Foundations of logic programming

First of all, functors is a functional symbol, used to enriches the set of terms when describing the structure of data.
For example:

- `cons(a, [])` – describes the list `[a]`. `[]` is an individual constant, standing for the empty list. The `cons` functor ha syntactic sugar notation as an infix operator |: `cons(a,[])` is written: `[a | []]`.
- `cons(b,cons(a,[ ]))` the list `[b, a]`, or `[b | [a | []]]`. The syntax `[b, a]` uses the printed form of lists in Prolog.
- `cons(cons(a, []), cons(b, cons(a, [])))` – the list `[[a], [b, a]]`, or `[[a | []] | [b | [a, | []]]]`.
- `time(monday, 12, 14)`
- `street(alon, 32)`
- `tree(Element, Left, Right)` – a binary tree, with `Element` as the root, `Left` and `Right` as its sub-trees.
- `tree(5, tree(8, void, void), tree(9, void, tree(3, void, void)))`

There are a lot of definitions about how functors work. By the way, every functor has an arity, we can use it to specify the arguments that the functor expects to show.

Also, there are a lot of different types of functors, like nested. functors can be nested , so we can use(cons(cons(cons ……) forever.

- `p(f(f(f(g(a, g(b, c))))))` – `p` is a predicate symbol, while `f`, `g`, are functors.
- `ancestor(mary, sister_of(friend_of(john)))` – `ancestor` is a predicate symbol, and `sister_of`, `friend_of` are functors.
- `course(ppl, time(monday, 12, 14), location(building34, 201))` – `course` is a predicate symbol, and `time`, `location` are functors.
- `address(street(alon, 32), shikun_M, tel_aviv, israel)` – `address` is a predicate symbol, and `street`, `shikun_M` are functors.

Logic variables

It's just a placeholder, because in traditional programming languages, variables are labels of storing locations.
The most fundamental difference lies in the concept of variables. In functional programming, variables represent specific values. Local variables only represent operations in that part of the area. Here we only consider them in Haskell:

```
> let v = iterate (tail) [1..3]
> v
[[1,2,3],[2,3],[3],[],*** Exception: Prelude.tail: empty list
```

After the fourth element, the value is indeterminate. However, you can safely use the first four elements:

```
> take 4 v
[[1,2,3],[2,3],[3],[]]
```

Please note that the syntax in the function program is strictly restricted to avoid undefined variables.

In logic programming, variables do not need to refer to specific values. Therefore, if we want to list 3 elements, we can say:

```
?- length(Xs,3).
Xs = [_G323, _G326, _G329].
```

Obviously, the elements of the list are variables. All of these variables may be solutions. Like Xs = [1,2,3]. Now, let's assume that the other elements of the first element are different:

```
?- length(Xs,3), Xs = [X|Ys], maplist(dif(X), Ys).
Xs = [X, _G639, _G642],
Ys = [_G639, _G642],
dif(X, _G642),
dif(X, _G639).
```

Now, we may need all the elements in Xs are same

```
?- maplist(=(_),Xs).
Xs = [] ;
Xs = [_G231] ;
Xs = [_G231, _G231] ;
Xs = [_G231, _G231, _G231]  ;
Xs = [_G231, _G231, _G231, _G231] .
```

Therefore, what we are showing here is a list without a third element, where the first element is different from the other elements, and all elements are equal.

Unification

It’s a process of algorithm when solving expressions. It’s like a “bound” to bind different variables, and can be used in different type of interfaces.

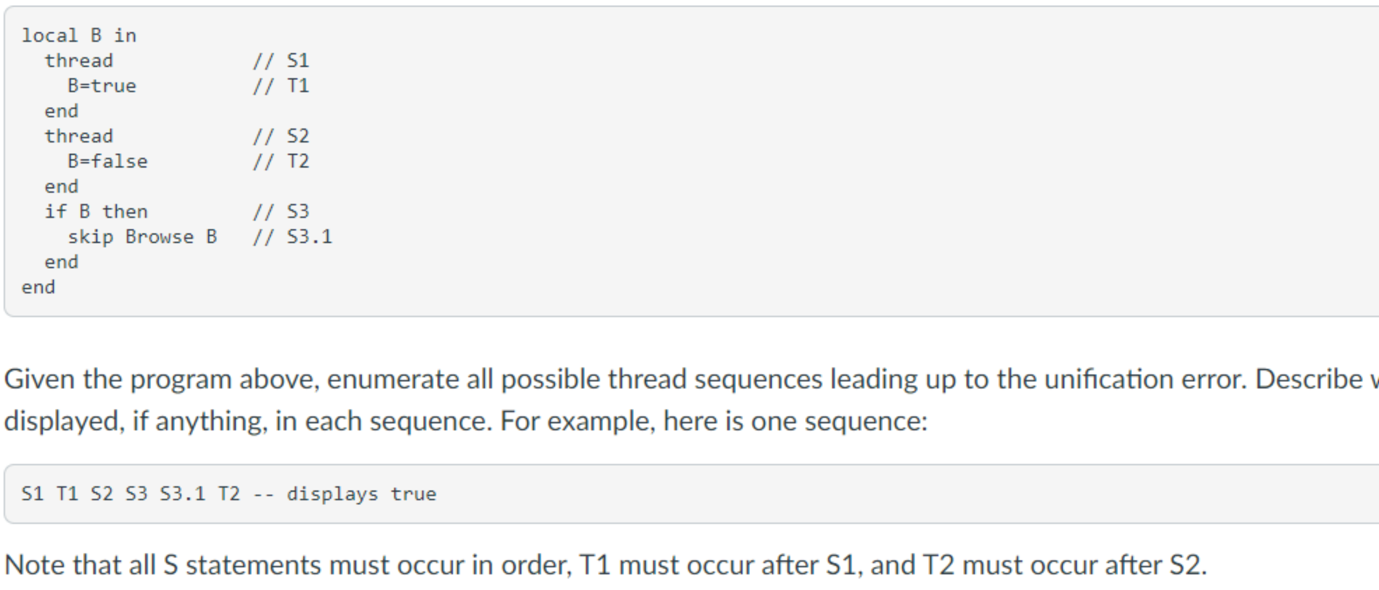
The main purpose is to make expressions look the same. we can use “substitution” to replace one variable with another

