PROLOG IV MANUAL

Tutorial
Basic concepts
Primitives
Environment

PrologIA

Parc Technologique de Luminy – Case 919
13288 Marseille cedex 09 – FRANCE

PROLOG IV is a PrologIA product, which is a result of a tight collaboration between PrologIA and the *Constraint Programming* team from LIM, the Marseille Computer Science Laboratory, which associates the Université de la Méditerranée, the Université de Provence and the Centre National de la Recherche Scientifique in URA 1789.
The following have been involved in its development: Frédéric BENHAMOU, Pascal BOUVIER, Alain COLMERAUER, Henri GARRETA, Jean-Luc MASSAT, Guy Alain NARBONI, Stéphane N'DONG, Jean-François PIQUE, TOURAÏVANE, Michel VAN CANEGHEM, Eric VÉTILLARD, with the help of Bruno GILLETA, Robert PASERO et Jianyang ZHOU.
Its development has been made possible by studies performed in the realm of two ESPRIT projects: the ACCLAIM project 7195 about concurrent constraint programming, and the PRINCE project 5246, about the environment and constraint extensions to a Prolog dedicated to the industrial and financial world.

For any questions regarding this manual and software, please contact:

PrologIA
Parc Technologique de Luminy – Case 919
13288 Marseille cedex 09 – FRANCE

Tél : +33 (0) 4 91 26 86 36
Fax : +33 (0) 4 91 41 96 37
E-mail: `prologia@prologianet.univ-mrs.fr`

Contents

1

A Prolog IV tutorial

1.1 A first Prolog IV session

Prolog IV is a programming environment. As such, it contains the usual components of such an environment: a compiler and a library. The compiler is used to transform programs into a form which can be used directly by the machine, and the library contains all the predefined code used in Prolog IV. However, Prolog IV differs from many programming languages (like C or C++), by the fact that it includes an interactive mode with which the user can interact directly with the machine.

To start a Prolog IV session, you need either to type the `prolog4` command at your shell prompt, or to click on the Prolog IV icon, depending on the platform that you are using. If you encounter any problems when starting Prolog IV, please contact your local system administrator or refer to the installation guide which has been provided to you together with your Prolog IV kit.

How Prolog IV is Prolog

If Prolog IV starts normally, you will see a banner like the following one (except for the date and version numbers):

```
machine% prolog4

Prolog IV (beta III (204), cp4.8), June 1996 (C) PrologIA 1995,1996
[ Mon Jun 17 10:48:57 MET DST 1996 ]

>>
```

The symbol >> is called the *prompt*. It indicates that Prolog IV awaits an order (which is usually called a *query* in the Prolog world). Let's start by typing the query « write(helloWorld),nl. ».

```
>> write(helloWorld), nl.

helloWorld
true.

>>
```

Of course, you only need to type the query itself (*i.e.* the text which is typeset in bold), and the rest of the text will be displayed by Prolog IV. In this first example, the query is made of two literals, separated by a comma. A *literal* is an atomic Prolog IV command, which may be used to start a program or perform a predefined action. The first literal, `write` is a predefined output command, which may be used at any time during a Prolog IV session to display things. The `nl` literal also is a predefined command, which outputs a carriage return character (`nl` is a shortcut for « new line »). The query ends with a period. After performing the actions requested, Prolog IV outputs `true`, which indicates that the query succeeded (from a logical viewpoint).

After trying a query, let's write a first program. It is made of three rules, which state that Bill and Hillary live in Washington and that Jacques lives in Paris.

```
>> consult.

Consulting ...
   lives(bill,washington).
   lives(hillary,washington).
   lives(jacques,paris).
   end_of_file.
true.

>>
```

The `consult` commamd is one of the commands which can be used to enter a program in the system. The line « `end_of_file` » is used to indicate to the system the end of the program[1]. After that line, the prompt is displayed again.

Each line of the program (more precisely, each sequence terminated by a period) is called a *rule*. The rules of the program are stored in a *rule base*, in the order in which they are entered.

Let's try that program out: does Bill live in Washington?

```
>> lives(bill, washington).

true.

>>
```

In this query, we have used Prolog IV to verify a statement; the next step is to interrogate the rule base with questions like « Where does Bill live? ».

Before to do so, we need to introduce the notions of *identifier* and *variable*. An identifier is a name (*i.e.* a Prolog IV constant). It is a sequence of letters and digits which starts by a lowercase letter[2]. A variable is a sequence of letters and digits which starts by an uppercase letter. To each variable is associated a *domain* of acceptable values at a given time. These values may include identifiers, numbers, and complex constructions using numbers and identifiers, which are called trees.

When a variable is created during the execution of a program or query, its domain is initialised to the set of all values which are valid in Prolog IV. As the program or query runs, this domain reduces until it is reduced to a single value or the program ends.

[1]Under UNIX, it is also possible to type the `Control-D` keys.

[2]This is why we have entered names starting by a lowercase letter in our program; this problem may be avoided by a technique which is described precisely in the chapter about syntax.

Let's now use identifiers and variables to ask questions with one or more unknown values:

Where does Bill live ?

```
>> lives(bill, X).

X = washington.

>>
```

Who lives in Washington ?

```
>> lives(X,washington).

X = bill;

X = hillary.

>>
```

We have obtained two answers to that question, *i.e.* the two possible values for variable `X`. In Prolog IV, the solutions are separated by a semicolon (`;`) and the last solution is followed by a period, just like queries.

Who lives where ?

```
>> lives(X,Y).

X = bill,
Y = washington;

X = hillary,
Y = washington;

X = jacques,
Y = paris.

>>
```

Each answer is a possible assignment of the variables of the query to values which are values with respect to the program.

In order to end a Prolog IV session, you can type the following command :

```
>> halt.

machine%
```

You then return to your operating system's command interpreter. If Prolog IV is used within its graphical environment, you may also click on the « Quit » button in the main button bar.

1.2 More examples, now with constraints

So far, we only have seen a small Prolog IV program (which also is a Prolog program), in which all constraints were equalities (expressed via the unification mechanism). Prolog IV can be used to write much more complex programs, and we will now see a few examples which show how the various Prolog IV objects may be used.

1.2.1 Numerical constraints

The most innovative part of Prolog IV is its handling of numbers, or more precisely, its handling**s** of numbers, since there are two ways to handle them.

When a numerical problem is made exclusively of linear equations, inequations and disequations like $\frac{3}{4}x - \frac{1}{5}y \geq 7z - 1$, the *linear solver* is used, and Prolog IV always finds the exact solution to the problem.

When a numerical problem is made of more complex elements (for instance, using algebraic or trigonometric functions like $x = 2sin^2x$, the *approximated solver* is used, and Prolog IV may not always be able to find the exact solution to the problem.

Let's start by examining the linear solver.

