Prolog Libraries

-Constraint Logic Programming

-Solving Sudoku

-Solving Cryptarithmetic puzzle, etc

Constraint Logic Programming

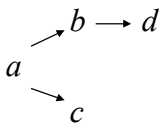
- CLP is a useful paradigm for formulating and solving problems that can be naturally defined in terms of constraints among a set of variables.
- CLP combines ***constraint satisfaction*** approach with logic programming.
- **CLP(*X*)** stands for constraint logic programming over the **domain *X***.
- There are dedicated constraint solvers for several important domains:
  - CLP(FD) for **integers** (section A.9)
  - CLP(B) for **Boolean** variables (section A.8)
  - CLP(Q) for **rational** numbers (section A.10)
  - CLP(R) for **floating point** numbers (section A.10)
- CLP is most preferably used to solve problems that consist of finding combinations of values of variables that satisfy a set of constraints.

Constraint Satisfaction

- A constraint satisfaction problem is stated as follows:
- Given:
  - A set of variables
  - The domains from which variables can take values,
  - Constraints that the variables have to satisfy
- Find:
  - An assignment of values to the variables, so that these values satisfy all given constraints.
- Constraint satisfaction is a two-step process.
  I. constraints are discovered and propagated as far as possible throughout the system.
  II. then if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.
- Constraint propagation terminates for one of two reasons:
  - A contradiction may be detected
  - The propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge.
- **A general rule: the more powerful the rules for propagating constraints, the less need there is for guessing.**

A SCHEDULING PROBLEM

- tasks a, b, c, d
- durations 2, 3, 5, 4 hours respectively
- precedence constraints



- **Constraints**:

$0 \le Ta$
$Ta + 2 \le Tb$
$Ta + 2 \le Tc$
$Tb + 3 \le Td$
$Tc + 5 \le Tf$
$Td + 4 \le Tf$

CORRESPONDING CONSTRAINT SATISFACTION PROBLEM

- **Variables**: *Ta*, *Tb*, *Tc*, *Td*, *Tf*
- **Domains**: All variables are non-negative real numbers

- **Constraints**:

  $0 \leq Ta$ (task *a* cannot start before 0)

  $Ta + 2 \leq Tb$ (task *a* which takes 2 hours precedes *b*)

  $Ta + 2 \leq Tc$ (*a* precedes *c*)

  $Tb + 3 \leq Td$ (*b* precedes *d*)

  $Tc + 5 \leq Tf$ (*c* finished by *Tf*)

  $Td + 4 \leq Tf$ (*d* finished by *Tf*)

- **Criterion**: minimise *Tf*

CLP over finite domains: CLP(FD)query format

- In CLP(FD) query, domains of variables are sets of integers.
- For example: the query 1+X=5 fails with answer false/no.

```
?- 1+X=5.                    % false
```