For example:
in prolog

```
?- X = Y.            ** Unification binds two differently named variables **
   X=_125451         ** to a single, unique variable name **
   Y=_125451
```

this is easiest expression to represent unification

In our homework of lab6



```
local B in
  thread            // S1
    B=true          // T1
  end
  thread            // S2
    B=false         // T2
  end
  if B then         // S3
    skip Browse B   // S3.1
  end
end
```

Given the program above, enumerate all possible thread sequences leading up to the unification error. Describe
displayed, if anything, in each sequence. For example, here is one sequence:

```
S1 T1 S2 S3 S3.1 T2 -- displays true
```

Note that all S statements must occur in order, T1 must occur after S1, and T2 must occur after S2.

we can do it like:
1:
S1 T1 S2 T2 S3 S3.1 -- T2 does not unify
becauseT1: B already returned true, for the second one T2: B cannot be used again to represent different variables.

2.
S1 S2 S3 S3.1 T2 T1 -- Does not unify
because S1 may match T2, S2 may match T1. both cannot match to the original “T”

3.
S2 T2 S1 T1 S3 S3.1 -- T2 does not unify
because T2 and T1 both represent the same “B”, but different values, once S2 matched T2, S1 and T1 cannot match.cause S2 and T2 return false, but s3 and s3.1 only If B is true, then skip browse B. that’s why cannot unify

4.
S2 T1 S3 S3.1 S1 T2 -- display true, but can’t unify B in value creation

because S2 matched to true, then went S3 and S3.1. After that, we recalled thread (S1), but it cannot unify the first thread.

5.
S1 T2 S3 S3.1 T2 T1 -- Does not unify
B already represents false, then EXU1 (B is true) then skip, but B is not true, so cannot skip browse B, which means they are not unify.

6.
S1 T2 S3 S3.1 S2 T1 can't uniify B in value creation
from thread S1 then T2 (B)= false, after that, pass through S3 and S3.1, only if B is true then skip browse B. So cannot unify B in value creation.

7. S1 S2 T1 T2 S3 S3.1 cannot unify
because S1 match to T1 , then S2 matches T2, but they are not the same variable. it will cause error

8. S2 S1 T2 T1 S3 S3.1 does not unify
It's the same reason with the No.7

9 S1 T2 S2 T1 S3 S3.1 does not unify
also same reason like the previous 2, S1 S2 and T1 T2 all match before S3 and s3.1. In this case, True and False all matched, so they cannot unify

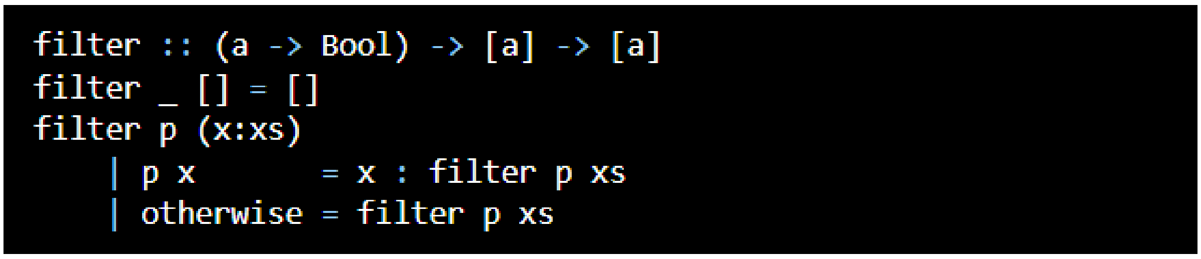
Predicate

In logic programming, we often use a set of predicates. A predicate is usually defined as a set of clauses: define a function, and each predicate has its own independent meaning. For example, factorial, which is the most common in our basic learning of programming languages:

```
factorial(0, 1).              // it is true that a factorial of 0 is 1
factorial(X, Y) :-            // it is true that a factorial of X is Y, when all
    X1 is X - 1,                  // there is a X1, equal to X - 1,
    factorial(X1, Z),             // and it is true that factorial of X1 is Z,
    Y is Z * X.                   // and Y is Z * X
```

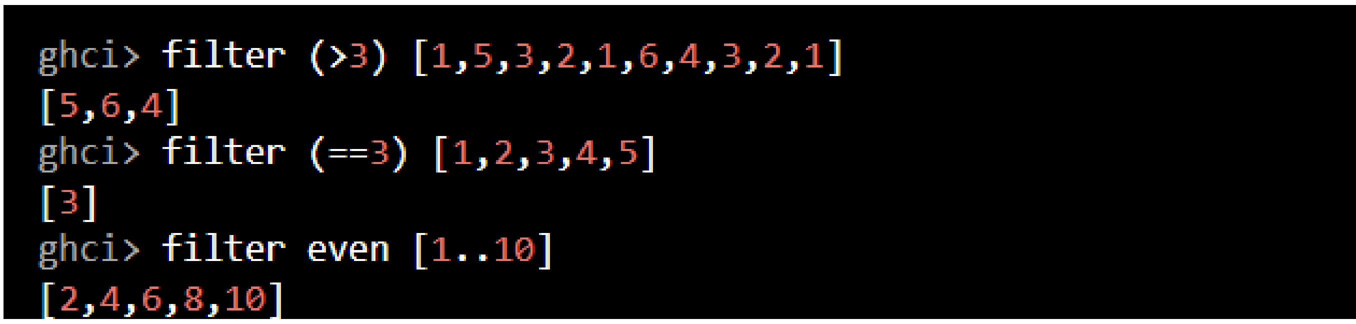
There are a lot of predicates in Haskell

Like "Filter", if p x is true, go x: filter p xs, otherwise filter p xs



```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
     | p x       = x : filter p xs
     | otherwise = filter p xs
```