1.2.2 Linear numerical constraints

Let'start by a small example :

```
>> X+Y = 3 , X-Y = 1.

Y = 1,
X = 2.

>>
```

Here, Prolog IV has been used to solve the following system of linear equations:

$$\begin{cases} x + y = 3 \\ x - y = 1 \end{cases}$$

Let's now try to solve a small problem: we have a number of cats and birds, as well as the number of their limbs and heads, and we want to express the relationships which link these values. If we call c the number of cats, b the number of birds, l the number of limbs and h the number of heads, we have the following equations:

$$\begin{cases} h = c + b \\ l = 4c + 2b \end{cases}$$

Both equations are linear, and they can be expressed naturally in Prolog IV:

```
>> H = C+B , L = 4*C+2*B .

L ~ real,
B ~ real,
C ~ real,
H ~ real.

>>
```

Prolog IV answers by a list of constraints which indicate that the domain of each variable is the set of all real numbers. This shows that there exists a solution to that problem (in fact, an infinity of them). However, the system does not have enough information to restrict further the domain of the variables (the system above has four unknowns and two equations). Let's now give some fixed values to two of the variables; for instance, let's say that we have 14 limbs and 5 heads:

```
>> H = C+B , L = 4*C+2*B , L=14 , H=5 .

L = 14,
B = 3,
C = 2,
H = 5.

>>
```

This time, Prolog IV displays the unique solution of the system, by showing the values of the variables for which all constraints of the system are satisfied. Now, before to go on with this example, we will write a rule which expresses the relationship between the variables:

```
>> consult.

Consulting ...
   catsandbirds(C,B,L,H) :- H = C+B , L = 4*C+2*B .
   end_of_file.
true.

>>
```

The meaning of the rule above is as follows: *if the constraints $h = c + b$ and $l = 4c + 2b$ are feasible, then the constraint catsandbirds(c, b, h, l) also is feasible.* We can now use this program by typing the following queries:

```
>> catsandbirds(C,B,14,5).

C = 2,
B = 3.

>> catsandbirds(1,V,L,5).

V = 4,
L = 12.

>> catsandbirds(1,1,5,2).

false.

>> catsandbirds(C,B,6,4).

C = -1,
B = 5.

>>
```

Prolog IV answers `false` to the third query, because the constraint system is not solvable (a limb is missing somewhere). In the fourth query, one of the variables takes a negative value in the answer. It is true that we never specified that the solutions were to be positive. We therefore need to update the definition of the `catsandbirds` relation in order to constrain all four variables to take positive values:

```
>> reconsult.

Reconsulting ...
   catsandbirds(C,B,L,H) :-
       H=C+B , L=4*C+2*B ,
       gelin(C,0), gelin(B,0), gelin(L,0), gelin(H,0).
   end_of_file.
true.

>>
```

The `reconsult` command has been used in order to replace an existing program by a new one. Using the `consult` command instead would have led to an error condition. The constraint `gelin(X,Y)` simply adds the constraint[3] $X \geq Y$. The additional constraints therefore state that all the variables in the rule are positive.

```
>> catsandbirds(C,B,6,4).

false.

>>
```

Prolog IV now answers that there is no solution to that query, as expected. Next, we can ask ourselves what will happen if we give a number of heads and limbs such that the solution is not an integer solution:

```
>> catsandbirds(C,B,7,3).

B = 5/2,
C = 1/2.

>>
```

The answer is half a cat and two and a half birds, which is not an acceptable solution for an animal lover! It is true that we have never specified that the solutions should involve only integer values. With the linear solver, such a constraint is off-limits, since it is not possible to state in a system of linear equations that the solution should be integer. We will therefore stop here for this first example with numerical constraints.

1.2.3 Constraints over real numbers (approximated solver))

Prolog IV includes a set of relations which allow us to build constraints and to reason over real numbers. It is very important to make a distinction between the linear constraints which have been introduced in the previous section and the constraints over reals which are the topic of the present section. This distinction will become evident across the pages.

When reasoning about real numbers, we soon get in trouble. A number as simple as $\sqrt{2}$ cannot be represented exactly in a machine as a number. We can compute as many digits of that number as we wish, and get a much better approximation of it than 1.414, but we will never be able to find all the digits of that number.

[3]Here, `gelin` stands for **g**reater or **e**qual in the **lin**ear solver. This notation looks heavy, but it is required, since the notation $>=$ is reserved by the ISO Prolog standard, and the `ge` symbol is used by the approximated solver.

One alternative is to use a solver which is able to reason formally about mathematical expressions, and therefore to eliminate the representation problem. Such systems exist, but their speed is presently too slow for practical applications. The practical solution therefore lies in making a trade-off between the precision of the computations performed and the performance (speed and memory consumption) of the system.

For all these reasons, the constaints over reals are handled by a solver which has little to do with the solver used to solve linear constraints. This solver over real numbers has the following basic properties:

- *The resolution of* a basic constraint *is* sound *and* complete.
- *The resolution of* a set of constraints *is* sound *but* incomplete.

Here, *incomplete* means that Prolog IV may not be able to detect the fact that a system of constraints has no solution. The *soundness* of the algorithm guarantees, however, that if a system of constraints has at least one solution, then Prolog IV will never infer that the system does not have any solution. We will show later many examples which make these properties very visible!

Incompleteness seems annoying, since it makes it more difficult for us to decide whether or not a constraint system is solvable. In fact, Prolog IV gives us some guarantees concerning the result obtained, which are listed below. We here make the difference between the *exact* problem, which is the original problem (mathematical, physical, ...) and the *approximated* problem, which is its natural translation in Prolog IV.

- It is guaranteed that if an approximated problem is found unsolvable by Prolog IV, then the corresponding exact problem also is unsolvable.
- It is guaranteed that if an exact problem has a solution, then this solution is among the solutions found by Prolog IV when solving the approximated problem.
- It is guaranteed that if Prolog IV finds a solution to the approximated problem in which all variables have a unique value, then this solution is a solution of the exact problem.

Let's now look at a few examples. First, let's try to compute the value of $\sqrt{2}$. There is a relation in Prolog IV which allows us to compute square roots, which we can use here:

```
>> X = sqrt(2).

X ~ cc(`>1.4142135`,`>1.4142136`)

>>
```

The expression `sqrt(2)` is called a *pseudo-term*. A pseudo-term is a term in which appears, at any level, at least one relation symbol whose name and arity$+1$ are reserved[4].

For instance, our example is equivalent to the following one, in which the relational notation is used:

[4] A list of all such reserved symbols, including the symbol **sqrt/2** will be given later on.

```
>> sqrt(X, 2).

X ~ cc('>1.4142135','>1.4142136')

>>
```

Prolog IV answers by giving us the domain of the variable X, which in that case is a set of real numbers. However, such a domain cannot be just any set of real numbers. Only a few subsets of **R** are used in Prolog IV:

- the singletons[5] containing a single rational number (we usually refer to such domains as values).
- the intervals whose bounds can be represented by IEEE single precision floating-point numbers.

We can say a little more about these numbers and intervals. First, we can note that all the IEEE single precision floating-point numbers are rational numbers[6]. Then, there is a finite number of such intervals, since there is a finite number of IEEE rationals.

Back to our example, we know that there exists a solution of the exact problem (this solution is $\sqrt{2}$). According to the principles stated above, we know that this solution is included by the solution proposed by Prolog IV to the approximated problem. That solution was given by Prolog IV by using the relations $\sim$ and cc. The constraint $X \sim E$, where X is a variable, means « X is an element of the set described by the pseudo-term E. » For instance, $X \sim cc(1, 2)$ means that X is an element of the interval $[1, 2]$. Here, cc is a shortcut for « closed–closed, » meaning that the interval is closed on the left and on the right; we also use the letter « o » for open intervals.

The values of the boundaries always are rational numbers, since only these numbers can be expressed with Prolog IV. For instance, the bound `` `>1.4142135` `` represents the smallest IEEE rational which is larger than 1.4142135. In fact, it means that the bound is slightly larger than 1.4142135. Prolog IV could of course give the exact number, but

$11863283/8388608$

just does not remind us at all of the value of $\sqrt{2}$. Similarly, the notation `` `>1.4142136` `` simply denotes the smallest IEEE rational which is larger than 1.4142136. At the end, Prolog IV has given us a very detailed result which is exactly

$$\sqrt{2} \in [1.41421353816986083984375, 1.414213657379150390625],$$

which is the smallest interval with IEEE rational bounds which contains the solution, and that we can interpret as « the value of $\sqrt{2}$ is around 1.4142136. »

However, one of the strengths of Prolog IV is that it is able to compute exact values (which are rational), even when it handles such constraints:

```
>> X = sqrt(4).

X = 2.

>>
```

[5]Singletons are one-element sets.

[6]In the rest of this document, we will refer to these numbers as IEEE rationals.

