with (gp) clause]    :- parent('Imre', P), parent(P, GP'), write(GP').
[red. with (p1) clause]    :- parent('István', GP'), write(GP').
[red. with (p3) clause]    :- write('Géza').
```

- We can read the result of the run from the argument of `write`.

The reduction step

- The clauses used in the example and the query:

```
parent('Imre', 'István').                              (p1)
parent('István', 'Géza').                              (p3)
grandparent(GC, GP) :- parent(GC, P), parent(P, GP).   (gp)
:- grandparent('Imre', GP'), write(GP').
```

- Reduction step: a query + a related clause $\Rightarrow$ new query.
- The reduction step is tried for **all** clauses of the predicate (one by one):
  - The **first** goal of the query is brought to a form identical to that of the clause head by variable substitution.
  - Both the clause and the query are **specialized** using this substitution. For `(gp)` this is:
    `grandparent('Imre', GP') :- parent('Imre', P), parent(P, GP'). (gp*)`
  - The first goal is replaced by the body of the clause, ie. the goal is replaced by its precondition. In the example: `parent('Imre', P), parent(P, GP'), write(GP').`
- Next, we reduce the goal using clause `(p1)`, specializing the **query** by `P = 'István'`:
  `parent('István', GP'), write(GP').`
  Because we reduced the query using a fact (with an empty body), its length decreases.
- Next a similar step can be made using `(p3)`, resulting in the final query: `write('Géza').`

Reduction step —further details

- Handling of variables
  - The scope of a variable is a single clause (cf. $\forall X_1 \ldots X_j (F \leftarrow T)$).
  - Before the reduction step the clause has to be copied, systematically replacing all variables by new ones (cf. recursion).
- **Unification**: Two terms/statements are brought to an identical form, by variable substitution.
  - The variables can be substituted with arbitrary terms, including other variables.
  - The unification produces the **most general** common form eg.

`sum_tree(leaf(X), X)` and `sum_tree(T, V)` — common form: `sum_tree(leaf(X), X)` and not eg. `sum_tree(leaf(0), 0)`

  - The result of the unification is the substitution, which results in the most general common form. This is unique, except for variable renaming. In the example: `T=leaf(X), V=X`.
  - Examples:

Call:	*Head:*	*Substitution:*
`grandparent('Imre', GP')`	`grandparent(C, GP)`	`C ='Imre', GP = GP'`
`parent('Imre', P)`	`parent('Imre', 'István')`	`P='István'`
`parent('Imre', P)`	`parent('István', 'Géza')`	*not unifiable*
`love('István', Who)`	`love(X, X)`	`X = 'István', Who = 'István'`
`love(Who1, Who)`	`love(X, X)`	`X = Who, Who = Who1`

Choice points, Backtracking

- We were “lucky” in the example, the sequence of the reduction steps led to a solution
- In the general case we can reach a dead end (a non-reducible query), eg.

```
:- grandparent('Imre', 'Civakodó Henrik').              (gp)
:- parent('Imre', P), parent(P, 'Civakodó Henrik').     (p1): parent('Imre', 'István')
:- parent('István', 'Civakodó Henrik').                 ???
```

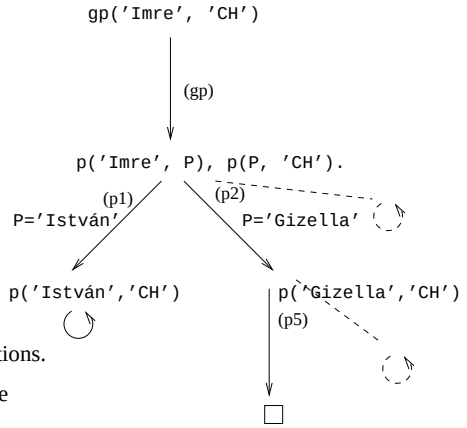
- The 2nd query was reduced with clause `(p1)`, but to get to the solution we need `(p2)`:
  `parent('Imre', 'Gizella')` — Not only the the first clause has to be tried but all clauses!
- If a query is reduced with any but the last of the clauses then a **choice point** is created, in which we store the query and the number of the clause used for the reduction.
- When a **dead end** is reached or **further solutions** are requested: we go back to the choice point visited last (the youngest) and then continue the search among the **remaining** (untried) **clauses**.
- If no new clause can be found at a choice point, then it is deleted and we backtrack further. If there are no more choice points the execution of the query fails.
- In the above example: we backtrack to the step 2 and there we try the second `(p2)` clause:

```
(...) :- parent('Imre', P), parent(P, 'Civakodó Henrik').
(p1)  :- parent('Gizella', 'Civakodó Henrik').
(p5)  □
```

Visualisation of Backtracking with a Search Tree

```
p('Imre', 'István').      % (p1)
p('Imre', 'Gizella').     % (p2)
p('István', 'Géza').      % (p3)
p('István', 'Sarolt').    % (p4)
p('Gizella', 'CH').       % (p5)
p('Gizella', 'BG').       % (p6)

gp(GC, GP) :-
    p(GC, P), p(P, GP).   % (gp)
```



- The search tree
  - the nodes are execution states
  - labels appearing on
    - nodes are queries,
    - edges are clause numbers and substitutions.
- The Prolog search: traversal of the search tree
  - from left to right,
  - depth-first search.
- The dashed line denotes unsuccessful clause searches, so called *first argument indexing* eliminates the top one.

The trace of the search space

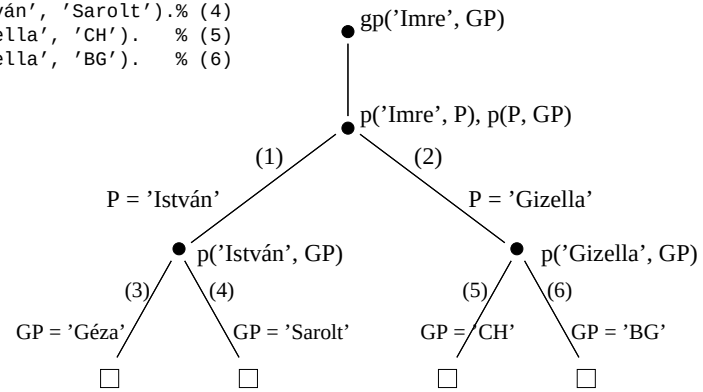
- This is an (edited) dialog with the reduction trace program. We eliminated some failing unifications.