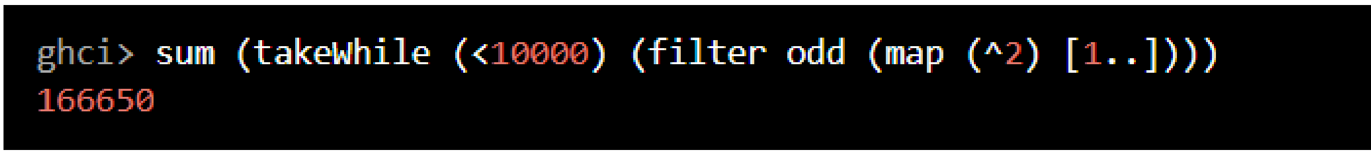
let’s go this one:



```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
```

for all the numbers are greater than 3, equal to 3, and even, we output it

Also, there is a predicate called takewhile



```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

it starts from the beginning of the list, then return the numbers when predicate is true

backtracking search

Backtracking is a general algorithm, when we don’t find the answers we want in some questions, we may need it

```
procedure bt(c)
  if reject(P,c) then return
  if accept(P,c) then output(P,c)
  s ← first(P,c)
  while s ≠ Λ do
    bt(s)
    s ← next(P,s)
```

Including riddles, like crosswords, Sudoku, through combination optimization questions, we will also need to go back to find the correct answer.

in the other example: L=[3, 8, 13, 14, 15, 17] and N=40, then we need to use backtracking to find the solution.
we can use:

```
subl([],[]).
subl([X|Q],[X|P]):-subl(Q,P).
subl([_|Q],P):-subl(Q,P).

sums([],0).
sums([X|Q],N):-sums(Q,M),N is M+X.

discr(L,R):-subl(L,R),sums(R,40).
```

then we get [3, 8, 14, 15] is a solution, and for sure there is more than one solution in this question.

negations-as-failure
In logic programming, negation is used as a failure.
`?- not((P,Q)).`

means: If P or Q fails, we will succeed, and if both fail, we will fail.
For this example, we can describe like:

If you do not win the lottery, you need to find a job!

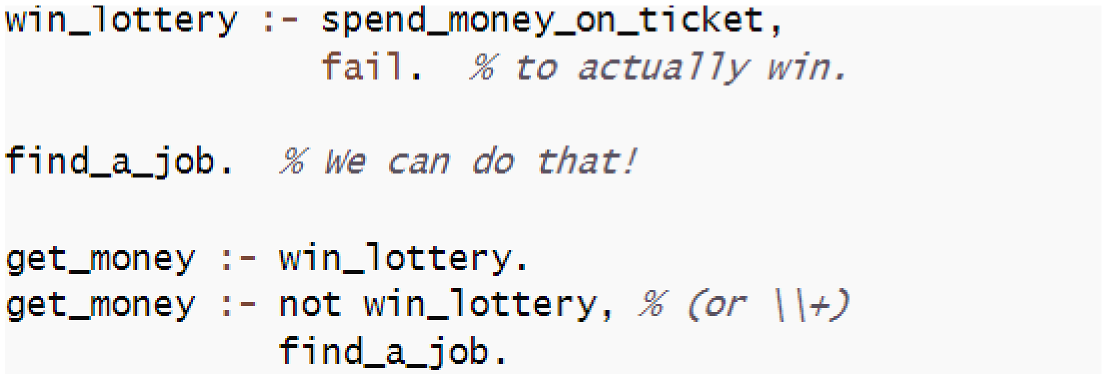
Okay, I want to buy a ticket!

...Later...

I think I need to find a job.

Okay, I'm going to find a job (because I don't know I will buy lottery tickets).

Therefore, "default negation" is the default number, unless otherwise specified, and "negation is failure" means to try first before knowing the failure.



```
win_lottery :- spend_money_on_ticket,
               fail.  % to actually win.

find_a_job.  % We can do that!

get_money :- win_lottery.
get_money :- not win_lottery, % (or \\+)
             find_a_job.
```

The logical semantics of section 9.3.2
There are two ways of expressing logic programs: logic view and operation view. In the logical view, it is just a logical statement. inside
Operation view, which defines the execution on the computer.
Let’s make an example of this:

$(p \land q) \rightarrow (\sim r \lor p)$. This formula is composed of conjunction, disjunction, implication and negation. We have established functions for negation. In order to distinguish between them, we call the negation function fn, the conjunctive function fc, the disjunctive function fd, and the implication fi. In this way, we have the following Four functions:
$fn(x)=\sim x$
$fc(x, y) = x \land y$
$fd(x, y) = x \lor y$
$fi(x, y) = x \rightarrow y$
Substitute the above value into:
$fn(x)=\sim x, x = r, \therefore fn(r)=\sim r$
$fd(x, y) = x \lor y, x = fn(r), y = p, \therefore fd(fn(r), p) = \sim r \lor p$
$fc(x, y) = x \land y, x = p, y = q, \therefore fc(p, q) = p \land q$

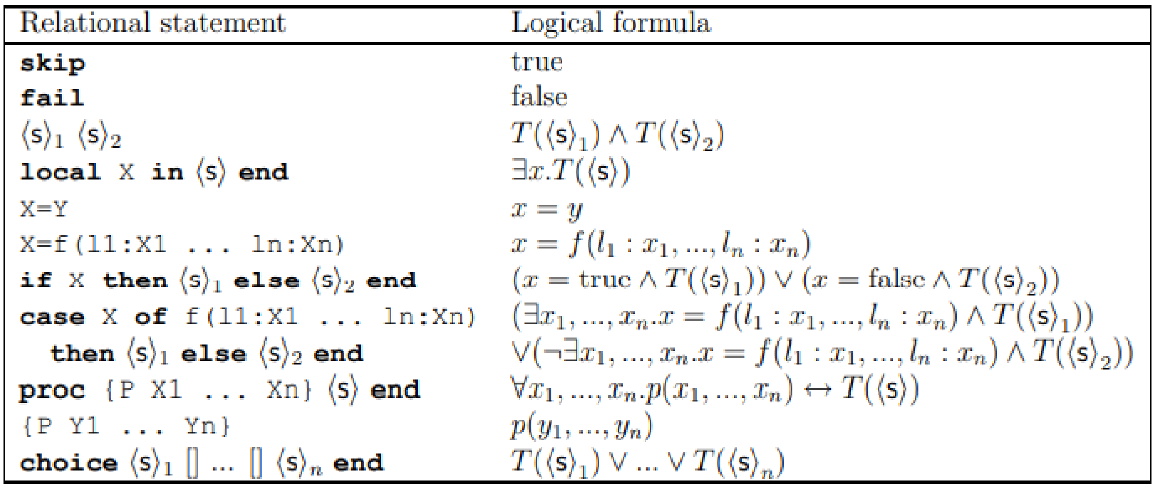
ƒi(x, y) = x → y, x = ƒc, y = ƒd(ƒn(r), p), ∴ ƒi(x, y) = ƒc → ƒd = (p ∧ q) → (~r ∨ p)
These symbols look a bit dizzying, but in fact they are the basic operations of propositional logic that we are familiar with, but we "rewrite" them with function symbols.

Also, we always use “append” in Oz language