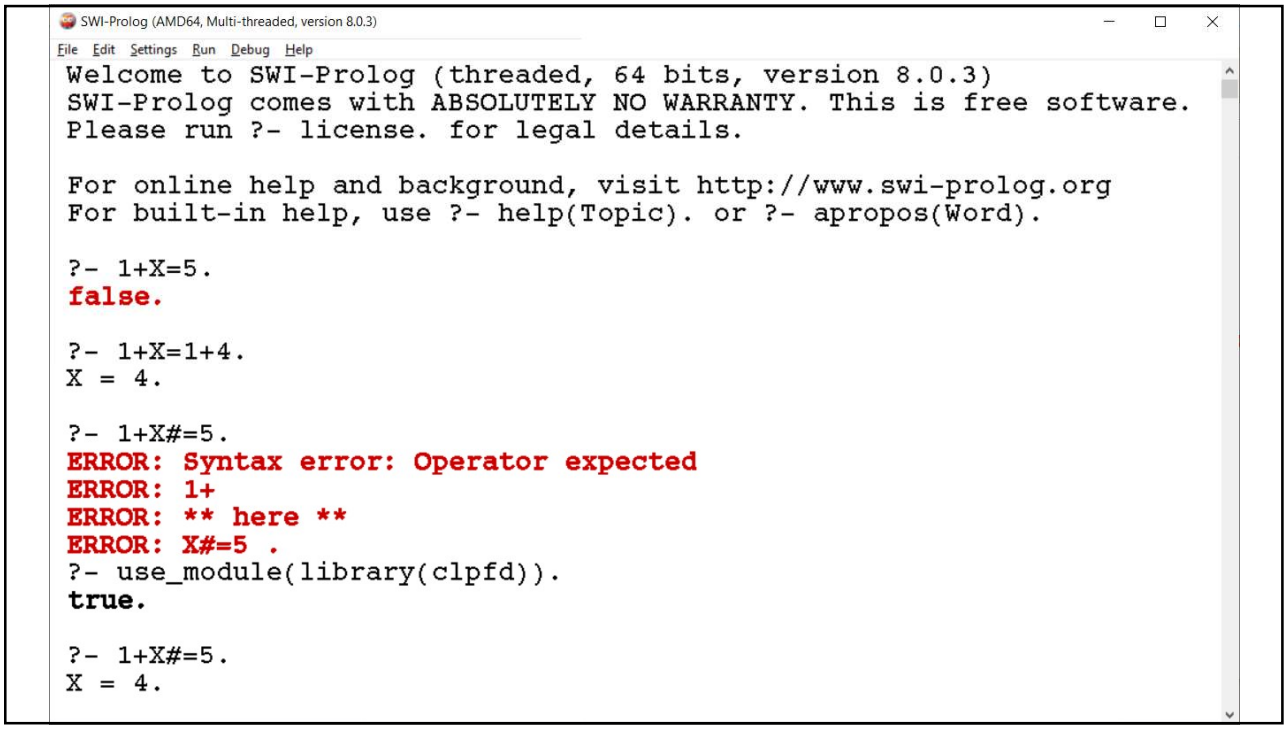
- This is how we turn it into CLP query:

```
?-1+X # = 5.                 % X = 4
```

- This is another way of posing this same query:

```
?-{1+X=5}.                   %X=4
```

- But before using this CLP query, clpfd library must be used at the command prompt.



APPLICATIONS OF CLP

- scheduling
- logistics
- resource management in production, transportation, placement

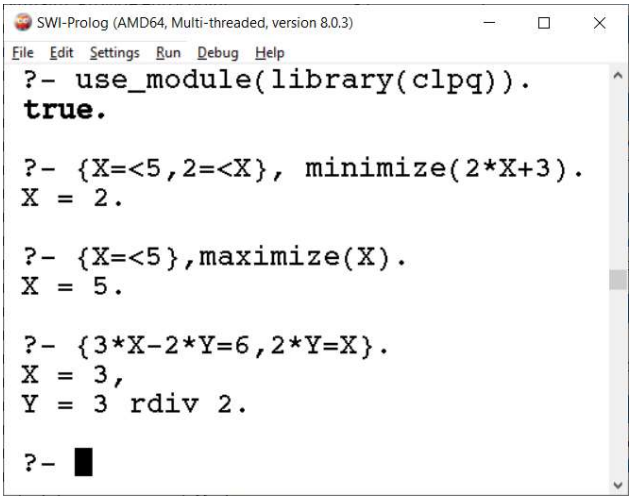
APPLICATIONS OF CLP

Typical applications involve assigning resources to activities

- machines to jobs,
- people to rosters,
- crew to trains or planes,
- doctors and nurses to duties and wards

- **clpq** library for handling constraints over the rational numbers
- **clpr** library for handling constraints over the real numbers (using floating point numbers as representation).
- {}(list of Constraints separated by comma)
  - Adds the constraints given by Constraints to the constraint store.
- minimize(Expression)
  - Minimizes Expression within the current constraint store. This is the same as computing the infimum and equating the expression to that infimum.
- maximize(Expression)
  - Maximizes Expression within the current constraint store. This is the same as computing the supremum and equating the expression to that supremum.

use_module(library(clpq))



Source: https://www.swi-prolog.org/pldoc/man?section=clpqr

clpq: CLP OVER RATIONAL NUMBERS

- Real numbers represented as quotients between integers
- Example:

?- { X = 2*Y, Y = 1-X }.