```
|| ?- grandparent('Imre', 'Civakodó Henrik').
G0:   grandparent('Imre','Civakodó Henrik') ?            <--- continues when RET is pressed
|                                                  Trying clause 1 of grandparent/2 ... successful
| (1) {Child_1 = 'Imre', GrandParent_1 = 'Civakodó Henrik'}<--- variable renaming
|
G1:   parent('Imre',Parent_1), parent(Parent_1,'Civakodó Henrik') ?
|                                                  Trying clause 1 of parent/2 ... successful
|   (1) {Parent_1 = 'István'}
|
|-----G2:   parent('István','Civakodó Henrik') ?
(...)                                                <--- G3-G8 6 unsuccessful clause matches
|     |<<<< Failing back to goal G1                  <--- Does 'Imre' have other parents?
|                                                  Trying clause 2 of parent/2 ... successful
|   (2) {Parent_1 = 'Gizella'}
|
|-----G9:   parent('Gizella','Civakodó Henrik') ?
|     |                                            Trying clause 5 of parent/2 ... successful
|     |   (5) {}
|     |-----G14:  [] ?                                 <--- empty clause, success
|     |     |++++ Solution:  ?
|     |<<<< Failing back to goal G1              <--- see previous slide, bottom dashed line
|
|<<<< No more choices                              <--- see previous slide, top dashed line
```


Search Tree —Another Example

```
p('Imre', 'István').  % (1)
p('Imre', 'Gizella'). % (2)
p('István', 'Géza').  % (3)
p('István', 'Sarolt').% (4)
p('Gizella', 'CH').   % (5)
p('Gizella', 'BG').   % (6)
```

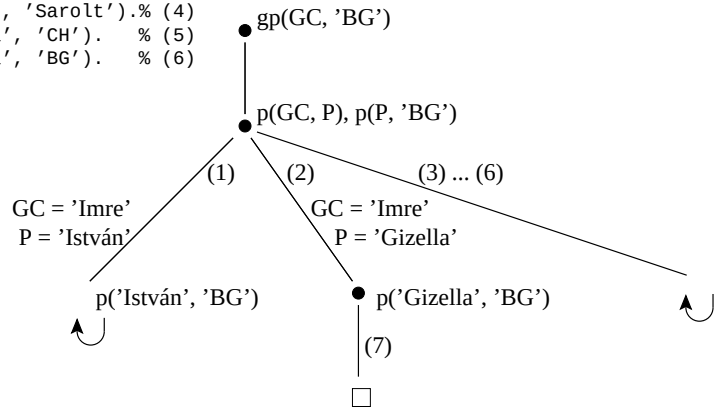
```
gp(GC, GP) :-
    gp(GC, P), sz(GC, GP).
```



Search Tree —Yet Another Example

```
p('Imre', 'István').  % (1)
p('Imre', 'Gizella'). % (2)
p('István', 'Géza').  % (3)
p('István', 'Sarolt').% (4)
p('Gizella', 'CH').   % (5)
p('Gizella', 'BG').   % (6)
```

```
gp(GC, GP) :-
    p(GC, P), p(P, GP).
```



PROCEDURAL MODELS OF PROLOG

Procedural models of Prolog execution

- A procedure is a set of clauses with the same functor
- A procedure is called via pattern matching (unification) of the call and a clause head.
- Models of Prolog execution:
  - Procedure-reduction model
    - Essentially this is the same as the goal-reduction model.
    - The base step: the reduction of a sequence of calls (i.e. a query) using a clause (this is the already known reduction step).
    - Backtrack: going back to an earlier query and trying it with another clause.
    - Advantages of the model: can be defined exactly, the search space is explicit.
  - Procedure-box model
    - Main idea: execution consists of passing through “ports” of nested procedure boxes.
    - Basic ports of a procedure box: entry, (successful) exit, failure.
    - Backtrack: asks for a new solution from an already exited procedure (“redo” port).
    - Advantages of the model: it is similar to the traditional recursive procedure model, the built-in Prolog tracing mechanism is based on this.

The Procedure-reduction Execution Model

- The main idea of the reduction execution model
  - A state of the execution: a query
  - The execution consists of two kinds of steps:
    - reduction step: a query + a clause → a new query
    - backtrack (in case of dead end): back to the last choice point
  - Choice point:
    - creation: at a reduction step which uses a any but the last clause
    - activation: at backtracking, return to the query of the choice point and try **further** clauses for matching
      (Therefore the choice point has to store in addition to the query the serial number of the used clause.)
    - the number of the choice points can be reduced by *indexing*
- The reduction model can be represented with a search tree
  - During the execution the nodes of the tree are traversed using depth-first search
  - The prolog execution engine has to store the choice points on the path from the root to the current node of the search tree—this the choice point stack.

The foundation of the reduction model: reduction step

- Reduction step: reduce a query to another query
  - reduction with a program clause (if the first goal calls a user-defined procedure):
    - The clause is **copied**, every variable systematically changed to a new variable.
    - The query is split into the first call and a residual query.
    - The first call is **unified** with the clause head.
    - The necessary substitutions are performed on the **body** of the clause and on the residual **query**
    - The new query: clause body prepended to the residual query
    - If the call and head of the clause cannot be unified then the reduction step fails.
  - reduction of a built-in call (the first goal calls a built-in procedure)
    - The query is split into the first call and a residual query.
    - The built-in procedure call is executed
    - This can be successful (and may substitute variables) or it can fail.
    - In case of success the substitutions are performed on the residual query.
    - The new query: the residual query
    - If the call of the built-in procedure fails then then the reduction step fails.

The Execution Algorithm of Prolog

1. *(Initialization:)* The stack is empty, `QU := query`
2. *(Built-in procedure:)* If the first call of `QU` is built-in then execute it,
   a. If it fails ⇒ step 6.
   b. If it is succeeds, `QU :=` the result of reduction step ⇒ step 5.
3. *(Initial value for the clause counter)* `I = 1`.
4. *(Reduction step:)* Let us consider the list of clauses applicable to the first call of `QU`. If there is no indexing, then this list will contain all clauses of the predicate, with indexing this will be a filtered subsequence. Assume the list has `N` elements.
   a. If `I > N` ⇒ step 6.
   b. Reduction step between the `I`th clause of the list and `QU` query.
   c. If this fails, then `I := I+1` ⇒ step 4.
   d. If `I < N` (non-last clause), then push `<QU,I>` on the stack.
   e. `QU :=` the result of reduction step
5. *(Success:)* If `QU` is empty, then execution ends with success, otherwise ⇒ step 2.
6. *(Failure:)* If the stack is empty, then execution ends with failure.
7. *(Backtrack:)* If the stack is not empty, then pop `<QU,I>` from the stack, `I := I+1`, and ⇒ step 4.

Indexing (preview)

- What is indexing?
  - Fast selection of the clauses applicable to a call (clauses potentially matching the call),
  - involving a **compile time** classification of the clauses of the procedure
- Most of the Prolog systems, including SICStus Prolog, do **first argument indexing**.
- The indexing is based on the outermost functor of the head argument.
  - in case of a number or a name constant `C`: the functor is `C/0`;
  - in case of structure `R` with `N` arguments: the functor is `R/N`;
  - in case of variable the functor is not defined (the clause is associated with all functors).
- Implementation of indexing:
  - At compile time: for each functor the list of the applicable clauses is built
  - At runtime: the appropriate clause list is obtained in practically constant time.
  - *Important:* if the list has a single element, no choice point is created!
- Example: the `parent('István', X)` call selects a two element clause list, but for `parent(X, 'István')` all 6 clauses are kept in the list (because the SICStus Prolog indexes the first argument only)

Reduction model —advantages and disadvantages

- Advantages
  - (relatively) simple and (relatively) precise definition
  - the search space is explicit, and graphically describable
- Disadvantages
  - It hides the exit from a predicate, eg.