```
fun {Append A B}
   case A
   of nil then B
   [] X|As then X|{Append As B}
   end
end
```

we can explain like: in function { Append A B}
case of A, if nil(nothing) then we return B
otherwise X| {Append As B}

And in logic programming
we always we relational statement “skip” and “fail” to represent “true” and “false” in logical formula



Relational statement	Logical formula
skip	true
fail	false
$\langle s\rangle_1$ $\langle s\rangle_2$	$T(\langle s\rangle_1) \wedge T(\langle s\rangle_2)$
local X **in** $\langle s\rangle$ **end**	$\exists x.T(\langle s\rangle)$
X=Y	$x = y$
X=f(l1:X1 ... ln:Xn)	$x = f(l_1 : x_1, ..., l_n : x_n)$
if X **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**	$(x = \text{true} \wedge T(\langle s\rangle_1)) \vee (x = \text{false} \wedge T(\langle s\rangle_2))$
case X **of** f(l1:X1 ... ln:Xn)	$(\exists x_1, ..., x_n.x = f(l_1 : x_1, ..., l_n : x_n) \wedge T(\langle s\rangle_1))$
then $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**	$\vee(\neg\exists x_1, ..., x_n.x = f(l_1 : x_1, ..., l_n : x_n) \wedge T(\langle s\rangle_2))$
proc {P X1 ... Xn} $\langle s\rangle$ **end**	$\forall x_1, ..., x_n.p(x_1, ..., x_n) \leftrightarrow T(\langle s\rangle)$
{P Y1 ... Yn}	$p(y_1, ..., y_n)$
choice $\langle s\rangle_1$ [] ... [] $\langle s\rangle_n$ **end**	$T(\langle s\rangle_1) \vee ... \vee T(\langle s\rangle_n)$

Those are pretty important, and included most of expressions in programming
There are 3 different types of “append”. First is normally append we mentioned above. second is another deterministic append, and nondeterministic append.

```
proc {Append A B ?C}
   case A
   of nil then C=B
   [] X|As then Cs in
      C=X|Cs
      {Append As B Cs}
   end
end
```

comparing both of logic.