In that case, the solution is precise, and we are certain that it is a solution of the constraint. The difference with the answer above is that the answer is rational, which allows us to express it. In our first example, the solution was correct, but could not be expressed precisely. Let's now look at the following example:

```
>> X = sqrt(2) , X = sqrt(2.0000001).

X ~ oo(`>1.4142135`,`>1.4142136`).

>>
```

The result is almost the same as previously, and it shows a success, even though it is obvious for us that $\sqrt{2} \neq \sqrt{2.0000001}$. The reason is that the precision given by floating-point intervals does not allow us to distinguish between these two numbers. The relative precision of the single precision IEEE floating-point numbers is about 10^{-6} (*i.e.* 6 significant digits), and since these numbers are used as bounds of Prolog IV's intervals, the numbers above cannot be distinguished. However, the solution above satisfies our basic principles: *« if the exists a solution to the real problem, then it is included in the solution of the approximated problem »*. In this case the solution space is empty, and the principle is trivially satisfied.

Of course, this is an extreme example, and removing one 0 from our second number brings us back to more traditional grounds:

```
>> X = sqrt(2) , X = sqrt(2.000001).

false.

>>
```

This series of examples stresses the fact that when working with the approximated solver, one must never forget that the solver works on the approximated problem, not directly on the real one.

Let's now take a look at the *square* relation, which establishes a relation betwen a number and its square. It is similar to *sqrt*, except from the fact that it will also work with negative numbers. Let's now look at a first example:

```
>> square(4,X), sqrt(Y,4).

Y = 2,
X ~ cc(-2,2).

>>
```

In fact, the solution to the first constraint can be either -2 or 2. The solution given here is the interval $[-2, 2]$, which of course is the smallest interval which includes both values. For the second constraint, there is only one solution, since a square root is always positive (by definition).

Let's now combine these relations:

```
>> square(sqrt(X)) = 2.

X ~ cc('>1.9999998','>2.0000004').

>> sqrt(square(X)) = 2, ge(X,0).

X = 2.

>>
```

The answer to the first question gives a good approximation (in a readable form) of the equation $(\sqrt{X})^2 = 2$, for which the mathematical solution is $X = 2$.

In the second question, we also use the relation *ge*, which is a shortcut for *greater or equal*, and must not be mistaken with the *gelin* relation, which has the same mathematical definition, but is handled by the linear solver. Here, X is constrained to be positive. Here, we can see that, even though the constaint is complex (several relations are used), the solution has only one value, and is therefore exact, according to one of Prolog IV's principles: *If Prolog IV finds a solution to the approximated problem in which all the variables are bound to a single value, then this solution also is a solution of the real problem.*

1.2.4 Why incompleteness?

Let's see in an informal manner how a constraint is handled by the approximated solver:

1. From the constraint, we consider the domains of its arguments. Let's assume that the constraint is based on a relation $\mathcal{R}$, whose arity is n. We can call the arguments' domains $D_1, \ldots, D_n$.
2. The constraint is completely solved, *i.e.* the solver computes the subset E or R^n such that $E = \mathcal{R} \cap D_1 \times \cdots \times D_n$. We will assume that E is not empty (*i.e.* we don't get a failure).
3. Next, we approximate that set E by the smallest block[7] which contains it. Each side of our block will be very close to at least one point of the set E.
4. The block obtained is a cartesian product of intervals of R. The final step is to approximate these intervals into IEEE intervals which can be represented in Prolog IV. This approximation is obtained by extending slightly the intervals to the closest IEEE floating-point numbers available. The approximation obtained always is equal or larger to the original block.
5. Each dimension of the block is represented by an interval, which can then be assigned to a variable. A fundamental property of the resolution mechanism is that it guarantees that the new domains $D'_1, \ldots D'_n$ are smaller than or equal to the original domains $D_1, \ldots, D_n$.

To be complete, one must add that a specific mechanism is then used in order to handle properly the possible exact solutions to a problem. If sufficiently many arguments are bound to rational numbers, then the (exact) values of the other arguments can sometimes be computed.

[7] A block is a rectangle in 2 dimensions, a parallepiped in three dimensions, ...

The resolution of a *set* of constraints is done very simply by handling each constraint as shown above. The variables are the only interfaces between the various constraint, which means that the interface is only done through approximated sets. Because of this, some significant precision loss may occur in the handling of such sets. An extreme case occurs when the overevaluation of two disjoint sets gives them a non-empty intersection. Let's look at a small example:

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>> X = oo(-1,1), Y = oo(-1,1), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>>
```

For both queries, the answer gives us back the subdomains for X and Y unchanged. Of course, this is correct, since whatever number we take in the interval $(-1, 1)$, we can always find another number in that interval which is strictly greater (resp. lower) than the first one. So far, the answers are the ones expected. Trouble starts when we combine these queries:

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).
```

We get a success, even though the real problem expressed here has no solution. Of course, our problem comes from the combination of the two comparison constraints. Since each one is handled separately, they communicate through the subdomains of their arguments, which never « shrink. » This leads us directly to success.

This weakness in the communication between constraints (which can also be seen as a weakness in the approximation used) is the reason for the incompleteness of Prolog IV's approximated solver.

Notes:

- The domain declaration constraints (like *c*, for instance) usually do not cause problems, since all their arguments except one are known numbers.

- Another reason for incompleteness happens when the domain of a variable must be enlarged in order to become an IEEE interval. We already have seen an example of that with the confusion between $\sqrt{2}$ and $\sqrt{2.0000001}$.

- A constraint of the form $r(X, X)$ is in fact the conjunction of constraints $\exists A\ r(X, A), X = A$, which contains *two* constraints and is therefore subject ot the problems shown above.

1.2.5 Unions of intervals

Next, let's focus on the two possible modes in which the approximated solver may work. In the mode that we have been using so far, we may have the following:

```
>> X = sqrt(2).

X ~ cc(`>1.4142135`,`>1.4142136`).

>> square(X) = 2.

X ~ cc(-`>1.4142136`,`>1.4142136`)

>>
```

The second query return quite a large domain, simply because this is the smallest domain in which both solutions of the constraint fit (these solutions are $-\sqrt{2}$ and $\sqrt{2}$.

We can now use another mode, which is called the *unions of intervals* mode. This is done by typing the first query below. We can then try out another example:

```
>> set_prolog_flag(interval_mode, union).

true.

>> sqrt(square(X)) = 2.

X ~ -2 u 2.

>>
```

Here, the solution is far more precise, since it indicates that the solution may be either -2 or 2, which is much better than the interval $[-2, 2]$. This is indicated by the pseudo-term t_1 u t_2, which denotes the set of all trees which belong to t_1 or t_2 (here the u stands for *union*).

We can also ask the same question as above:

```
>> square(X) = 2.

X ~ cc(-`>1.4142136`,-`>1.4142135`)u cc(`>1.4142135`,`>1.4142136`).

>>
```

We here obtain the union of two small intervals, which are respectively around $-\sqrt{2}$ and $\sqrt{2}$, which is a nice improvement.

We have seen in these two simple examples that the unions of intervals mode leads to a much greater precision in the answers. Of course, this has a price, which is a significant loss in the performance of the solver. This loss is due to the increased complexity of the data structures used in the algorithm. In practical programs, this mode is in fact seldom used, because of this drawback.

To come back in the default mode (which is called the *simple intervals* mode) is as follows:

```
>> set_prolog_flag(interval_mode, union).

true.

>>
```

1.2.6 Existential quantifiers

The existential quantifier is used to build constraints of the form $X\ ex\ P$ in which X is an anonymous (or existential) variable and P a constraint (or a conjunction of constraints like a query). Such a constraint reads *« There exists X such that P »* and it has the same truth value as P. However, the variable X is left unconstrained. This construct is not part of the ISO standard.