```
p :- q, r.                G0:   p ?
q :- s, t.                G1:   q, r ?
s.                        G2:   s, t, r ?
t.                        G3:   t, r ?
r.                        G4:   r ?      ⇐ exit from q
                          G5:   [] ?
```

  - it does not reflect the real execution mechanism of Prolog implementations
  - it cannot be used to trace a “real” Prolog program (long queries)
- That is why another model is needed :
  - Procedure box model
  - (it is also called the 4-port-box or Byrd box model)
  - the built-in trace function of most Prolog systems is based on this model

Procedure-Box model

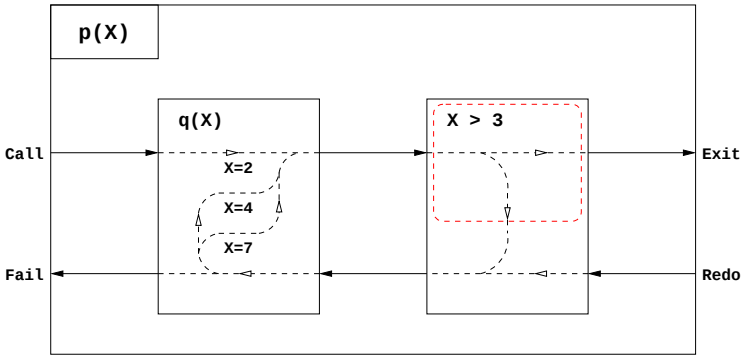
- The two phases of the Prolog procedure-execution
  - forward execution: nested procedure entries an exits
  - backward execution: requesting a new solution from an exited procedure
- A simple example:

```
q(2).  q(4).  q(7).                    p(X) :- q(X), X > 3.
```

  - Enter the `p/1` predicate (Call port)
  - Enter the `q/1` predicate (Call)
  - The `q/1` predicate exits successfully with the result `q(2)` (Exit port)
  - The `> /2` built-in predicate is entered with a `2>3` call (Call)
  - The `> /2` predicate fails (Fail port)
  - (backward execution): backtrack into the (already exited) `q/1`, asking for a new solution (Redo Port)
  - The `q/1` predicate exits with the result `q(4)` (Exit)
  - Predicate `> /2` is entered with a `4>3` call and exits successfully (Call, Exit)
  - The `p/1` predicate exits successfully with the `p(4)` result (Exit)

Procedure-Box model —graphical representation

```
q(2).  q(4). q(7).                    p(X) :- q(X), X > 3.
```



Procedure-Box model —the trace of a simple example

- The previous example tracking in SICStus Prolog

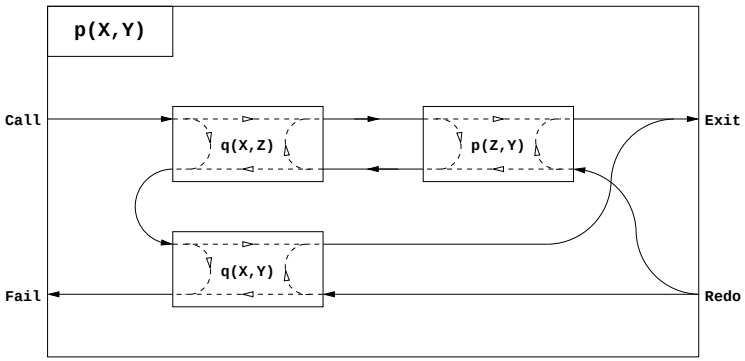
```
q(2).  q(4). q(7).

p(X) :- q(X), X > 3.


| ?- trace, p(X).
        1      1 Call: p(_463) ?
        2      2 Call: q(_463) ?
?       2      2 Exit: q(2) ?          % ? ≡ non-deterministic exit
        3      2 Call: 2>3 ?
        3      2 Fail: 2>3 ?
        2      2 Redo: q(2) ?          % backward execution
?       2      2 Exit: q(4) ?
        4      2 Call: 4>3 ?
        4      2 Exit: 4>3 ?
?       1      1 Exit: p(4) ?
X = 4 ? ;
        1      1 Redo: p(4) ?          % backward execution
        2      2 Redo: q(4) ?          % backward execution
        2      2 Exit: q(7) ?
        5      2 Call: 7>3 ?
        5      2 Exit: 7>3 ?
        1      1 Exit: p(7) ?
X = 7 ? ; no
```


Procedure-Box: A more complex example

```
p(X,Y) :- q(X,Z), p(Z,Y). p(X,Y) :- q(X,Y).
q(1,2). q(2,3). q(2,4).
```



Procedure-Box model —principles of connection

- How is the box of a “parent” predicate built from boxes of predicates called in it?
- We can assume that all clause heads contain (distinct) variables only, as the head unifications can be transformed to calls of the = /2 built-in predicate.
- Forward execution:
  - The Call port of the parent is connected to the call port of the first call of the first clause.
  - The Exit port of a predicate call is connected to
    - the Call port of the following call,
    - the Exit port of the parent if there are no following calls
- Backward Execution:
  - The Fail port of a predicate call is connected to
    - the Redo port of the preceding call, or
    - to the Call port of the first call of the following clause if there are no preceding calls
    - to the Fail port of the parent if there are no following clauses
  - The Redo port of the parent is connected to the Redo port of the last call of each clause
    - always go back to the clause from which the control exited previously

Procedure-Box model —object oriented view

- Each predicate is transformed to a class which has a constructor function (to get the call parameters) and has a method to give the "(next) solution".
- The class registers the number of the clause in which the control is.
- At the first call of the method we pass the control to the first Call port of the first clause.
- When we arrive at a Call port of a body goal an instance is **created** of the predicate called, then
- the "next solution" method is called of this predicate instance (*)
  - If this call returns with success then the control jumps to Call port of the next call or to the Exit port of the parent
  - If this call fails then the predicate instance is destroyed and we jump to Redo port of the previous call, or to the beginning of the following clause, etc.
- When we arrive at the Redo port then we continue at step (*)
- The Redo port of the parent (which corresponds to the non first call of the "next solution" method) gives the control to the last Redo port of the clause whose clause number is stored in the instance.

Box of OO approach: p/2 C++ code of method of "next solution"