```
proc {Append A B ?C}
   if B==C then A=nil
   else
      case C of X|Cs then As in
         A=X|As
         {Append As B Cs}
      end
   end
end
```

the last two argument of this version is input, and first argument is output. It looks different, but same logic semantics.

in my own example:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

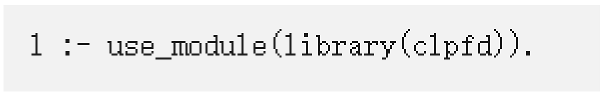
The append function defines the list obtained by appending ys to the end of xs. We think that append is a function from two lists to another, and the runtime system is designed to calculate the result of the function when the function is called on the two lists.

Logic attachment can be used to define the relationship between several lists. We regard append as a predicate that is true or false for any three given lists, and the runtime system aims to discover that the predicate is true when calling certain parameters bound to a specific list and certain remaining parameters. value. Unbound.

What makes logical append unique is that you can use it to calculate the list obtained by appending one list to another, but you can also use it to calculate the list that needs to be appended to the end of another list to get the third A list (or whether such a list does not exist), or a calculation that requires another list to be appended to obtain a third list, or to give you two possible lists, which can be appended together to obtain a given third List (and explore all possible methods).

Part2
Finite domain constraints(Swish CLP)
First of all, this chapter talks about how we use constraints logic programming by finite domain constraints. CLP is a respectful way to figure out how to get the relation of the prolog.

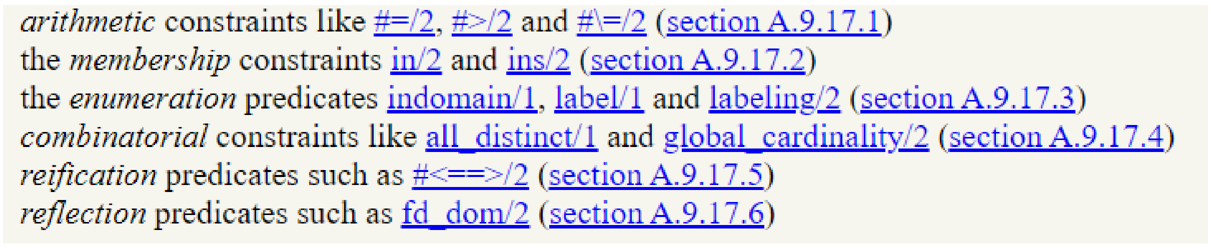
The clp(fd) library can be easily activated:



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

Some clp (fd) constraints can be optimized, and runtime libraries can be loaded.

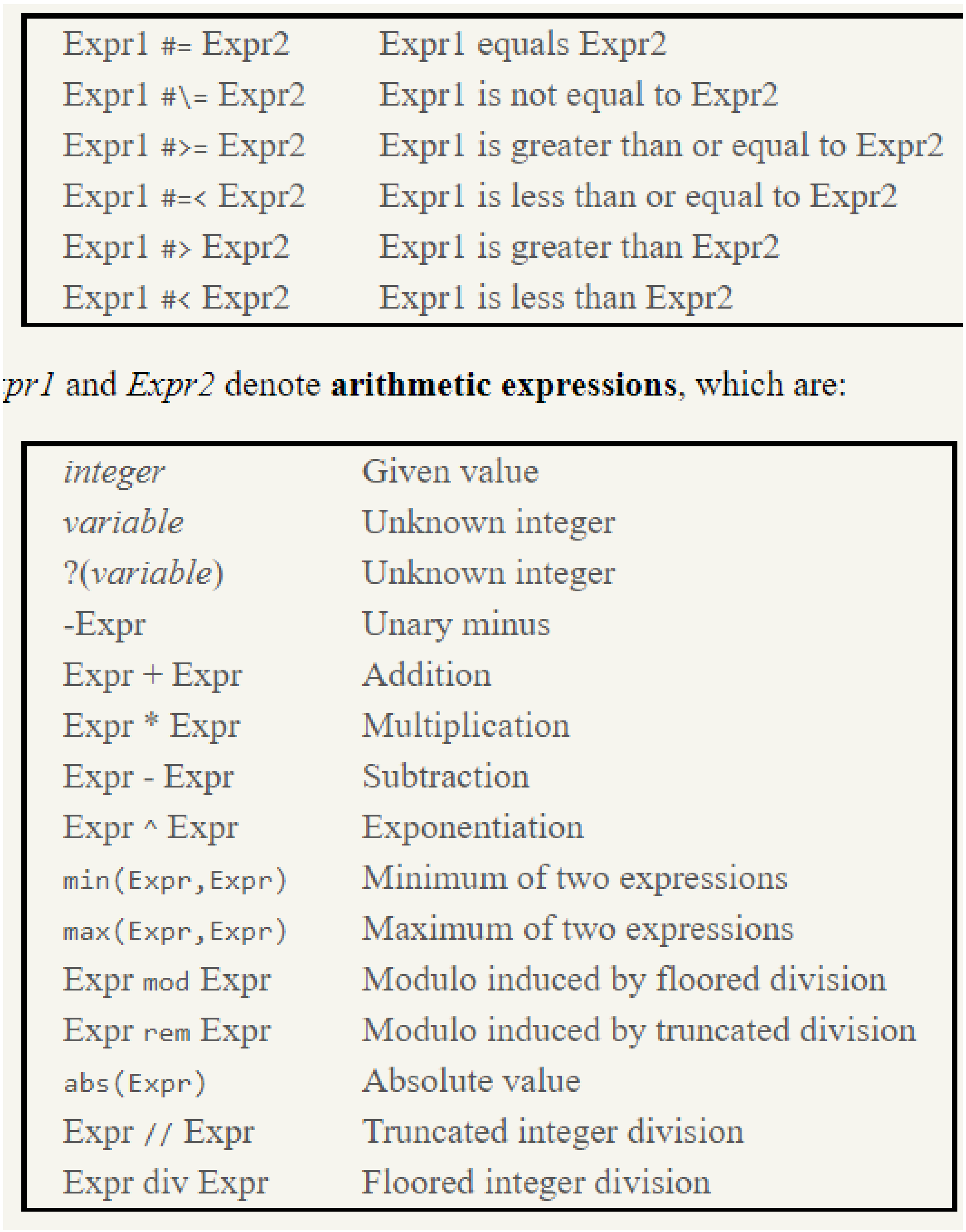
There are many constraints mentioned in the book, such as: arithmetic, membership, enumeration predicates, combination, reification, and reflection predicates. Among these predicates, the **arithmetic constraint** is what we really need to use, and the others are more constraints, so we'd better use the simplest predicate as much as possible at compile time



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)
combinatorial constraints like all_distinct/1 and global_cardinality/2 (section A.9.17.4)
reification predicates such as #<==>/2 (section A.9.17.5)
reflection predicates such as fd_dom/2 (section A.9.17.6)

Many students are accustomed to using very simple language to write programs, so the main purpose of learning CLP (FD) is to enable you to use common primitives in prolog writing in the future, and use low-level code as little as possible.
https://www.metalevel.at/swiclpfd.pdf
If you are interested in this chapter, I do recommend you to read the content inside of the above address. Also, if you have questions about fix your code, you can go to stackoverflow, with the symbol “clpfd” , it’s such a great chance to discuss with some experts in prolog.

Regarding the arithmetic constraints, I think that as long as you look at the use of the official website, you can easily understand it, especially for students who have already learned C++ and have some basics. But I will still write an example to give you an immediate impression



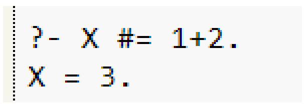
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

pr1 and *Expr2* denote **arithmetic expressions**, which are:

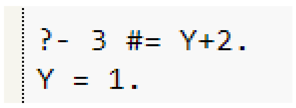
integer	Given value
variable	Unknown integer
?(*variable*)	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
`min(Expr,Expr)`	Minimum of two expressions
`max(Expr,Expr)`	Maximum of two expressions
Expr `mod` Expr	Modulo 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

For example, like integer, variable….etc, I think you already know it from c++ or some similar classes, just read those again to get more familiar with.

Regarding the constraints of declarative arithmetic. In prolog, only one #= is used to express, which is not very different from other programming languages, but the practicality is better, because the coding is easier to understand



```
?- X #= 1+2.
X = 3.
```



```
?- 3 #= Y+2.
Y = 1.
```

, for both of these, even someone who never learned about programming can also understand. So, we use CLP(FD) is to constraint or called replace low level predicates

According to this example, we can see exactly the difference.