More practically, this construct indicates to Prolog IV that the variable X is not interesting for the programmer, and that its value or domain should not be displayed in the solution:

```
>> X = Y.

X = Y,
Y ~ tree.

>> X ex X = Y.

Y ~ tree.

>> X ex X = f(g(2),g(2)).

true.

>>
```

In the answer to the second query, no domain is displayed for X. In the third one, Prolog IV simply answers that the system is solvable, without giving a solution.

Existential variables may also be used as « local variables, » disconnected from the context, which will be free outside of the ex construct.

In the following query, the variable X has two different roles, including one as a local variable in the bracketed constraint. There is therefore no problem for x to take the values 1 and 3, since these instances of X are totally disconnected:

```
>> X = Y, (X ex X = 1), Y = 3.

Y = 3,
X = 3.

>>
```

Of course, such a use of existential variables is strongly discouraged, because of the confusion that it generates. To conclude, here are three examples which show some more likely uses of existential variables:

```
>> X ex Y = f(X,X)

A ex
Y = f(A,A),
A ~ tree.

>> X ex Y = f(X).

Y ~ f(tree).

>> X ex Y ex Z = f(X,Y).

Z ~ f(tree,tree).

>>
```

In the first query, we create a constraint which includes two occurrences of the same existential variable. Since the identity of the two subtrees is significant, an existential variable also appears in the answer. One may note that the name of this variable is not the same as in the query. This is normal, since the name of existential variables is not significant.

On the opposite, in the second query, the existential variable does not appear in the answer, because it only has one occurrence. It can therefore be replaced by its domain (*i.e. tree*, which represents the set of all trees).

The third query shows how several quantifiers can be used in the same query, in order to have several existential variables. The query reads: « There exists an X such that there exists an Y such that Z equals $f(X, Y)$. » One may also note that the associativity is from right to left.

1.2.7 Are things *compatible* or *equal* ?

When should we use « $\sim$ » or « $=$ » in constraints ?

The two relations $\sim$ (compatible) and $=$ (equal) are very similar, and one must admit that they seem to perform the same action. The main reason is that when the programmer abusively uses $=$ instead of $\sim$, Prolog IV is able to make the changes by itself. On the other hand, Prolog IV always makes a proper use of these relations when it outputs solutions; it uses $=$ as often as possible, and $\sim$ whenever necessary.

Now, let's get to the explanation. $=$ is a standard relation, which denotes the equality between terms (*i.e.* the Prolog IV extension of Prolog's unification). In particular, it should not be used with pseudo-terms as arguments. On the other hand, $\sim$ has been designed to work with pseudo-terms, and it works in the following way (where t and t' represent terms and p and p' represent pesuo-terms):

- $t = t'$ is the equality (it is equivalent to $t = t'$).
- $t = p$ is true if the item represented by t is a member of the set represented by p.
- $p = t$ is true if the item represented by t is a member of the set represented by p, and false otherwise.
- $p = p'$ is true if the intersection of the sets represented by p and p' is not empty, and false otherwise.

1.2.8 Anonymous variables

Anonymous variables are variables which are written with the single character « _ ». Each occurrence of that symbol represents a different variable. Such variables simply are existential variables that one does not need to name or quantify.

```
>> X = f(_,_).

X ~ f(tree,tree).

>>
```

We see in that example that, even though the same symbol is used twice, the answer refers to two independant subtrees.

1.2.9 Relations and pseudo-terms

In the examples above, we have used indifferently some mathematical examples in functional form (pseudo-term) and in relational form (literal). In fact, all of Prolog IV's basic constraints can be noted in both forms; the only difference then is the syntactical position, and of course, the arity.

The relationship between these two concepts (and denotations) is shown below:

$$r(X_0, X_1, \ldots, X_n) \ \equiv\ X_0 \sim r(X_1, \ldots, X_n)$$

We see immediately that the argument which is used as « result » of the operation is the first argument of the relation. The order of the other arguments is left unchanged. If we apply this equivalence to a constraint, we get for instance: $ge(X, 0) \ \equiv\ X \sim ge(0)$. The pseudo-term $ge(0)$ denotes the set of all real numbers which are greater or equal to 0 (*i.e.* the positive numbers).

The « result » is here positioned as the first argument, which may be quite a shock to Prolog aficionados, since the use in Prolog is to have the leftmost arguments be the input arguments and the rightmost ones being the output arguments. The choice we have taken may be explained by a few arguments:

- What we call « result » is not really a result, but only one of the arguments of a relation. One has to recall that there are no functions in Prolog IV's constraints, only relations.
- Relations are seldom used as such, and more often as pseudo-terms, which are often easier to read[8].
- The « result » argument should always be at the same place (for consistency reasons). by looking at some of Prolog IV's « true » relations, like for instance the comparison relation[9], one may see that it is far more natural for these to consider the first argument as being the result.

1.2.10 Powerful, surprising and beautiful queries

In the following, we will not show programs, but just a few queries, without defining a rule base. Unless otherwise indicated, we will assume that we work

[8]This is particularly true with relations which are a strong functional flavor, such as $plus$ or cos.

[9]See the ge example just above.

in the unions of intervals mode.

Let's start by computing the set of all X whose square is strictly greater than 1:

```
>> gt(square(X),1).

X ~ lt(-1)u gt(1).
```

The result is the interval $(-\infty, -1) \cup (1, +\infty)$, which is expressed as the union of the set of all numbers strictly smaller than -1 ($lt(-1)$) and of the set of all numbers strictly greater than 1 ($gt(1)$).

An elegant deduction

If Y is a square ...

```
>> X ex square(X) = Y.

Y ~ ge(0).
```

... then it must be positive or zero !

Note the use of an existential quantifier in order not to see the variable X in the answer.

Some queries over integers

We here introduce the pseudo-term $X\,nY$ in which n is supposed to recall the intersection symbol ($\cap$). It constructs the set whose elements belong to one of its arguments. The pseudo-term int denotes the set of all integers. The next query asks for the elements of the interval $[1, 9]$ which are integers:

```
>> X ~ cc(1,9) n int.

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9.
```

We obtain the list of solutions because we are in unions mode[10]. If you want to get the list of all integers (*i.e.* if you don't put the cc constraint), you will of course be in trouble, which is thoroughly explained in the Problems section below.

Let's now look at two queries which compute the sets of even and odd numbers between 1 and 9:

```
>> X  ~  cc(1,9) n times(2,int).

X ~ 2 u 4 u 6 u 8.

>> X  ~  cc(1,9) n int n times(2,nint).

X ~ 1 u 3 u 5 u 7 u 9.
```

The pseudo-term $nint$ denotes the set of all real numbers which are not integers. In the example above, we have therefore defined an even number as the double of an integer, and an odd number as an integer number which is the double of a non-integer number. The second definition is slightly tricky, but fits very well the definition of an odd number.

[10] In simple intervals mode, we just obtain the domain $cc(1, 9)$ for X, since there cannot be « holes » in such an interval.

Now, we can compute the set of all prime numbers between 10 and 100. Those are all the integers which are multiple or none of the prime numbers which are smaller or equal to $\sqrt{100} = 10$, *i.e.* 2,3,5 and 7.

```
>> X  ~  cc(10,100) n int n times(2,nint) n times(3,nint)
            n times(5,nint) n times(7,nint).

X ~ 11 u 13 u 17 u 19 u 23 u 29 u 31 u 37 u 41 u 43 u 47 u 53 u 59 u 61
u 67 u 71 u 73 u 79 u 83 u 89 u 97
```

Rational vs. irrational numbers

The pseudo-term pi represent the set π. It is therefore a singleton, which only contains an irrational number. Since Prolog IV is not able to represent irrational numbers, we only get an approximation of that set, which is the smallest floating-point interval which contains the value π:

```
>> X ~ pi.

X ~ oo(`>3.1415925`,`>3.1415927`).
```

However, don't think that Prolog IV is stupid enough to let you give a value for π and accept it. Let's try it with a reasonably precise number (yes, those are the 60 first decimals of π):

```
>> pi ~ 3.141592653589793238462643383279502884197169399375105820974944.

false.
```

This is expected, since we all know that the number above is not π. It is close to it, but it is not it. On the opposite, we get a success as an answer to the following query:

```
>> oo(`>3.1415925`,`>3.1415927`) ~
        3.141592653589793238462643383279502884197169399375105820974944.

true.
```

The difference between those two queries is the fact that when an relation argument becomes a constant, some additional checks are performed by Prolog IV. In the case of the relation pi, this check is very simple, and invariably leads to a failure: All numbers which can be represented in Prolog IV are rational numbers, and π is not; we therefore immediately know that any constant provided cannot by π.