- A CLP(Q) solver gives:

X = 2/3, Y = 1/3

- A CLP(R) solver gives something like:

X = 0.666666666, Y = 0.333333333

LINEAR OPTIMISATION FACILITY

```
?- {X >=2, Y >=2, Y =< X+1, 2*Y =< 8-X, Z = 2*X +3*Y},
   maximize(Z).

X = 4.0
Y = 2.0
Z = 14.0

?- { X =< 5}, minimize( X).  no
```

SIMPLE SCHEDULING

```
?- { Ta + 2 =< Tb,      % a precedes b
     Ta + 2 =< Tc,      % a precedes c
     Tb + 3 =< Td,      % b precedes d
     Tc + 5 =< Tf,      % c finished by finishing time Tf
     Td + 4 =< Tf},     % d finished by Tf
     minimize( Tf).

Ta = 0.0, Tb = 2.0, Td = 5.0, Tf = 9.0
{Tc =< 4.0}
{Tc >= 2.0}
```

Application areas of CLP(FD)

- There are two major use cases of CLP(FD) constraints:
  - **declarative integer arithmetic** (section A.9.3)
  - solving **combinatorial problems** such as planning, scheduling and allocation tasks.
- The predicates of this library can be classified as:
- *arithmetic* constraints like #= /2, #> /2 and #\= /2 (section A.9.17.1)
- the *membership* constraints in /2 and ins /2 (section A.9.17.2)
- the *enumeration* predicates indomain /1, label /1 and labeling /2 (section A.9.17.3)

Arithmetic constraints in clpfd

- In total, the arithmetic constraints are:

Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

Arithmetic Expressions with clpfd

- *Expr1* and *Expr2* denote **arithmetic expressions**, which are:

  where *Expr* again denotes an arithmetic expression

  bitwise operations (\ ) / 1 , (/ \ ) / 2 , (\ /) / 2 , (>>) / 2 , (<<) / 2 , lsb / 1, msb / 1 , popcount / 1 and (xor) / 2 are also supported.

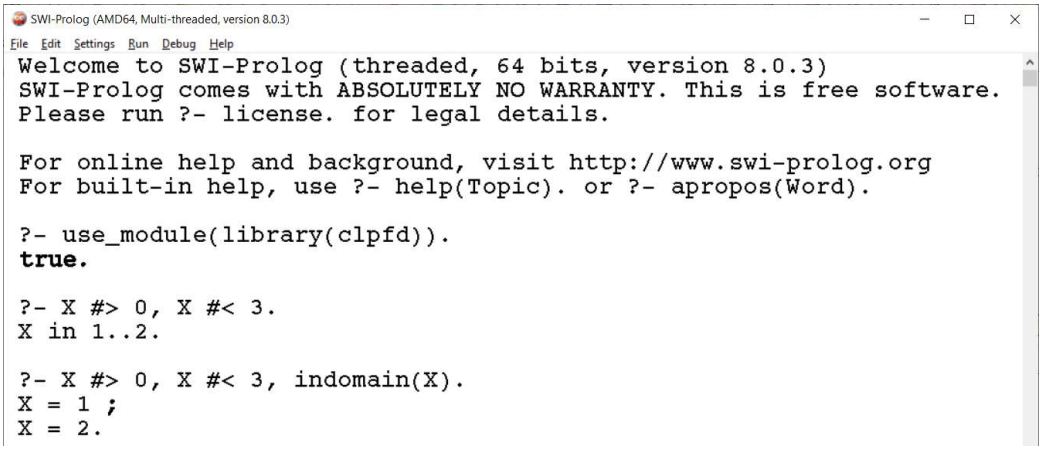
integer	Given value
variable	Unknown integer
?(*variable*)	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
Expr rem Expr	Modulo induced by truncated division
abs(Expr)	Absolute value
Expr // Expr	Truncated integer division
Expr div Expr	Floored integer division

#\ Q	True iff Q is false
P #\/ Q	True iff either P or Q
P #/\ Q	True iff both P and Q
P #\ Q	True iff either P or Q, but not both
P #<==> Q	True iff P and Q are equivalent
P #==> Q	True iff P implies Q
P #<== Q	True iff Q implies P

-here P and Q are Boolean values.
-When reasoning over Boolean variables, also consider using CLP(B) constraints as provided by library(clpb).

indomain (enumeration predicate)