```
boolean p::next()
{ switch(clno) {
  case 0:                    // entry point for the Call port
    clno = 1;                // enter clause 1:                          p(X,Y) :- q(X,Z), p(Z,Y).
    qaptr = new q(x, &z);    // create a new instance of subgoal q(X,Z)
  redo11:
    if(!qaptr->next()) {     // if q(X,Z) fails
      delete qaptr;          // destroy it,
      goto cl2;              // and continue with clause 2 of p/2
    }
    pptr = new p(z, py);     // otherwise, create a new instance of subgoal p(Z,Y)
  case 1:                    // (enter here for Redo port if clno==1)
    /* redo12: */
    if(!pptr->next()) {      // if p(Z,Y) fails
      delete pptr;           // destroy it,
      goto redo11;           // and continue at redo port of q(X,Z)
    }
    return TRUE;             // otherwise, exit via the Exit port
  cl2:
    clno = 2;                // enter clause 2:                          p(X,Y) :- q(X,Y).
    qbptr = new q(x, py);    // create a new instance of subgoal q(X,Y)
  case 2:                    // (enter here for Redo port if clno==1)
    /* redo21: */
    if(!qbptr->next()) {     // if q(X,Y) fails
      delete qbptr;          // destroy it,
      return FALSE;          // and exit via the Fail port
    }
    return TRUE;             // otherwise, exit via the Exit port
  } }
```


Backtracking —an arithmetic example

- Example: search for "good" numbers
- The task: Find a two digit number whose square has three digits and the first two digits of the square are the same as those of the original number, but in reverse order. (For example, 27: $27^2 = \mathbf{72}9$.)
- The program:

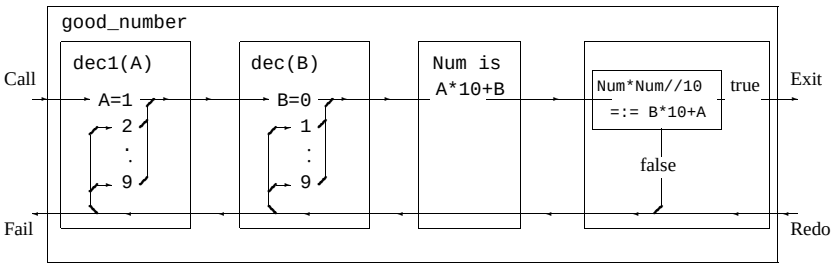
```
% dec1(J): J is a positive decimal digit.
dec1(1). dec1(2). dec1(3). dec1(4). dec1(5). dec1(6). dec1(7).
dec1(8). dec1(9).

% dec(J): J is a decimal digit.
dec(0). dec(J) :- dec1(J).

% Square of Num is a 3 digit number and begins with Num reversed.
good_number(Num):-
        dec1(A), dec(B),
        Num is A * 10 + B, Num * Num // 10 =:= B * 10 + A.
```


Prolog Execution —the 4-port procedure box model

```
good_number(Num):-
        dec1(A), dec(B),
        Num is A * 10 + B, Num * Num // 10 =:= B * 10 + A.
```



Backtracking Search —Enumeration of an Interval

- `dec(J)` enumerated the integers between 0 and 9
- Generalization: let's enumerate the integers between `N` and `M` (`N` and `M` are integers)

```
% between(M, N, I): M =< I =< N, I integer.
between(M, N, M) :-
        M =< N.
between(M, N, I) :-
        M < N,
        M1 is M+1,
        between(M1, N, I).

% dec(X): X is a decimal digit
dec(X) :- between(0, 9, X).

| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ;
Z = 14 ? ;
Z = 23 ? ;
Z = 24 ? ;
no
```


The SICStus Procedure-Box Model Debugger —the Most Important Commands

- Baisc tracing commands
  - `h <RET>` (help) — displays the list of commands
  - `c <RET>` (creep) or `<RET>` — continue tracing, stopping at every port
  - `l <RET>` (leap) — just stop at breakpoints, but keep building boxes at all ports
  - `z <RET>` (zip) — just stop at breakpoints, do not build boxes
  - `+ <RET>` resp. `- <RET>` — put/remove a spypoint on the current predicate
  - `s <RET>` (skip) — skips the body of the predicate (Call/Redo ⇒ Exit/Fail)
  - `o <RET>` (out) — exits from the body of the predicate
- The commands which influence the Prolog execution
  - `u <RET>` (unify) — unifies the current goal with a user-supplied term, instead of execution.
  - `r <RET>` (retry) — retries the execution of the current call (jumps to the Call port)
- Other commands
  - `w <RET>` (write) — writes out the call without observing the depth-limit
  - `b <RET>` (break) — enters a new, embedded Prolog interaction level
  - `n <RET>` (notrace) — switches off tracing
  - `a <RET>` (abort) — aborts the current run

FURTHER CONTROL STRUCTURES

Disjunction, example: the "ancestor" predicate

- The "ancestor" relation is the transitive closure of the "parent" relation: a parent is an ancestor (1), and an ancestor of an ancestor is also an ancestor (2), thus:

```
% ancestor0(E, Anc): Anc is ancestor of E.
ancestor0(E, P) :- parent(E, P).                                  %  (1)
ancestor0(E, Anc) :- ancestor0(E, Anc0), ancestor0(Anc0, Anc).    %  (2)
```

- The definition of `ancestor0` is mathematically correct, but gives an infinite search space:

```
parent(child,father). parent(child,mother). parent(mother,grandfather).

| ?- ancestor0(child, Anc).
   Anc = father ? ; Anc = mother ? ;  {later:} ! Error: insufficient memory
```

- The cause of the infinite recursion is that the goal `:- ancestor0(father, X).` fails at clause (1), and (2) leads to a `:- ancestor0(father, Y), ancestor0(Y, X).` goal and so on ...
- Let us eliminate the left recursion:

```
ancestor1(E, P) :- parent(E, P).                                  %  (3)
ancestor1(E, Anc) :- parent(E, P), ancestor1(P, Anc).             %  (4)

| ?- ancestor1(child, Anc).
Anc = father ? ; Anc = mother ? ; Anc = grandfather ? ; no
```

- This executes all `parent(X,Y)` subgoals twice: in (3) and in (4).

The disjunction

- The `ancestor1` predicate can be made more efficient by merging its clauses:

```
ancestor2(E, Anc) :- parent(E, P), self_or_ancestor(P, Anc).

self_or_ancestor(E, E).                                    (1)
self_or_ancestor(E, Anc) :- ancestor2(E, Anc).
```

- The `self_or_ancestor` predicate can be eliminated with the introduction of a **disjunction**:

```
ancestor3(E, Anc) :-
        parent(E, P),
        (   Anc = P
        ;   ancestor3(P, Anc)
        ).
```

- SICStus Prolog implements the above disjunction by building an auxiliary-predicate equivalent to `self_or_ancestor` and transforms `ancestor3` to `ancestor2`.
- (Recall:) The `X=Y` in-built predicate unifies its two arguments.
- The `X=Y` procedure could be defined using the fact: `U = U.` ≡ `=(U, U)`, cf. `(1)`.

The disjunction as a syntactic sugar

- The disjunction can have multiple branches. The ';' operator binds less tightly than ',' therefore the disjunction must be parenthesised, while its branches don't have to be. Example:

```
a(X, Y, Z) :-
        p(X, U), q(Y, V),
        (   r(U, T), s(T, Z)
        ;   t(V, Z)
        ;   t(U, Z)
        ),
        u(X, Z).
```

- The disjunction can always be eliminated with auxiliary-predicates.
  - We look for the variables which can be found both in the disjunction and outside it, as well
  - The auxiliary-predicate will contain these variables as arguments
  - Each clause of the auxiliary-predicate corresponds to a branch of the disjunction

