Declarative programming

Péter Hanák
hanak@inf.bme.hu

Department of Control Engineering and Information Technology

Péter Szeredi, Gergely Lukácsy, András György Békés
{szeredi,lukacsy,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 frame-program
- 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

Programming languages — styles

- **Imperative**
  - *Fortran*
  - *Algol*
  - *C*
  - *C++*
  - ...
- **Declarative**
  - **Functional Languages**
    - *LISP*
    - *ML*
    - ...
  - **Logic Languages**
    - *SQL*
    - *Prolog*
    - *CLP 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 **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!

Right Recursive Functions

- Is it possible to write a right recursive code for the exponentiation (`pow(a,n)`) task?
  - The problem is that when "coming out" from the recursion we are not able to do anything more, so the result has to be available inside the last call.
  - The solution: define an auxiliary function, which has an additional argument, a so called accumulator.
- Right 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) symbol:
  - terminal symbol: `[terminal]`
  - non-terminal: `non_terminal`
  - repetition (0, 1, or more repetition, is 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

- The sequence of symbols, which is equal to an other sequence of symbols written twice in a row, is called a stutterer. More precisely:
  - a sequence of symbols is a stutterer if its length is even ($2n$) and the first n elements are the same as the last n elements
  - Examples: `adogadog`, `10311031`
- A program is to be written in the Cékla language which solves the following problem:
  - the main function should be: `stutterer(a) = b` meanomg: `b` is the smallest natural number such that the `a` natural number written in base `b` is a stutterer.
  - Examples:
    - `stutterer(4) = 3`
    - `stutterer(10) = 2`
    - `stutterer(6) = 5`
    - `stutterer(8) = 3`

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
  - backtrack
- New style 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 traversing 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
  - More 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 trial 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 stb.)
1990–. . .	Prolog appears on parallel computers
	Highly-efficient Prolog compilers
	.....

Information about Logic Programming

- Implementations of Prolog:
  - 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 intensionally 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` és `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) :-
powd2(A, N, P) :-
        (   N > 0                              N > 0,
        ->  N1 is N-1,                         N1 is N-1,
            powd(A, N1, P1),                   powd2(A, N1, P1),
            P is A*P1                          P is A*P1.
        ;   P = 1                      powd2(A, N, 1) :-     N =< 0.
        ).
```

- If a predicate has multiple clauses, Prolog tries **all of them** :
  - If the 2nd parameter of `pow2` (`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 Exercise:

  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):C’s parent P.
parent(’Imre’, ’István’).
parent(’Imre’, ’Gizella’).
parent(’István’, ’Géza’).
parent(’István’, ’Sarolt’).
parent(’Gizella’,
             ’Civakodó Henrik’).
parent(’Gizella’,
             ’Burgundi Gizella’).

%  Child’s grandparent is  Grandparent.
grandparent(Child, Grandparents) :-
    parent(Child, Parents),
    parent(Parents, Grandparents).
```

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

Data Structures in Declarative Languages —Example

- The binary tree data structure is
  - either a node (`node`) which joins two subtrees (`left,right`) into a single tree
  - 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))`
  - syntactical "sweeteners", ie. operators: `X is Y+1` ≡ `is(X, +(Y,1))`
- Variable (*var*)
  - eg. `X, Parent, X2, _valt, _, _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)