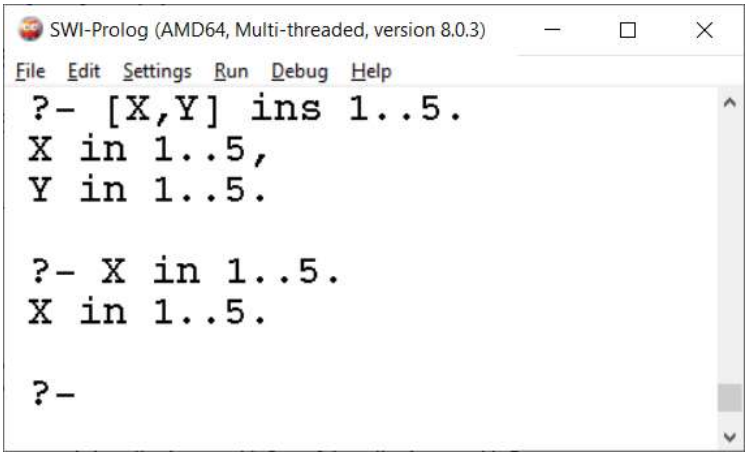
- **Built-in predicate indomain(X) assigns through backtracking all possible values to X.**



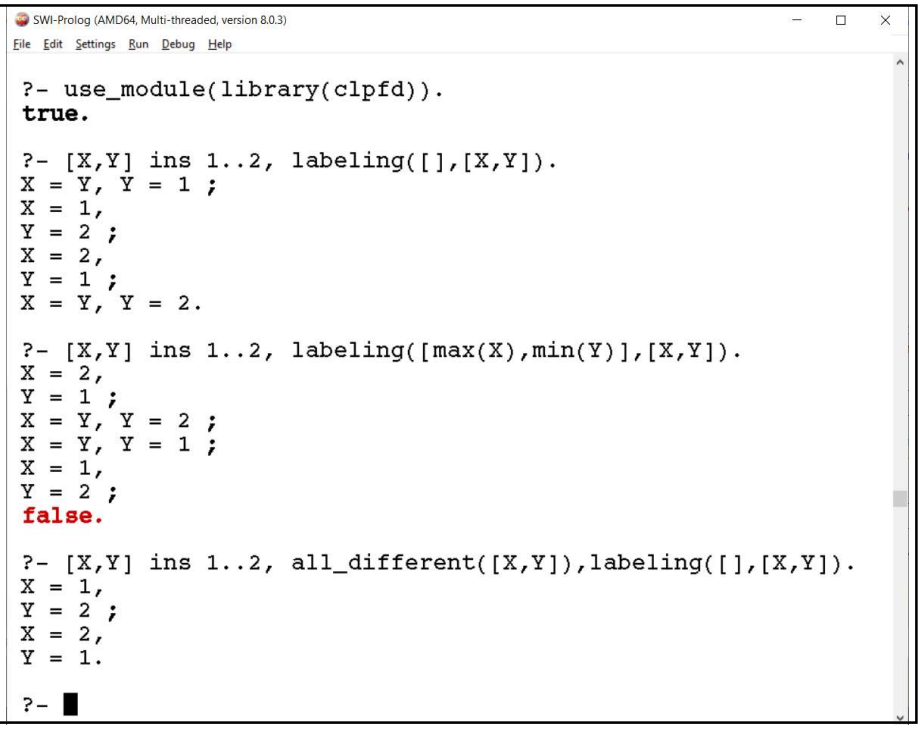
Domain and labeling

- Built-in predicate **domain(L, Min, Max)** means all the variables in List L have domains Min..Max.
- Built-in predicate **labeling(Options, L)** generates concrete possible values of variables in list L.
  - Here Options variable is a list of options regarding the order in which variables in L are labelled.
  - If Options = [ ] then by default the variables **are labelled from left to right, taking the possible values one by one from smallest to largest**.
- label: equivalent to labeling([ ], Vars).

Specifying range of numbers (in/ins arity 2)



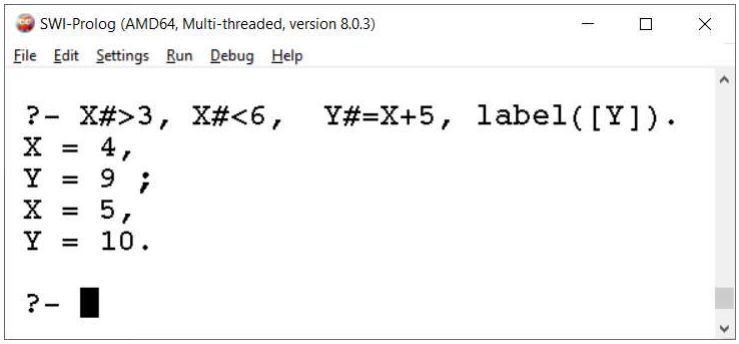
labeling demo



label

- The builtin predicate label is same as labeling but it takes just one parameter and does not provide any order for variable assignment.