```
auxiliary(U, V, Z) :- r(U, T), s(T, Z).
auxiliary(U, V, Z) :- t(V, Z).
auxiliary(U, V, Z) :- t(U, Z).

a(X, Y, Z) :-
        p(X, U), q(Y, V),
        auxiliary(U, V, Z),
        u(X, Z).
```

- The semantics of the disjunction can be defined with this auxiliary-predicate conversion.

Disjunctions —comments

- Are the clauses in 'AND' or 'OR' relation?
  - The clauses of the database are in **AND** relation, e.g.
    `parent('Imre', 'István').` `parent('Imre', 'Gizella').`
    means: Imre has parent István **AND** Imre has parent Gizella.
  - The clauses in AND relationship lead to disjuctive answers (which are in OR relation):
    `:- parent('Imre' Sz).` ⇒ `Sz = 'István' ? ; Sz = 'Gizella' ? ; no`
    The answer to the question "Who is a parent of Imre?" is: István OR Gizella.
- The above predicate with two clauses can be converted to a single clause using a disjunction:

```
parent('Imre', P) :-
        (   P = 'István'          (*)
        ;   P = 'Gizella'         (*)
        ).
```

  Thus the conjunction has been changed to a disjunction (De Morgan's laws).
- In general: all predicates can be converted to have only one clause:
  - The clauses are transformed to have identical heads using new variables and equalities:

```
parent('Imre', P) :- P = 'István'.
parent('Imre', P) :- P = 'Gizella'.
```

  - The bodies are collected into a disjunction, which forms the body of the new predicate (`(*)`).

Negation

- Task: Let us find (in the database) a parent who is **not** a grandparent!
- For this we need negation:
  - Negation by failure: the `\+ Call` structure runs the `Call` and succeeds if and only if the `Call` fails.
- A solution to the above task:

```
| ?- parent(_, X), \+ grandparent(_, X).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- An equivalent solution:

```
| ?- parent(_C, X), \+ parent(_, _C).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- What happens if the two calls are switched?

```
| ?- \+ parent(_, _C), parent(_C, X).
no
```

NF —Negation by Failure

- The `\+ Call` is a built-in meta-predicate (cf. $\nvdash$ — unprovable)
  - executes the `Call` call,
  - if `Call` completes successfully, then fails
  - else (ie. if `Call` fails) succeeds.
- During the execution of `\+ Call` at most one solution of `Call` is obtained.
- `\+ Call` never binds variables.
- Problems with the Negation by Failure:
  - "closed world assumption" (CWA) — anything that is unprovable is considered false.

```
| ?- \+ parent('Imre', X).          ----> no
| ?- \+ parent('Géza', X).          ----> true ?
```

  - $\backslash + H$ declarative semantics: $\neg\exists X(H)$, where X denotes the variables in H that are unbound *at the moment of call*.

```
| ?- \+ X = 1, X = 2.               ----> no
| ?- X = 2, \+ X = 1.               ----> X = 2 ?
```


Example: determining the co-effi cient in a linear expression

- Formula: a number, the 'x' name constant or structures built from these using the '+' and '*' operators.
- `% :- type term == {x} \/ number \/ {term+term} \/ {term*term}.`
- Linear formula: a number appears at least on one side of the '*' operator.

```
% coeff(Term, E): The coefficient of x is E in the Term linear formula.
coeff(x, 1).
coeff(Term, E) :-
        number(Term), E = 0.
coeff(T1+T2, E) :-
        coeff(T1, E1),
        coeff(T2, E2),
        E is E1+E2.
```

```
coeff(T1*T2, E) :-
        number(T1),
        coeff(T2, E0),
        E is T1*E0.
coeff(T1*T2, E) :-
        number(T2),
        coeff(T1, E0),
        E is T2*E0.
```

```
| ?- coeff(((x+1)*3)+x+2*(x+x+3), E).
E = 8 ? ;
no
```

```
| ?- coeff(2*3+x, E).
E = 1 ? ;
E = 1 ? ; no
```


Co-effi cient problem: eliminating repeated results

- with the use of negation:

```
(...)
coeff(K1*K2, E) :-
        number(K1), coeff(K2, E0), E is K1*E0.
coeff(K1*K2, E) :-
        \+ number(K1),
        number(K2), coeff(K1, E0), E is K2*E0.
```

- with the more efficient conditional construct:

```
(...)
coeff(K1*K2, E) :-
        (   number(K1) -> coeff(K2, E0), E is K1*E0
        ;   number(K2), coeff(K1, E0), E is K2*E0
        ).
```


Conditional constructs

- Syntax (`condition`, `then`, `else` are arbitrary goals):

```
(...) :-
        (...),
        (   condition -> then
        ;   else
        ),
        (...).
```

- Declarative semantics: the above form is equivalent to the following one, if the `condition` is a simple condition (cannot be solved in multiple ways):

```
(...) :-
        (...),
        (   condition, then
        ;   \+ condition, else
        ),
        (...).
```

Conditional constructs (continued)

- Procedural semantics
  The execution of a `(condition->then;else),continuation` goal is as follows:
  - The `condition` call is executed.
  - If `condition` succeeds, then the `then,continuation` subgoal remains, with the substitutions resulting from the *first* solution of the `condition`. The other solutions of the `condition` subgoal are ignored.
  - If `condition` fails, then the `else,continuation` subgoal remains without any substitution.
- Multiple branching with nested conditional constructs:

```
(   cond1 -> then1                    (   cond1 -> then1
;   cond2 -> then2                    ;  (cond2 -> then2
;   ...                               ;   ...
)                                      ...))
```

- The `else` part can be omitted, the default is: `fail`.
- The `\+ cond` negation can be replaced with the `( cond -> fail ; true )` conditional construct.

Conditional constructs —examples

- Factorial:

```
% fact(+N, ?F): N! = F.
fact(N, F) :-
        (   N = 0 -> F = 1                          % N = 0,  F = 1
        ;   N > 0, N1 is N-1, fact(N1, F1), F is N*F1
        ).
```

- The above conditional has the same meaning as the disjuntion obtained by replacing `->` with a comma (see comment), but it is more efficient, as it does't leave a choicepoint.
- Sign of number:

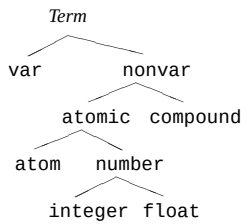
```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```


The Prolog Term: the Data Structure of Prolog

- Simple data objects:
  - Constants
    - Integers (infinity size in practice)
    - Floating point numbers
    - name constants (max 65535 characters in SICStus Prolog)
  - Variables
- compound data objects:
  - structures: ⟨ structure name ⟩`(`⟨ arg_1 ⟩`,...,`⟨ arg_n ⟩`)`
  - ⟨ structure name ⟩ is an arbitrary name constant
  - ⟨ arg_i ⟩ is an arbitrary term
  - The number of the arguments of a structure is also called its *arity*.
  - The arity of structures is between 1 and 255 in SICStus Prolog
  - The *functor* of the structure: ⟨ structure name ⟩`/`n