Trigonometry and cosines

We want to know which are the integer cosines of any angle whose measure is between 1 radian and 100 radians. This can be translated by:

```
>> X ex X ~ cc(1,100), Y ~ cos(X) n int.

Y ~ -1 u 0 u 1
```

We could also write the following query :

« `Y ~ cos(cc(1,100)) n int` ».

Which are the x *such that* $\cos x = 1$, *with* $-10 \leq x \leq 10$ *?*

```
>> X ~ cc(-10,10), cos(X) ~ 1.

X ~ oo(-'>6.2831854',-'>6.283185')u 0 u oo('>6.283185','>6.2831854')
```

The answer may be translated in mathematical language by the set:

$$]-(2\pi+\varepsilon),\ -(2\pi-\varepsilon)[\cup\{0\}\cup]2\pi-\varepsilon,\ 2\pi+\varepsilon[$$

We here recognize the approximation of the set $\{-2\pi, 0, 2\pi\}$.

There are of course a number of other trigonometric functions. Among them, we have sin, tan, cot, and their reciprocal functions $arcsin$, $arccos$, $arctan$. We also have the hyperbolic functions $cosh$, $sinh$, $tanh$ et $coth$.

In order to terminate this list of transcendental functions, one must cite exp, ln (natural logarithm) and log (decimal logarithm).

1.2.11 Algebraic functions and more

We will now describe a few functions and pseudo-operations over real numbers. The four classical operations over real numbers ($+$, $-$, $*$, $/$) are implemented as four relations over trees ($plus$, $minus$, $times$, div). The unary plus and minus are implemented by $uplus$ and $uminus$. The pseudo-term notation allow us to look at these relations as pseudo-operations over real numbers.

These basic functions have also been associated to shortcuts. For practical reasons, some of then have the same chortcut (with a different arity, of course):

- The symbol `.+.`[11] is a shortcut for the relations $plus$ and the relation $uplus$. The arities (number of arguments) of these relations are different; $plus$ takes three arguments (two numbers and their sum), while $uplus$ only takes two. The relations are therefore distinguished through their arity.
- The symbol `.-.` is a synonym of both $minus$ and $uminus$.
- The symbol `.*.` is a synonym of $times$.
- The symbol `./.` is a synonym of div.

Furthermore, these shortcuts are all defined as being operators, which allows us to use them syntactically as operators in pseudo-terms.

Here is a query which uses some of these shortcuts :

```
>> X ~ cc(-2,2), Y ~ cc(-1,1), Z ~ .-.sqrt(X.+.Z)./.2 .+. 3.*.Y .

Z ~ cc(-2,3),
Y ~ cc(-'<0.6666667',1),
X ~ cc(-2,2).
```

[11] The periods around the addition sign are used to distinguish this symbol from the symbol +, which is used by the linear solver.

There are other relations over reals which can be used in pseudo-terms. For instance, $floor$ and $ceil$ « return » the largest (resp. smallest) integer which is smaller (resp. larger) than their other argument. We also have abs to compute absolute values, as well as min and max to compute the minimum (resp. maximum) value of two numbers.

1.2.12 A few words about the reduction of numerical domains

Let's look a little more closely at the way in which the domains of variables are reduced, by examining a small example involving a simple sum. Let's consider the equation $z = x + y$, with $x \in [1, 5]$ and $y \in [-1, 8]$.

Which is the domain z ?

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8).

Y ~ cc(-1,8),
X ~ cc(1,5),
Z ~ cc(0,13).
```

The answer is that $z \in [0, 13]$, and the domains of the other variables are unchanged.

Same question, with an additional constraint which forces z to belong to the interval $[-1, 1]$:

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8), Z ~ cc(0,1).

Y ~ cc(-1,0),
X ~ cc(1,2),
Z ~ cc(0,1).
```

Here, the domains of all variables are reduced. This is very important, since it shows that Prolog IV does not simply « compute » operations, but is also able to reduce the domain of an operation's arguments when some information is known about the result. This is why the basic mathematical items in Prolog IV are relations, not functions.

Now, let's try to force z to be negative (or 0):

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8), Z = le(0).

Y = -1,
X = 1,
Z = 0.
```

Here, the domains are all reduced to a single value for all variables. According to one of our basic principles, we now are able ensure that this solution is actually a true solution of the problem stated.

Note:

- The order of constraints is not significant.

1.2.13 Shortcuts and operators

Here is a table which summarizes all the shortcuts which are available in Prolog IV, as far as pseudo-terms are concerned. These shortcuts are all infix or prefix operators.

Pseudo-terms	Shortcut
$uplus(t)$	`.+.` t
$uminus(t)$	`.-.` t
$plus(t_1, t_2)$	t_1 `.+.` t_2
$minus(t_1, t_2)$	t_1 `.-.` t_2
$times(t_1, t_2)$	t_1 `.*.` t_2
$div(t_1, t_2)$	t_1 `./.` t_2
$upluslin(t)$	$+\, t$
$uminuslin(t)$	$-\, t$
$pluslin(t_1, t_2)$	$t_1 + t_2$
$minuslin(t_1, t_2)$	$t_1 - t_2$
$timeslin(t_1, t_2)$	t_1 `*` t_2
$divlin(t_1, t_2)$	t_1 `/` t_2
$conc(t_1, t_2)$	t_1 o t_2
$n(t_1, t_2)$	t_1 n t_2
$u(t_1, t_2)$	t_1 u t_2
$index(t_1, t_2)$	t_1 `:` t_2

There are a few important facts to know about the priority of these operators:

- Unary operators have priority over the binary operators.
- multiplicative operators (like `.*.` or `./.`) have priority over the additive ones.
- The associativity of these operators is left-to-right, *i.e.* the expression $t_1\ op\ t_2\ op\ t_3$ is interpreted as $(t_1\ op\ t_2)\ op\ t_3$.
- The intersection (n) has priority over the union (u).

1.2.14 Enumeration

We will now take a look at the enumeration of integer values, through the computation of pythagorician triangles, *i.e.* triangles with a square angle whose sides have integer sizes. In order to limit the number of solutions, we will voluntarily limit the sizes of the triangles sides integers between 1 and 10. Let's state that X, Y and Z belong to $[1, 10] \cap \mathbb{N}$, and add the constraint $Z^2 = X^2 + Y^2$. We will first use the simple intervals mode:

```
>> set_prolog_flag(interval_mode, simple).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ cc(2,10),
Y ~ cc(1,9),
X ~ cc(1,9).
```

And once again, in the unions of intervals mode:

```
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8.
```

In both cases, we obtain a unique solution which contains the triplets that we are interested in. With simple intervals, we get a block of R^3, and with unions, we get a cartesian product of discrete values. One can note that X and Y have symmetrical roles in both cases, and therefore have the same domain.

Now, let's write a small predicate which will allow us to have shorter queries:

```
>> consult.

Consulting ...
   side(X,B) :- X ~ cc(1,B) n int.
   end_of_file.
true.