?- X#>3, X#<6, Y#=X+5, label([Y]).



Solving puzzle SEND+MORE=MONEY

```
:- use_module(library(clpfd)).
puzzle([S,E,N,D] + [M,O,R,E] =
[M,O,N,E,Y]) :-
  Vars = [S,E,N,D,M,O,R,Y],
  Vars ins 0..9,
  all_different(Vars),
          S*1000 + E*100 + N*10 + D +
          M*1000 + O*100 + R*10 + E #=
M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

- Sample queries for puzzle.

```
?- puzzle(As+Bs=Cs),
label(As).
```

Or we can try just labelling one variable:

```
?- puzzle([S,E,N,D] +
[M,O,R,E] = [M,O,N,E,Y]),
label([N]).
```

```
?- puzzle([S,E,N,D] +
[M,O,R,E] = [M,O,N,E,Y]),
label([E]).
```

Output of Cryptarithmetic puzzle

```
% d:/prolog2020/crypt_arith.prolog compiled 0.00 sec, 2 clauses

?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.

?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]), label([N]).
S = 9,
E = 5,
N = 6,
D = 7,
M = 1,
O = 0,
R = 8,
Y = 2 ;
false.

?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]), label([E]).
S = 9,
E = 5,
N = 6,
D = 7,
M = 1,
O = 0,
R = 8,
Y = 2 ;
false.
```

Built-in predicate append with arity 2

- **append**(*ListOfLists, List*): Concatenate a list of lists.
  - Is true if *ListOfLists* is a list of lists, and *List* is the concatenation of these lists.
  - Used to change collection of 1D lists into a 2D list/Matrix.

- **transpose***(Matrix, Transpose)*: *Transpose* a list of lists of the same length. Example:

```
?- transpose([[1,2,3],[4,5,6],[7,8,9]], Ts).
Ts = [[1, 4, 7],
      [2, 5, 8],
      [3, 6, 9]].
```

Built-in predicate length /2

- **length(*List, Int*)**
  - **True if *Int* represents the number of elements in *List*.**
- This predicate is a true relation and can be used to find the length of a list or **produce a list (holding variables) of length *Int*.**
- The predicate is non-deterministic, producing lists of increasing length if *List* is a *partial list* and *Int* is unbound.
- It raises errors if
  - *Int* is bound to a non-integer.
  - *Int* is a negative integer.
  - *List* is neither a list nor a partial list. This error condition includes cyclic lists.
- **same_length**(*List1, List2*): is true when *List1* and *List2* are lists with the same number of elements.
  - The predicate is deterministic if at least one of the arguments is a proper list.
  - It is non-deterministic if both arguments are partial lists.

maplist / 2

- True if *Goal* is successfully applied on all matching elements of the list.
- The maplist family of predicates is defined as:

```
maplist(P, [X11,...,X1n], ..., [Xm1,...,Xmn]) :-
P(X11, ..., Xm1), ... P(X1n, ..., Xmn).
```

- This family of predicates is deterministic iff *Goal* is deterministic and *List1* is a proper list, i.e., a list that ends in [ ].

Solving Sudoku

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

sudoku(Rows) :-
 length(Rows, 9),
        maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
         maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2],
[N7,N8,N9|Ns3]) :-
   all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
   blocks(Ns1, Ns2, Ns3).

 ?-problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
```