Prolog Terms

- Classification of Prolog terms — built-in predicates for classification



`var(X)`	X variable
`nonvar(X)`	X non-variable
`atomic(X)`	X constant
`compound(X)`	X structure
`atom(X)`	X name constant
`number(X)`	X number
`integer(X)`	X integer
`float(X)`	X fbat

- A classification predicate **checks** the **current** state of its argument, therefore it has no declarative semantics:

```
| ?- X = 1, integer(X).              ⟹ yes
| ?- integer(X), X = 1.              ⟹ no
| ?- atom('István'), atom(istvan).   ⟹ yes
| ?- compound(leaf(X)).              ⟹ yes
| ?- compound(X).                    ⟹ no
```

Unification —the Data Manipulation Mechanism of Prolog

- Unification: bringing two Prolog terms (eg. a procedure call and a procedure head) to an identical form, by possibly instantiating variables.
- Examples
  - Input parameter passing — substitutes the head variables:
    call: `grandparent('Imre', GP)`,
    head: `grandparent(C, G)`,
    substitution: `C = 'Imre', G = GP`
  - Output parameter passing — substitutes the variables of the call:
    call: `parent('Imre', P)`,
    head: `parent('Imre', 'István')`,
    substitution: `P = 'István'`
  - Input/Output parameter passing — substitutes the variables of both the call and the head:
    call: `sum_tree(leaf(5), Sum)`
    head: `sum_tree(leaf(V), V)`
    substitution: `V = 5, Sum = 5`

Unification: Substitution of Variables

- The concept of substitution:
  - The substitution is a function which assigns terms to certain variables.
    - Example: $\sigma = \{$`X`$\leftarrow$`a`, `Y`$\leftarrow$`s(b,B)`, `Z`$\leftarrow$`C`$\}$. Here $Dom(\sigma) = \{X, Y, Z\}$
    - The σ substitution assigns `a` to `X`, s(b,B) to `Y` and `C` to `Z`. Notation: $X\sigma = a$ etc.
  - The substitution function can be naturally extended to all terms:
    - $T\sigma$: σ applied to term T: the substitution σ is applied *simultaneously* to T.
    - Example: `f(g(Z,h),A,Y)`$\sigma =$ `f(g(C,h),A,s(b,B))`
  - The composition of σ and θ substitutions ($\sigma \otimes \theta$) — applying them one after the other
    - The substitution of $\sigma \otimes \theta$ assigns to $x \in Dom(\sigma)$ variables the $(x\sigma)\theta$ term, and to other $y \in Dom(\theta) \backslash Dom(\sigma)$ variables it assigns $y\theta$ ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):
      $$\sigma \otimes \theta = \{\, x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \cup \{\, y \leftarrow y\theta \mid y \in Dom(\theta) \backslash Dom(\sigma) \,\}$$
    - eg. in case of $\theta = \{$`X`$\leftarrow$`b`, `B`$\leftarrow$`d`$\}$ $\sigma \otimes \theta = \{$`X`$\leftarrow$`a`, `Y`$\leftarrow$`s(b,d)`, `Z`$\leftarrow$`C`, `B`$\leftarrow$`d`$\}$
- A term G is **more general** than S, if there exists a substitution ρ, such that $S = G\rho$
  - Example: $G =$`f(A,Y)` is more general than $S =$`f(1,s(Z))`, because in case of $\rho = \{$`A`$\leftarrow 1$, `Y`$\leftarrow$`s(Z)`$\}$, it holds that $S = G\rho$.

Unification: the Most Generic Unifier

- Terms A and B can be unified if there exists a substitution σ such that $A\sigma = B\sigma$. This $A\sigma = B\sigma$ is called a common instance of A and B.
- Two terms usually may have more common instances.
  - Example: Common instances of $A =$`f(X, Y)` and $B =$`f(s(U), U)` can be
    - $C_1 =$`f(s(a),a)` with the substitution of $\sigma_1 = \{$`X`$\leftarrow$`s(a)`, `Y`$\leftarrow$`a`, `U`$\leftarrow$`a`$\}$
    - $C_2 =$`f(s(U),U)` with the substitution of $\sigma_2 = \{$`X`$\leftarrow$`s(U)`, `Y`$\leftarrow$`U`$\}$
    - $C_3 =$`f(s(Y),Y)` with the substitution of $\sigma_3 = \{$`X`$\leftarrow$`s(Y)`, `U`$\leftarrow$`Y`$\}$
- The most generic common instance of A and B is a term C which is more general than all common instances of A and B.
  - In the example above C_2 and C_3 are the most generic common instances
- **Theorem**: The most generic common instance is unique, except for variable renaming.
- The most generic unifier of A and B is a substitution $\sigma = mgu(A, B)$ for which $A\sigma$ and $B\sigma$ are the most generic common instances of the two terms.
  - In the example above σ_2 and σ_3 are most generic unifiers.
- **Theorem**: The most generic unifier is unique, except for variable renaming.

The Unification Algorithm

- The Unification Algorithm
  - input: two Prolog terms: A and B
  - the task: determine the unifiability of the two terms
  - the result: in case of success, return the most generic unifier ($mgu(A, B)$).
- The unification algorithm, i.e. determining $\sigma = mgu(A, B)$
  1. If A and B are the same variables or constants, then $\sigma = \{\}$ (empty substitution).
  2. Else, if A is a variable, then $\sigma = \{A \leftarrow B\}$.
  3. Else, if B is a variable, then $\sigma = \{B \leftarrow A\}$.
  4. Else, if A and B are compounds with the same name and arity and their argument lists are $A_1,\ldots,A_N$ and $B_1,\ldots,B_N$ resp., and
     a. The most generic unifier of A_1 and B_1 is σ_1,
     b. The most generic unifier of $A_2\sigma_1$ and $B_2\sigma_1$ is σ_2,
     c. The most generic unifier of $A_3\sigma_1\sigma_2$ and $B_3\sigma_1\sigma_2$ is σ_3,
     d. ...

     then $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \ldots$.
  5. In all other case A and B are not unifiable.