>> side(X,10), side(Y,10), side(Z,10),
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8

>>
```

In order to get the exact solutions, we need to restrict the domain of some variables in order to help the constraint propagation. For this, we can use the predefined primitive `intsplit`, which takes as argument a list of variables, and tries all possible integer values in the domains of the variables in the list. By backtracking, we then obtain all possible solutions:

```
>> X = side(X,10), side(Y,10), side(Z,10),
      square(Z) ~ square(X).+.square(Y),
      intsplit([X,Y,Z]).

Z = 5,
Y = 4,
X = 3;

Z = 5,
Y = 3,
X = 4;

Z = 10,
Y = 8,
X = 6;

Z = 10,
Y = 6,
X = 8.

>>
```

Here, we have put all the variables in the list. This is not necessary, because we know that the value of Z can be computed as soon as we know the values of X and Y. We therefore obtain the same result with the following query:

```
>> X = side(X,10), side(Y,10), side(Z,10),
      square(Z) ~ square(X).+.square(Y),
      intsplit([X,Y]).

Z = 5,
Y = 4,
X = 3;

Z = 5,
Y = 3,
X = 4;

Z = 10,
Y = 8,
X = 6;

Z = 10,
Y = 6,
X = 8.

>>
```

Now what happens when we only put Z in the list to be split?

```
>> X = side(X,10), side(Y,10), side(Z,10),
      square(Z) ~ square(X).+.square(Y),
      intsplit([Z]).

Z = 5,
Y = 3 u 4,
X = 3 u 4;

Z = 10,
Y = 6 u 8,
X = 6 u 8.

>>
```

This time, some precision has been lost, because all the symmetrical solutions are grouped together. However, the result is more synthetic if we are only interested in the size of the largest side of the triangle. Finally, an important note is that since the domains of X and Y are not reduced to a single value, Prolog IV does not guarantee here that the solutions actually are true solutions to the real problem. One way to improve that is to enforce an order between the values of X and Y, which will remove some symmetry from the problem. Here we add the constraint $gt(X, Y)$, which states that $X > Y$:

```
>> X = side(X,10), side(Y,10), side(Z,10),
      square(Z) ~ square(X).+.square(Y), gt(X,Y),
      intsplit([Z]).

Z = 5,
Y = 3,
X = 4;

Z = 10,
Y = 6,
X = 8.

>>
```

We could keep on going with these examples for a long time. If you want to keep playing, you can try to change the maximal sizes of the sides, and you will find more interesting results.

1.2.15 Partitioning (or how to enumerate real numbers)

Let's try to find the (numerical) solutions to the equation $x = 2\cos x$. In Prolog IV, we simply write:

```
>> X = 2.*.cos(X).

X ~ oc(-`<0.8322952`,2).
```

This result means that if there are any roots to that equation, these roots are in the interval $(-0.8322952, 2]$. It is a valuable result, but we would definitely like to obtain something more precise. One solution is to partition the domain of X, for instance in two sub-intervals, and to check in both sub-intervals if the constraints are still satisfied, and what their effect is. Let's for instance try it by splitting the interval $(-0.8322952, 2]$ at 0:

```
>> X ~ co(-1,0), X = 2.*.cos(X).

false.

>> X ~ cc(0,2), X = 2.*.cos(X).

X ~ cc(`>1.0298662`,`<1.029867`).
```

We obtain two very interesting results. In the first query, we find out that in fact, there is no solution in the interval $[-1, 0]$. In the second query, we learn that the solution is very close to 1.029867, which is about as good a result we can get using single precision floating-point numbers. This result is very interesting, because we here notice that splitting the interval once has been sufficient to get to the solution that we were looking for.

Let's now try a very close problem, which is to solve the equation $x = 2\sin x$:

```
>> X = 2 .*. sin(X).

X ~ cc(-2,2).

>>
```

Like before, we obtain an interesting result, which was quite expected (since $sinx$ belongs to $[-1, 1]$ by definition). We will now try to refine that result by splitting this interval in two parts:

```
>> X ~ cc(-2,0), X = 2 .*. sin(X).

X ~ cc(-2,0).

>> X ~ cc(0,2), X = 2 .*. sin(X).

X ~ cc(0,2).

>>
```

This time, it does not work as well. Of course, we can continue the splitting:

```
>> X ~ cc(1,2), X = 2 .*. sin(X).

X ~ oo(`<1.895494`,`>1.8954945`).

>> X ~ cc(0,1), X = 2 .*. sin(X).

X ~ co(0,`<2.384186e-7`).

>> X ~ cc(-1,0), X = 2 .*. sin(X).

X ~ oc(-`>3.5762786e-7`,0).

>> X ~ cc(-2,-1), X = 2 .*. sin(X).

X ~ oo(-`>1.8954951`,-`>1.8954933`).

>>
```

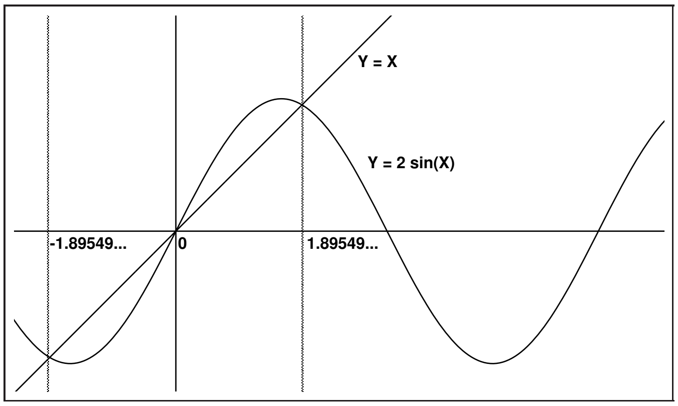
This time, it works, and we obtain a solution for each sub-interval. It is interesting to note that for both intervals which have 0 as a bound, the solution is very close to 0, which leads us to believe that the solution actually is 0. Of course, this is not given by Prolog IV, and there can perfectly be two solutions, with very small absolute values.

With this equation, we have splitted the interval twice before to get a result. With other problems, we may have to split the interval many more times before to get an interesting result. This is why there is in Prolog IV a primitive which is used to split intervals, called `realsplit`. Let's try it with our last problem:

```
>> X = 2 .*. sin(X), realsplit([X]).

X ~ oo(-`>1.8954951`,-`>1.8954933`);

X ~ oc(-`>3.5762786e-7`,0);

X ~ oo(0,`<2.384186e-7`);

X ~ oo(`<1.895494`,`>1.8954945`).

>>
```

Of course, this primitive is guaranteed to work, because there it can only split intervals with floating-point bounds, and there are a finite number of such intervals. However, even though the computation time is guaranteed to be finite, nothing says that it should be short. If the list of variables to be split is long, or if the constraints are very complex, it can take quite some time to obtain a solution, or all the solutions. You will find out that this time often exceeds the limits of human patience.

To close that example, let's have a look at the graphical representation of the problem.



We find out that the intersection of the functions $y = x$ and $y = 2\sin x$ contains three points, which confirms our initial hypothesis. More mathematically, there is a way to confirm our solutions in a precise manner. Our equation is exactly like finding the zeros of the function $y = 2\sin x - x$. We have found the only three intervals in which this function can have a zero value. Mathematically, we know that there is a root in such an interval if

- The sign of the $2\sin x - x$ is different on the bounds of the interval.
- The derivative of $2\sin x - x$ does not have a zero in the interval.

This derivative is $2\cos x - 1$. Let's use Prolog IV to confirm one of the solution:

```
>> X = `<1.895494`, Y = 2.*.sin(X) .-. X .

X = 3975139/2097152,
Y ~ oo(`>3.5762786e-7`,`>5.960464e-7`).

>> X = `>1.8954945`, Y = 2.*.sin(X) .-. X .

X = 15900561/8388608,
Y ~ oo(-`>5.960464e-7`,-`<4.768372e-7`).