```
positive_integer(N) :- N #>= 1.
```

Then we can execute like below:

```
positive_integer(N) :-
        (   integer(N)
        ->  N >= 1
        ;   N #>= 1
        ).
```

I believe many students are familiar with Fibonacci. The simplest sentence is to consider the relationship between natural numbers and factorials. Like 4! = 4*3*2
3!=3*2 This is the simplest expression in mathematics, but in prolog, we will use a slightly more formal method

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N #> 0,
        N1 #= N - 1,
        n_factorial(N1, F1),
        F #= N * F1.
```

If the students are still confused, we can also try to express it in a way like C++

```
?- n_factorial(N, 1).
N = 0 ;
N = 1 ;
false.

?- n_factorial(N, 3).
false.
```

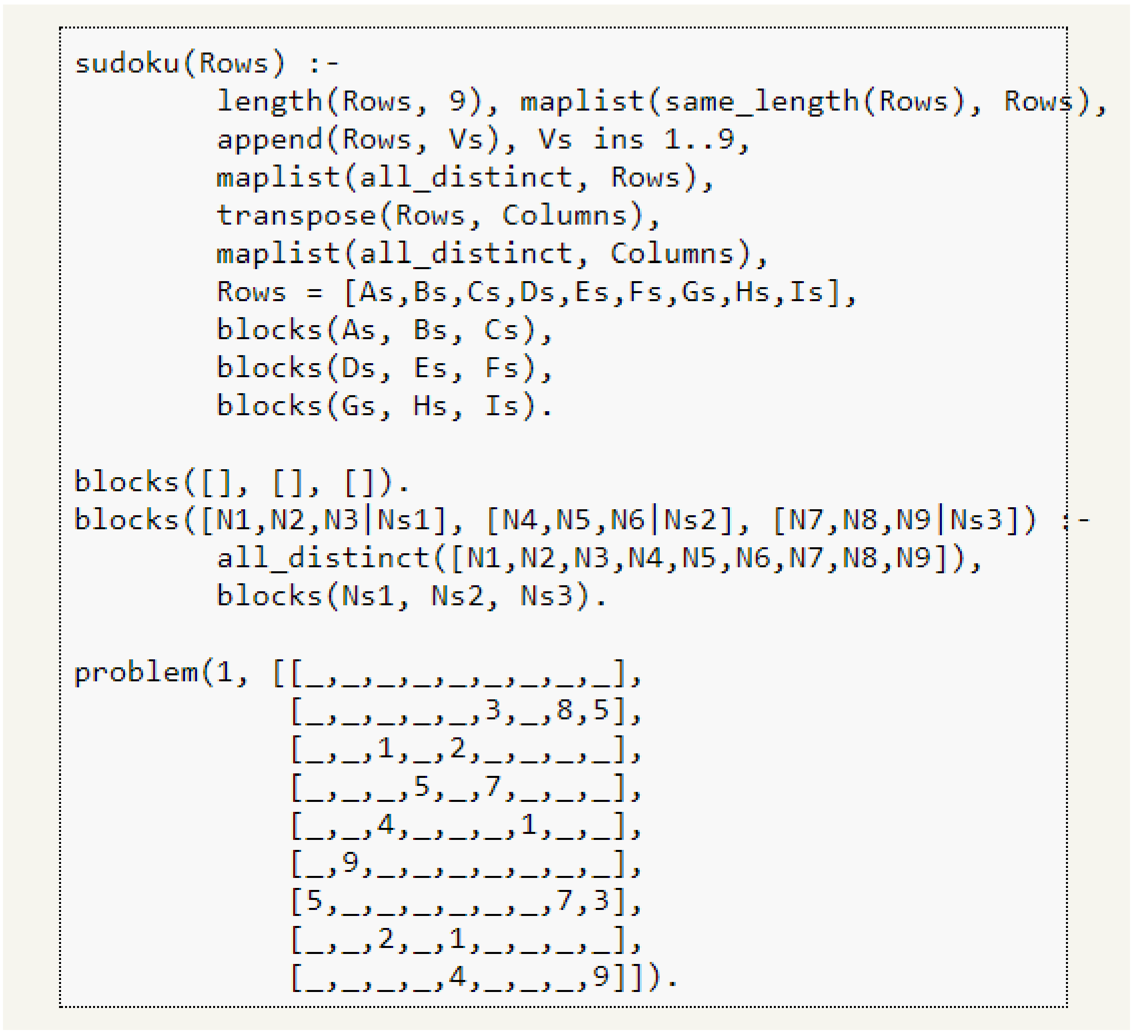
For prolog, we don't have cout, int or something, we can just call this function, then give the number, from N to whatever you want. But in the above example, the first one is instantiated, which causes the second sentence to be mutually exclusive with the first one. Here is another knowledge point: we want to make the predicate deterministic

Combination constraints: used to allocate a series of problems

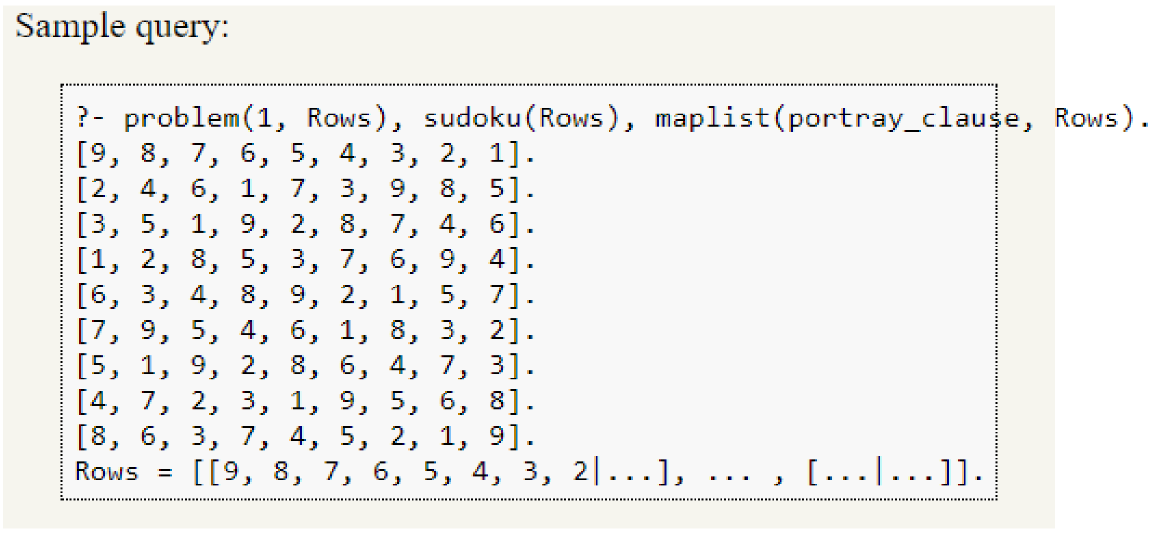
The domain is like the central idea in C++. In this domain, what do we want to write, what effect will be produced, and what result will be achieved.