Unification examples

- A `= sum_tree(leaf(V), V)`, B `= sum_tree(leaf(5), S)`
  - (4.) The name and arity of A and B is the same
    - (a.) `mgu(leaf(V), leaf(5))` (4th, then 2nd) $= \{$`V`$\leftarrow$`5`$\} = \sigma_1$
    - (b.) `mgu(V`σ_1`, S) = mgu(5, S)` (3rd) = $\{$`S`$\leftarrow$`5`$\} = \sigma_2$
  - so $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`V`$\leftarrow$`5`, `S`$\leftarrow$`5`$\}$
- A `= node(leaf(X), T)`, B `= node(T, leaf(3))`
  - (4.) The name and arity iof A and B is the same
    - (a.) `mgu(leaf(X), T)` (3rd) $= \{$`T`$\leftarrow$`leaf(X)`$\} = \sigma_1$
    - (b.) `mgu(T`σ_1`, leaf(3)) = mgu(leaf(X), leaf(3))` (4th, then 2nd) = $\{$`X`$\leftarrow$`3`$\} = \sigma_2$
  - so $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`T`$\leftarrow$`leaf(3)`, `X`$\leftarrow$`3`$\}$

Unification Examples in practice

- Built-in predicates related to unification:
  - `X = Y` unifies its two arguments, fails, if this is not possible.
  - `X \= Y` succeeds, if the two arguments are not unifiable, otherwise itfails.
- Examples:

```
| ?- 3+(4+5) = Left+Right.
       Left = 3, Right = 4+5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).
       T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.                % because 1+2*3 ≡ 1+(2*3)
       no
| ?- X*Y = (1+2)*3.
       X = 1+2, Y = 3 ?
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
       B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
       U = f(3), X = 3, Z = 2*2 ?
```


A Further Issue in Unification: the Occurs Check

- Question: Are `X` and `s(X)` unifiable?
  - The mathematical answer is : *no*, A variable is not unifiable with a structure in which it appears (checking this is called the “occurs check”).
  - The occurs check is costly, so it is not used by default, consequently cyclic terms may be created.
  - It is available as a standard predicate: `unify_with_occurs_check/2`
  - Extension (eg. SICStus): Proper handling of cyclic terms created by not performing occurs checks.
- Examples:

```
| ?- X = s(1,X).
       X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
       no
| ?- X = s(X), Y = s(s(Y)), X = Y.
       X = s(s(s(s(...)))), Y = s(s(s(s(...)))) ?
```


LISTS IN PROLOG

The concept of lists in Prolog

- The Prolog list
  - The empty list is 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 list tail, that is the list composed of the remaining elements.
  - Lists can be written in simplified form ("syntactic sugar").
  - The implementation of lists is optimized: it is more space- and time-efficient than for other compound structures.

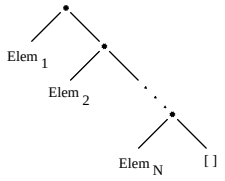
```
list_of_numbers(.(E,L)) :-
    number(E), list_of_numbers(L).
list_of_numbers([]).

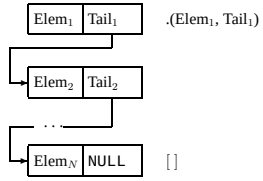
| ?- listing(list_of_numbers).
list_of_numbers([A|B]) :-
    number(A),
    list_of_numbers(B).
list_of_numbers([]).

| ?- 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 : `.(Elem`$_1$`,.(Elem`$_2$`,...,.(Elem`$_N$`,[])...))`
  - equivalent list notation: `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`]`
  - less convenient list notation: `[Elem`$_1$`|[Elem`$_2$`,...,[ Elem`$_N$`|[] ] ...]]`
- The tree structure of lists and their implementation





The notation of lists —syntactic sugar

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

- (Reminder:) Ground term: term containing no variables
- Pattern: a (usually non-ground) term, which "represents" all the terms which can be derived from it by variable substitution.
- List-pattern: pattern representing a list (but possibly other terms as well).
- Open ended list: a list pattern representing lists of any length.
- Closed list: a list(-pattern) representing 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 get ground values.
- (Some new functional languages, e.g. the Oz language support the logic variable.)
- Example: the goal below creates — in variable `L` — a list of two **identical** elements. The values of the 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` followed by those of `L2` (notation: `L3` = `L1`⊕`L2`) — two solutions:

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

```
    > 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 the `append0/3`?
  - `append/3` is **tail recursive**, equivalent to a loop (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 upfront* —using open ended lists

- The procedure append 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)` —programing the tracer, defi ning new debugger commands
  - new command: ‘`N` ⟨name⟩’ —names the argument at focus
  - standard command: ‘`^` ⟨arg number⟩’ —focuses on a given argument
  - new command: ‘`P` [⟨name⟩]’ —writes out the named terms (the one specifi ed 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 definition 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;
}
```

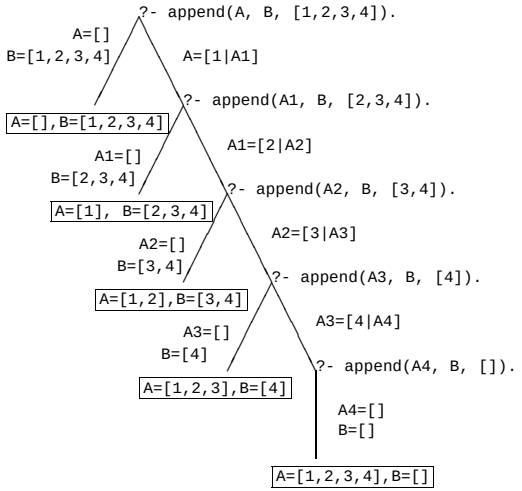
```
list revapp(list list1, list list2)
{ list l = list2;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }

  return l;
}
```


Splitting lists using append/3

```
% append(L1, L2, L3):
% The L3 list is built 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` ⊕ `L2` ⊕ `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 points
- 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).` ⇒ `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 neighbour to the right 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 an identical sublist.
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 definition 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 quoted 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, then the search space of the `select` call is infinite!

TYPES IN PROLOG

Binary tree

- Different definitions of a binary tree data type:
  - (Reminder:) Textual definition: 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 the type-notation to be introduced:

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

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

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

- Such a datatype is called **disjunctive union**, because the sets in the union are distinguished by the functors (`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(T`$_1$`, ..., T`$_n$`) }` $\equiv$ { `str(`e_1`, ..., `e_n`) | `$e_1 \in$`T`$_1$`, ..., `$e_n \in$`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}.
```

- Disjunctive union: a union in which all members have a different functor. 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`.`

```
:- 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 ⟩, ... `)` where ⟨ 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 (non-variable)
  - `-` out argument (variable)
  - `:` 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 workings of predicates described so far.