>> X ~ oo(`<1.895494`,`>1.8954945`), 2.*.cos(X) .-. 1 = 0 .

false.

>>
```

We now are sure that there exists a solution in this interval. We could perform exactly the same reasoning for the other solutions to our problem. As a conclusion, we can say that Prolog IV is a good tool to find possible solutions to equations and problems, but that it is not a formal tool, and that some human reasoning is very often required in order to confirm the existence of solutions.

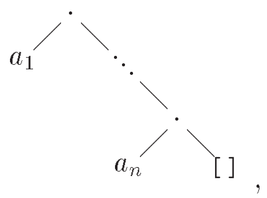
1.2.16 Lists and trees

The domain of Prolog IV is the set of all trees whose nodes are labeled

- either by a number, in which case the node has no children,
- or by an identifier, in which case the node has zero or more children.

We call *leaf* a tree without children. Another way to characterize such trees is to say that they are reduced to a single node.

A *list* is a tree a of the form



where the a_i's are trees. The list above is denoted $[a_1, \ldots, a_n]$, and we denote by $|a|$ the *length* n of the list a (this length may be zero if the list is empty). We also define the concatenation of two lists:

$$[a_1, \ldots, a_m] \circ [a_{m+1}, \ldots, a_n] = [a_1, \ldots, a_m, a_{m+1}, \ldots, a_n]$$

The empty list is denoted by [ ]. It is a list, and also a leaf (it has no children) and an identifier. Of course, its length is 0.

The pseudo-term $list$ represents the sets of all lists. The pseudo-term $X\,oY$[12], which can also be written as $conc(X, Y)$, represents the concatenation of the lists X and Y. Let's now give a few examples:

```
>> X ~ [1,2] o [3,4].

X = [1,2,3,4].

>> [1,2,3,4] ~ X o [3,4].

X = [1,2].

>> X ~ X o [].

X ~ list.

>> X ~ Y o [].

Y ~ list,
X ~ list.
```

The answer to the fourth query is not very satisfactory, since the equality of X and Y is not inferred. This is due to the fact that it is not possible to reduce the domains of X and Y using that approximation.

The pseudo-term $size(L)$, as used in $N \sim size(L)$ is the set of the possible sizes of the list L. It may be used either to determine the size of a list or to constrain this size.

[12]The symbol in the middle is a lowercase « o ».

```
>> 10 ~ size(L).

L ~ [tree,tree,tree,tree,tree,tree,tree,tree,tree,tree].

>> N ~ size([1,2,3,4] o [5,6]).

N = 6.

>> N = size(L).

L ~ list,
N ~ ge(0).

>> lt(0) = size(L).

false.

>> N = le(0), N =size(L).

L = [],
N = 0.
```

The first query creates a 10-element list, in which the elements are unconstrained. The second query computes the size of a 6-element list. The third query shows us that the size of a list necessarily is a positive number. The fourth and and fifth one confirm that fact, by showing that the length of a list cannot be strictly negative, and that if it is negative or zero, it will be zero (thus binding the value of the list to the empty list).

Let's now give a little trick using lists.

```
>> [a] o X  ~  X o [a] ,  10 ~ size(X).

X = [a,a,a,a,a,a,a,a,a,a].

>> [tree] o X  ~  X o [tree] ,  10 ~ size(X).

A ex
X = [A,A,A,A,A,A,A,A,A,A],
A ~ tree.
```

Both queries above build a 10-element list, in which all the elements are identical. It is quite easy to see how that come by drawing on two lines the list $[a]oX$ and $X\,o[a]$. The value a propagates in the whole list.

We have seen the basic constraints over lists. Prolog IV also provides some constraints which are slightly more complex, which allow us to analyse more finely the contents of a list, using the relations $index$ and $inlist$.

The $index$ relation, as suggested by its name, is used to index lists in order to retrieve a given element of the list, as we would do for an array. It allows us to express a constraint stating that a tree T is the N-th element of a list L. We will now show a few examples, in which the lists are purely numerical, since it allows us to infer more information.

First, let's get the fifth element of a list:

```
>> X ~ index([1,3,2,4,6,5,8,10], 5).

X = 6.
```

We can also use the $index$ relation « backward », *i.e.* in order to find where the element 5 is:

```
>> 5 ~ index([1,3,2,4,6,5,8,10], N).

N = 6.
```

Of course, we run into slight problems when 5 occurs several times in the list:

Where is 5 (what is its rank in the list) ?

```
>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ cc(6,10).
```

The result is not that bad, but it could be even better, for instance by using the unions mode:

```
>> set_prolog_flag(interval_mode, union).

true.

>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ 6 u 7 u 10.

>> set_prolog_flag(interval_mode, simple).

true.
```

This time, the result is precise; we know exactly where the 5's are. Let's now try to give less precise data to $index$:

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], N), N ~ cc(1,5).

N ~ cc(1,5),
X ~ cc(1,6).
```

We have asked for the domain of the 5 first elements of the list. The answer is that X is a number between 1 and 6. Once again, we could obtain more precise results using the unions mode.

Let's now try to solve a small problem: given a list of numbers, we will try to determine which elements of L are such that they are equal to their rank:

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ cc(1,10).
```

The answer is that X can be any of the 10 first elements. That means that only 13 has been eliminated (This is easy, since the list does not even contain 13 elements. In order to get more information, let's try the unions mode:

```
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 8.

>> set_prolog_flag(interval_mode, simple).

true.
```

We obtain a better result, with only seven possible values. Of course, one may wonder why we don't obtain the exact result. An informal answer is that if you look at the domain given as an answer, all the elements of the domain are such that they are the index of one of the elements of the domain (possibly another one). This shows that even when using unions of intervals, our approximation is far from perfect. In order to get an exact solution, we will enumerate on the value of X:

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X), intsplit([X]).

X = 1;

X = 4;

X = 8.
```

This time, each answer is an exact value. This means that Prolog IV guarantees that these solutions are indeed the exact solutions to the problem.

Finally, let's see what happens when not much is known, by asking for the third element of an (unknown) list L:

```
>> X ~ index(L, 3).

L ~ [tree,tree,X|list],
X ~ tree.
```

The system determines that L must be a list, and that it contains at least three elements (the third one being equal to X). It is interesting to note that L is forced to be a list, which means that it is not possible any more to unify it with a tree which is not a list.

The $inlist$ relation is similar to $index$, just simpler. It does not bother with indexing, and it simply checks whether a tree is a member of a list. Let's try it with a list of atoms:

```
>> X ~ inlist([a,b,c]), X = d.

false.
```

The system detects that this X is not a valid value. With numbers, it would do like with $index$, *i.e.* the domains of the variable would be shrunk as much as possible.

To finish the list of relations over lists and trees, we have the *if* relation, which allows us to have a numerical *if-then-else* which is directed by the value of one of its argument. Here are a few queries using *if*:

```
>> X ~ if(B, 3,4).

B ~ cc(0,1),
X ~ cc(3,4).

>> X ~ if(B, 3,4), X = 5.

false.

>> X ~ if(B, 3,4), X ~ cc(0,3.5).

B = 1,
X = 3.
```

One must be careful when using *if*, because all the arguments of the *if* constraint are evaluated.

1.2.17 Booleans

In Prolog IV, Booleans are integers which are constrained to take the values 0 (for « false ») or 1 (for « true »). This choice presents several advantages. One of the main ones is to be able to handle Booleans like other numbers and in particular to let them appear wherever numbers may be accepted. Of course, specific relations are defined over Booleans, in order to write Boolean constraints. For instance, let's translate in Prolog IV the expression $a \Rightarrow b$, $b \Rightarrow c$ and $c \Rightarrow a$:

```
>> impl(A,B), impl(B,C), impl(C,A).

C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
```

We don't obtain decisive results. Let's now try to fix one of the variables to the value 1:

```
>> impl(A,B), impl(B,C), impl(C,A), C = 1.

C = 1,
B = 1,
A = 1.
```

This time, we get a good deduction.