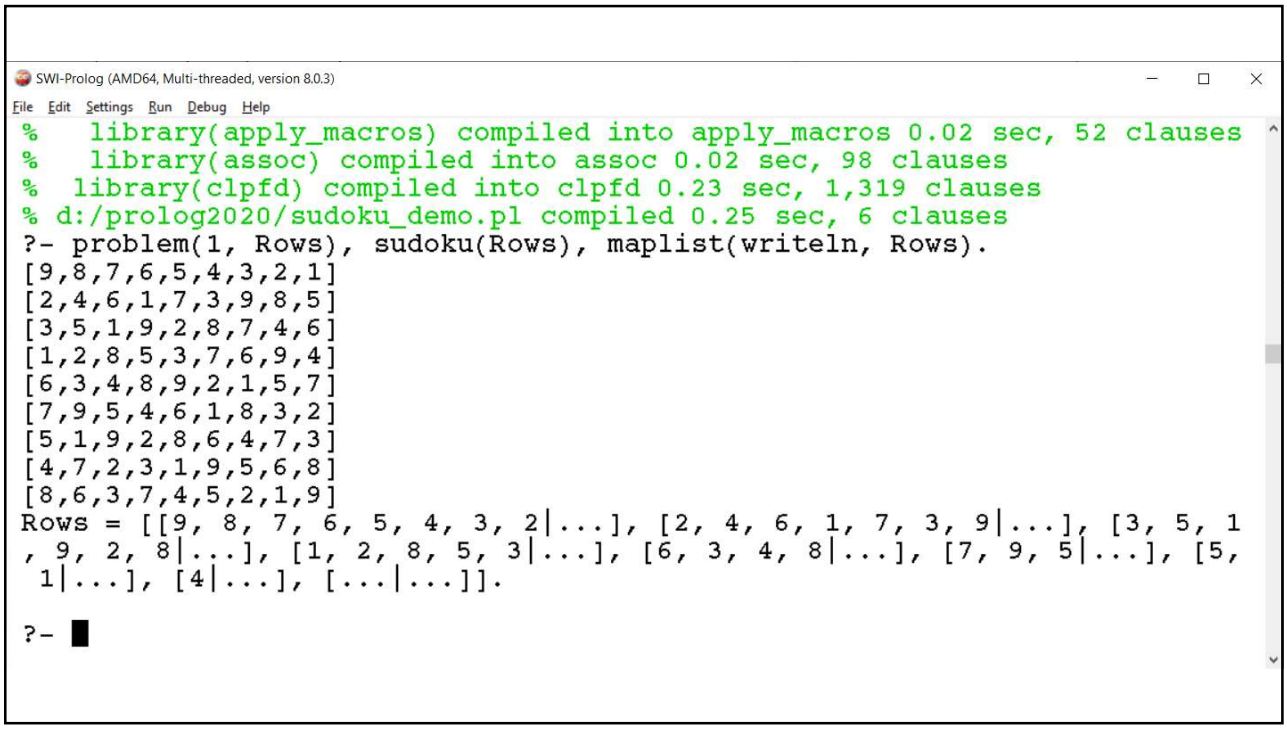
```
problem(1,
[[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,3,_,8,5],
[_,_,1,_,2,_,_,_,_],
[_,_,_,5,_,7,_,_,_],
[_,_,4,_,_,_,1,_,_],
[_,9,_,_,_,_,_,_,_],
[5,_,_,_,_,_,_,7,3],
[_,_,2,_,1,_,_,_,_],
[_,_,_,_,4,_,_,_,9]]).
```

Sudoku matrix

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

sudoku(Rows) :-
 length(Rows, 9),
maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).

blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
   all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
   blocks(Ns1, Ns2, Ns3).

 ?-problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
```

As	N1	N2	N3	N1'	N2'	N3'	N1''	N2''	N3''
Bs	N4	N5	N6	N4'	N5	(3) N6		8	5
Cs	N7	N8	(1) N9	N7'	(2) N8	N9'			
Ds				5		7			
Es			4				1		
Fs		9							
Gs	5							7	3
Hs			2		1				
Is					4				9

Explanation of predicate blocks(L1,L2,L3).


SWI-Prolog (AMD64, Multi-threaded, version 8.0.3)
File Edit Settings Run Debug Help
% library(apply_macros) compiled into apply_macros 0.02 sec, 52 clauses
% library(assoc) compiled into assoc 0.02 sec, 98 clauses
% library(clpfd) compiled into clpfd 0.23 sec, 1,319 clauses
% d:/prolog2020/sudoku_demo.pl compiled 0.25 sec, 6 clauses
?- problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
[9,8,7,6,5,4,3,2,1]
[2,4,6,1,7,3,9,8,5]
[3,5,1,9,2,8,7,4,6]
[1,2,8,5,3,7,6,9,4]
[6,3,4,8,9,2,1,5,7]
[7,9,5,4,6,1,8,3,2]
[5,1,9,2,8,6,4,7,3]
[4,7,2,3,1,9,5,6,8]
[8,6,3,7,4,5,2,1,9]
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], [2, 4, 6, 1, 7, 3, 9|...], [3, 5, 1
, 9, 2, 8|...], [1, 2, 8, 5, 3|...], [6, 3, 4, 8|...], [7, 9, 5|...], [5,
1|...], [4|...], [...|...]].
?-