Syntactic sugar —summary, practical advices

- 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 sugar —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 sugar:
  - 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 braces: `{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 are *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 parenthesis.

```
| ?- 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. (reminder)

- ⟨ name ⟩
  - a sequence of alphanumeric characters starting with a lower case letter (lower and upper case letters, digits and underscore 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 underscore.
  - 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`, respectively (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 ⟩
  - must contain a decimal point
  - at least one (decimal) digit on both sides of the point
  - an optional exponent separated by a 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 path between two points in a weighted and unguided graph. Edges:

```
% line(A, B, L): The is a line between cities A and B and its length is L 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 of length L.
way(Start, Destination, L) :-
     (   line(Start, Destination, L)
     ;   line(Destination, Start, L)
     ).
```


Path search task —continued

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

```
% path(N, A, B, L): There exists a path consisting of (exactly)
% N sections which has a length of L.
path(0, To, To, 0).
path(N, From, To, L) :-
    N > 0,
    N1 is N-1,
    way(From, Between, L1),
    path(N1, Between, To, L2),
    L is L1+L2.
```

- An example:

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

Acyclic path search

- Loading the library to import procedures with given functors.

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

- Helper argument: the list of the visited cities in reverse order

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

% path_2(N, A, B, Excluded, L): There exists an acyclic path
% of N sections and length L, which does not touch any cities in Excluded.
path_2(0, To, To, Excluded, 0).
path_2(N, From, To, Excluded, L) :-
    N > 0, N1 is N-1, way(From, Between, L1),
    \+ member(Between, Excluded),
    path_2(N1, Between, To, [Between|Excluded], L2), L is L1+L2.
```

- An example run:

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


Further improvement: acyclic path search with route planning

- Observation: the (reversed) route is built up in the `Excluded` list.
- A **new argument** is needed in the recursive procedure to return the forward route!

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

% path_3(N, A, B, Route, L): There exists an acyclic Route route between
% A and B consisting of (exactly) N sections which has a length of L.
path_3(N, From, To, Route, L) :-
    útvonal_3(N, From, To, [From], RRoute, L),
    reverse(RRoute, Route).

% path_3(N, A, B, RRoute0, RRoute, L): There exists an acyclic route
% between A and B of N sections and length L which does not go through RRoute0.
% RRoute = (reverse of route A → B) ⊕ RRoute0.
path_3(0, To, To, RRoute, RRoute, 0).
path_3(N, From, To, RRoute0, RRoute, L) :-
    N > 0, N1 is N-1, way(From, Between, L1),
    \+ member(Between, RRoute0),
    path_3(N1, Between, To, [Between|RRoute0], RRoute, L2), L is L1+L2.

| ?- path_3(2, 'Paris', _, Route, L).
        L = 1900, Route = ['Paris','Vienna','Berlin'] ? ;
        L = 1510, Route = ['Paris','Vienna','Budapest'] ? ; no
```


Representation of weighted graph as a list of edges

- The representation of the 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

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


Finding a path free of repetitions in a graph represented as a list

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

% path_4(N, G, A, B, P, L): There exists in the graph G a path P
% of N sections and length L going from A to B.
path_4(0, _Graph, To, To, [To], 0).
path_4(N, Graph, From, To, [From|Route], L) :-
       N > 0, N1 is N-1,
       select(Edge, Graph, Graph1),
       edge_endpoints_length(Edge, From, Via, L1),
       path_4(N1, Graph1, Via, To, Route, L2),
       L is L1+L2.

% edge_endpoints_length(Edge, A, B, L): The endpoints of
% Edge undirected edge are A and B, its length is L.
edge_endpoints_length(edge(A,B,L), A, B, L).
edge_endpoints_length(edge(A,B,L), B, A, L).

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

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 to the searched value then ''true''
leaf_of_tree(node(L,_), V) :-
        leaf_of_tree(L, V). % if in left tree, then in full tree as well
leaf_of_tree(node(_,R), V) :-
        leaf_of_tree(R, V). % if in right tree, then in full tree as well
```
- The underscore is a void variable, its occurences are all different variables!
- Examples: testing (1), enumerating the leaves of a tree (2), enumerating trees with a given leaf (3) (∞ search 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)
```


Conjuctive and disjuctive traversal of compound structures

- A compound data structure can traversed in two ways:
  - Conjuctively: traversal of parts is in an AND relation, usually gives one result
    - like: the sum of tree (`sum_tree`), tree checking (`itree`), tree printing:

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

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

  - Disjunctively: traversal of parts is in an OR relation, new result at backtracking
    - like: listing of tree leaves (`tree_leaf`)
- The disjunctive, enumerating traversal can easily be complemeted with further conditions
  - 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` built-in predicate always "fails" , can be used for e.g., closing an enumeration loop.

Omitting leaves from a binary tree

- Write a predicate to decide whether a value exists in a leaf of a tree! Return the tree which remains of the original after the omission of the found leaf.

```
% tlr(Tree, Value, Remainder): With the omission of the Value leaf of the
% Tree binary tree Remainder is the remaining tree. (tlr = tree_leaf_remainder)
tlr(node(leaf(V),T), V, T).   % if the left branch is the wanted leaf
                              % then the right branch is the remaining tree
tlr(node(T,leaf(V)), V, T).   % the same for the right leaf
tlr(node(L0,R), V, node(L,R)) :-
        tlr(L0, V, L).        % if the leaf can be omitted from the left tree
                              % then its remainder completed with the right
                              % tree will be the remainder of the full tree
tlr(node(L,R0), V, node(L,R1)) :-
        tlr(R0, V, R1).       % the same for the right branch
```

- The `tlr/3` predicate can be used for checking and decomposing 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 a leaf into a binary tree

- Let us write a predicate to insert a leaf with given value into 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, Remainder): With the omission of the Value leaf of the
% Tree binary tree Remainder is the remaining tree. Tree - Value = Remainder.

% tlr(Tree, Value, Remainder): The Tree binary tree can be composed by
% inserting a Value leaf into the Remainder tree. Tree = Remainder + Value.
tlr(node(leaf(V),T), V, T).   % A leaf inserted into T tree: a single leaf
(...)                         % tree is put in front of 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

four_leaved(X, Y, Z, U, Tree) :-  % Tree is made of X, Y, Z, U leaves
     tlr(Tree0, Y, leaf(X)), tlr(Tree1, Z, Tree0), tlr(Tree, U, Tree1).

| ?- findall(Tree, four_leaved(1,3,4,6,Tree), Trees), length(Trees,L). ⟹  L = 120, Trees = (...)
```

Example: producing a term with given value

- The task: write a Prolog program for the following problem:
  - Using the numbers 1, 3, 4, 6 and the four basic arithmetic operators, produce 24!
  - All four numbers have to be used exactly once, in any order.
  - The four operators can be used arbitrarily, with any number of parentheses.
- We already have a predicate (`four_leaved/5`) building any tree from four numbers.
- Define a predicate which constructs an arithmetical term from a tree!

```
% tree_term(Tree, Term): Term is an arithmetical term, identical in its shape
%   to Tree, built of the same numbers, using the 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 composed 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 (continued)

- Predicates defined earlier:
  - `four_leaved/5` lists trees constructed of 4 given values
  - `tree_term/2` lists arithmetical terms identicalto a tree
- Using these a predicate solving the task can be easily written:

```
% four_leaved_value(X, Y, Z, U, Value, Expr): Expr is a term built of X, Y,
% Z, U and the four basic operators, and evaluates to the value of Value.
four_leaved_value(X, Y, Z, U, Value, Expr) :-
        four_leaved(X, Y, Z, U, Tree),
        tree_term(Tree, Expr),
        Expr =:= Value.

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

- Notes
  - X, Y, Z and U can be instantiated not only with specific values but with ground arithmetic expressions as well.
  - `Value is Expr` in place of `Expr =:= Value` **would not work**! Why?