In CLP, we use the variable domain to integrate all integers and restrict their use. For example, if you take an integer #</5, then in this field, all integers are #</5. And this domain can be adjusted according to your needs at the time

Regarding Sudoku, we are also very familiar with other programming languages. Here we directly look at this code to understand. We know that Sudoku is a 9 x 9 number, so we directly use a simple template like CLP to express.

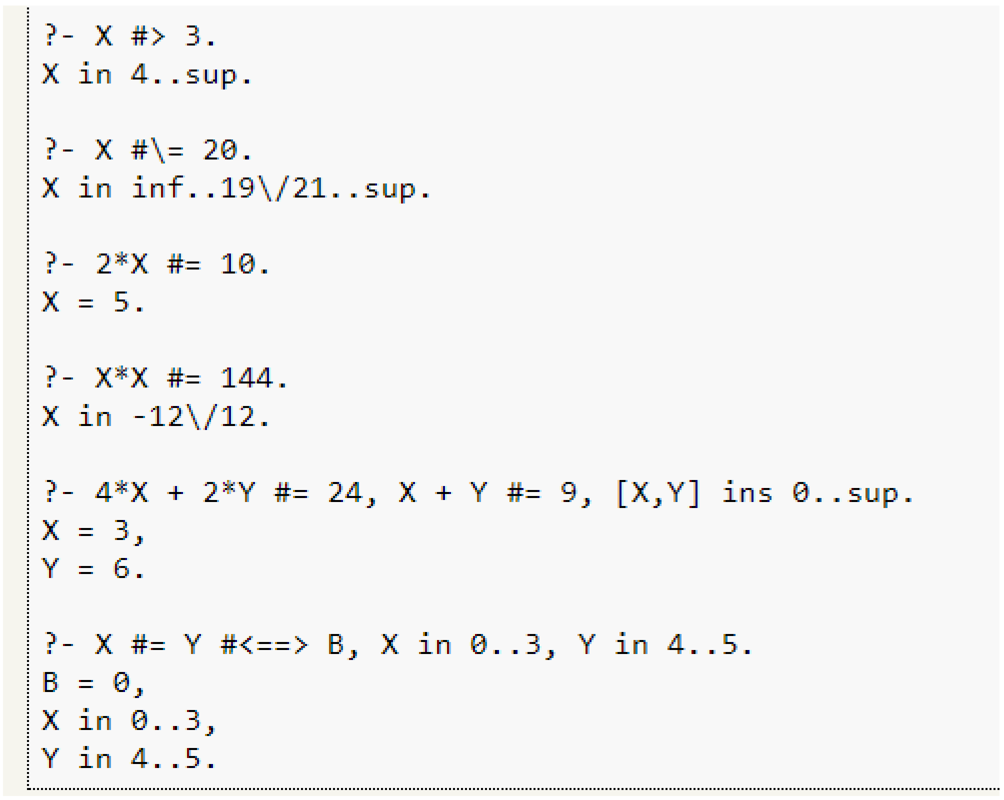


```
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        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, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```


Sample query:

```
?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, 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|...], ... , [...|...]].
```

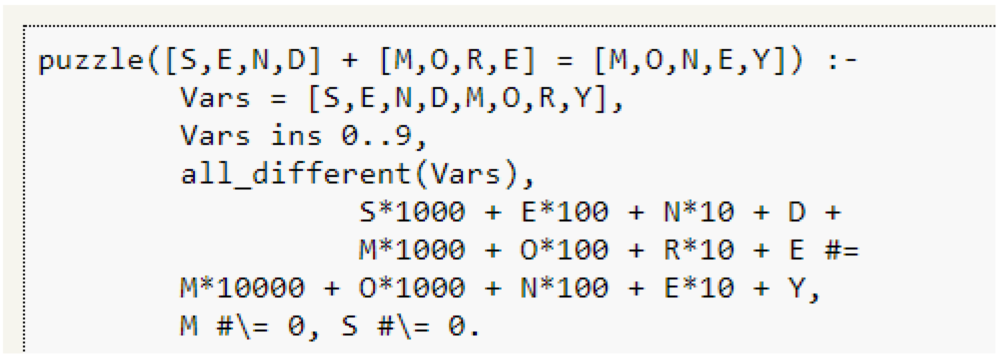
Regarding residual goals, we call the answers inside the residual answers. This is actually like doing a math problem, but we need to understand the meaning of the symbols to easily write it out. When you need a very accurate answer, you also need more constraints on the question



```
?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- X #= Y #<==> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
```

For example 1,2,3,4 we only used one constraint to get an answer, but for question 5, we used 3 constraints to ensure our answer goes the right way.

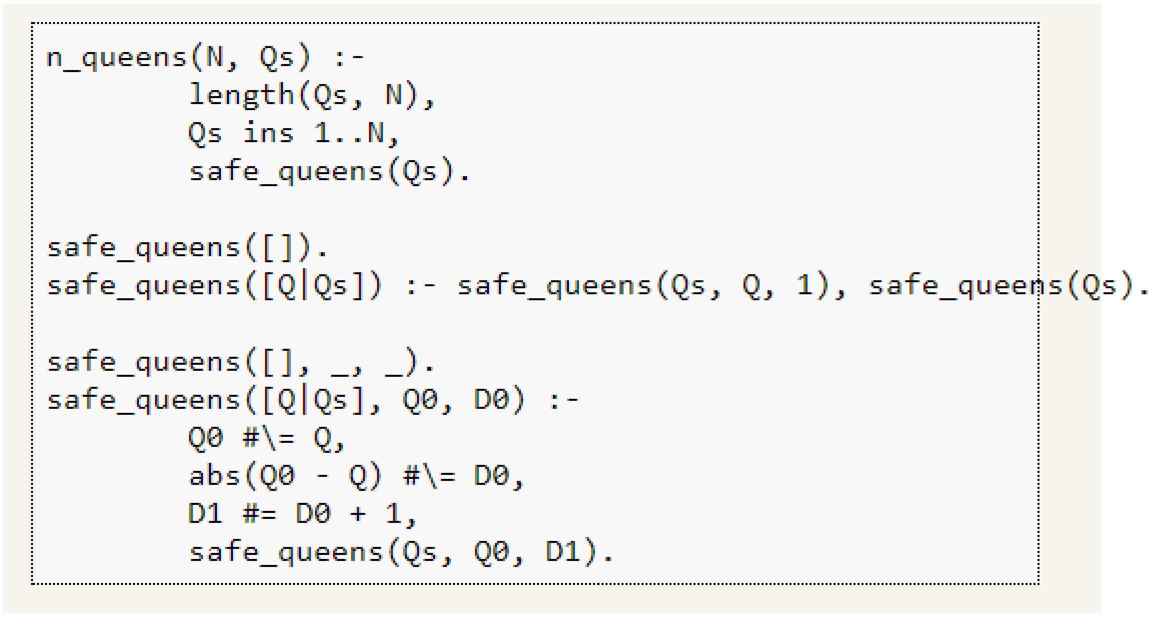
Regarding core relations and search, there are two stages. The first is when all constraints are expressed. Then we use enumeration to find out the specific solution. The central idea: Use a specific predicate to separate the statement from the actual answer, and we can find the answer in more ways



```
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.
```

In the first part we stated the problem and our requirements. Then use different vars to describe different letters to find specific answers. By the way, we have adopted a stricter separation for this issue to ensure that the two parts are completely separated

Regarding the eight queens, the same method is used: CLP is used to constrain, and then the integers are considered. Different constraints are used in different relationships. This is also part of the puzzle. If you are interested, you can think about it during the break, because there are many solutions to this problem. We are here to use an example to show you

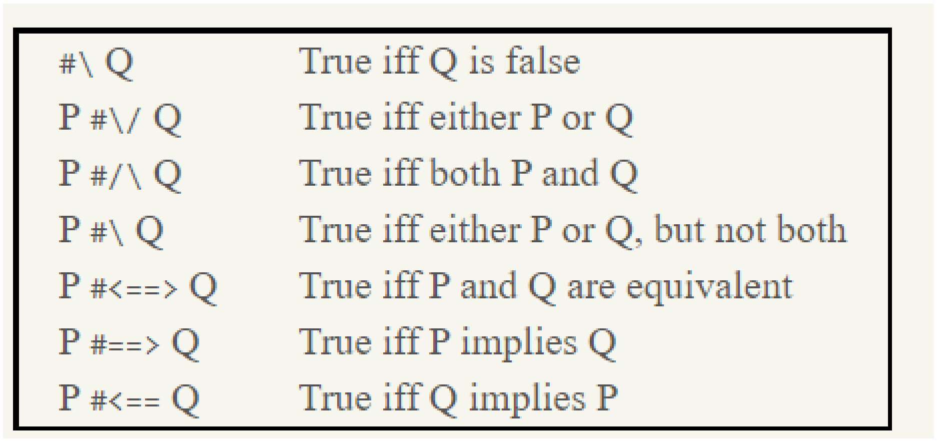


```
n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,
        abs(Q0 - Q) #\= D0,
        D1 #= D0 + 1,
        safe_queens(Qs, Q0, D1).
```

Optimisation

How to optimize has always been one of our most considerations. In prolog, it is easiest to minimize or maximize the value we need through the function of labeling/2. Because it can be solved by incrementing and decrementing. In prolog, we also have a special predicate to constrain to observe its properties and use different labels
In prolog, we can use like: once/1 …etc to optimize our expression.

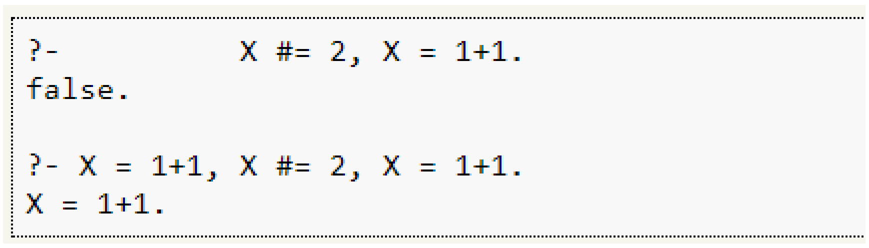
For Reification, as we mentioned before, we have something like:



#\ 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

Those constraints are works for reifiable as well.

Enabling monotonic: usually used by us to add a constraint by default to get some different solutions



```
?-           X #= 2, X = 1+1.
false.

?- X = 1+1, X #= 2, X = 1+1.
X = 1+1.
```

Using **clpfd_monotonic** to set for **True**, it is a method, but there is no new solution.

But we have a new method: custom constraints, we can describe the constraints we want to perform, so as to produce the desired effect

```
:- multifile clpfd:run_propagator/2.

oneground(X, Y, Z) :-
        clpfd:make_propagator(oneground(X, Y, Z), Prop),
        clpfd:init_propagator(X, Prop),
        clpfd:init_propagator(Y, Prop),
        clpfd:trigger_once(Prop).

clpfd:run_propagator(oneground(X, Y, Z), MState) :-
        (   integer(X) -> clpfd:kill(MState), Z = 1
        ;   integer(Y) -> clpfd:kill(MState), Z = 1
        ;   true
        ).
```

In this example, use make-propagator to change this constraint into an internal form, so that the domains of X and Y can be changed. Then use clpfd:trigger_once/1 to provide us with communication opportunities. This predicate is automatically called, when we compile

```
?- oneground(X, Y, Z), Y = 5.
Y = 5,
Z = 1,
X in inf..sup.
```

As we can see, here is an example by using the different constraints.

Regarding arithmetic constraints, the biggest advantage of using CLP (FD) is that the program can be used in many places

Finally, If you are so interested about prolog, I do prefer you read this link below:
http://eu.swi-prolog.org/man/clpfd.html