We must insist on the fact that Booleans are constrained to take as value either 0 or 1, and not any other number between 0 and 1. We can see it immediately in « unions » mode, but also in standard mode:

```
>> impl(B,_).

B ~ cc(0,1).

>> impl(B,_), dif(B,1).

B = 0.
```

The first query shows a method to build a Boolean, and the second one shows that as soon as this Boolean cannot take the value 1, it then takes the value 0.

There exists a family of relations which use Booleans as the truth value for the satisfaction of constraint. Their name start with the letter « b », to which we concatenate the name of the original relation. For instance, there exists a relation ble, which associates a truth value to the fact that two numbers verify the le constraint. Here is an example:

```
>> B = ble(X, 0), X = 3.

X = 3,
B = 0.
```

We have asked whether or not 3 is lower than or equal to 0, and the answer is « no » (*i.e.* 0). Now, let's combine some logical connectors with this function, by asking about the numbers such that $x \leq -1$ AND $1 \leq x$:

```
>> and(ble(X,-1), ble(1,X)).

false.
```

There are no such numbers.

These relations may be used for several purposes.

- First, one may use them to delay the addition of a constraint until some other condition is met. Here, the $le(X, -1)$ constraint is added as soon as B's value is fixed to 1:

  ```
  >> B = ble(X,-1), B = 1.

  B = 1,
  X ~ le(-1).
  ```

  Be careful with these constraints, because the opposite constraint (*i.e.* $gt(X, -1)$) will be added if B's value is fixed to 0.

- Actually, the problem mentioned above may be a feature, not a bug, in which we can enumerate on the value of a Boolean variable in order to add a constraint or its opposite to the system.

- These constraints may also be used to build complex Boolean constraints and expressions:

  ```
  >> and(binlist(X, [3,4,5]), bcc(X,4.5, 10)).

  X = 5.
  ```

Let's now give a few of these relations: $bimpl$, bor, $band$,..., bcc, bco,..., ble, blt, bge, bgt, beq, $bdif$, $binlist$, $bint$, $bnint$, ...

1.2.18 Constraints over trees

There are some more constraints over general trees, which are linked to the definition of Prolog IV's privileged subdomains.

For every such domain, we usually have a relation which tests the membership to the domain, and one which tests the non-membership (whose name usually starts by a n). For every such relation, we also have some relations (whose name starts with a b) which associate the domain membership to a Boolean value.

About finite trees : *finite*, *infinite*, *bfinite*, *binfinite*.

About lists : *list*, *nlist*, *blist*, *bnlist*.

About leaves (single node trees) : *leaf*, *nleaf*, *bleaf*, *bnleaf*.

About identifiers : *identifier*, *nidentifier*, *bidentifier*, *bnidentifier*.

About real numbers : *real*, *nreal*, *breal*, *bnreal*.

A few queries

```
>> X = f(f(X)).

X = f(X).

>> X ~ finite, X = f(f(X)).

false.

>> X = [A] n finite.

X = [A],
A ~ finite.

>> B = breal(B).

B = 1.

>> B = leaf n list.

B = [].
```

The first query shows an example of infinite tree. The second one confirms that this tree indeed is not finite. The third query shows a one-element list which is finite; in the answer, the « finite » constraint is applied directly to the element. The fourth constraint shows that a Boolean number is a real number. Finally, the fifth one shows that the only list which is a leaf is the empty list.

1.2.19 Constraints and subdomains

Some relations allow us to constrain their arguments to belong to one of Prolog IV's privileged subdomains. This leads us to cite some of these subdomains:

tree, *finite*, *list*, *nlist*, *leaf*, *nleaf*, *identifier*, *nidentifier*, *real*, *nreal*, $cc(r_1, r_2)$, $oo(r_1, r_2)$, $\ldots$, where the r_i's are rational numbers which can be represented by floating-point numbers.

All these subdomains are relations as well. Some of them are negations, in which case they mean that an item does not belong to another subdomain. These relations are in fact the only ones which are used in Prolog IV's output (they may be connnected by « n », since the intersection of two privileged subdomains still is a privileged subdomain.

Finally, in « unions » mode, Prolog IV may also use the u relation in order to express the union of several sets (we have seen example in which the u relation is used between singletons.

Finally, one must note that in and $nint$ are relations, but there are not privileged subdomains in Prolog IV.

1.3 The programming environment

This section is devoted to the description of the software's features which are not a part of the Prolog IV language as such. This includes the description of primitives as well as parameters.

1.3.1 A few command line options

Most of the command line options are devoted to the parametrisation of the size of the various work areas used by Prolog IV. Some other options can be used to initialise Prolog IV.

These options can be given to Prolog IV when it is used from a terminal, as well as from the graphical interface (where these options can be set through a « preferences » dialog).

In the following, we call cell a machine doubleword (*i.e.* eight bytes on a 32-bit machine).

Here are now a few of these options. We have put within brackets the default value used by PrologIA (when appropriate). Nb represents a positive integer to be specified.

`-help`	Shows the list of available options.
`-heap` Nb	Allocates Nb cells to the $heap$ stack [700000].
`-trail` Nb	Allocates Nb cells to the $trail$ stack [400000].
`-env` Nb	Allocates Nb cells to the env stack[50000].
`-choice` Nb	Allocates Nb cells to the $choice$ stack[50000].
`-global` Nb	Allocates Nb cells to the static zone (consult,record,...) [500000].
`-local` Nb	Allocates Nb cells to the local stack (unification, constraint solvers) [5000].
`-work` Nb	Allocates Nb cells to the temporary zone (rational computations, Gauss) [50000].
`-nbpcindex` Nb	Maximum Nb of literals for compiled programs (PCode) [10000].
`-union`	Starts Prolog IV in « unions » mode [simple intervals].
`-iso`	Starts Prolog IV in « ISO » mode [« prolog4 » mode].

1.3.2 Prolog IV and the ISO standard

Prolog IV conforms to the ISO standard, which it extends. The primitives described in the standard are implemented[13], and Prolog IV's basic mechanisms are compatible with the standard.

However, we have tried to keep it easy to use the innovations introduced in Prolog IV, since we believe that these innovations are more characteristic of Prolog IV than its conformance to the standard. There are therefore two distinct modes of operation for Prolog IV, which allow the user to choose between a strict adherence to the ISO standard or the use of the Prolog IV natural mode.

[13] A small subset of them isn't and will be in the next release of Prolog IV.

Since the beginning of this tutorial, we have only used the « `prolog4` » mode, and never the « `iso` » mode. This « `prolog4` » mode is the default one for Prolog IV, and we recommend that you use it, unless you have specific needs for ISO compatibility. Switching from one mode to the other is quite easy:

```
>> iso.

true.

?-
```

We note that in ISO mode, the prompt becomes « `?-` ». To come back to the default mode, we type:

```
?- prolog4.

true.

>>
```

The prompt is back to its original look.

Now what are the differences between these two modes? They are mostly syntactical, and here are the most important ones:

- in ISO mode, pseudo-terms are not undrestood by the system, and they are interpreted as classical non-evaluated terms:

  ```
  ?- X = 3 + X, functor(X, N,A).

  A = 2,
  N = (+),
  X = 3+X.
  ```

  Here, we don't get a failure, because the equality is not between two numerical expressions, but only between trees; they simply build an infinite tree, just like the equation $X = f(3, X)$ would do. This is verified by the use of the `functor` primitive, which takes a term X and outputs the label of its root in N and its arity (*i.e.* the number of its children) in A. Here, we find out that X is a tree whose root node is labeled by « $+$ » and has two children (3 and X).

- Numbers which have the syntax of a floating-point number are encoded as double-precision IEEE floating-point numbers in ISO mode, and not as infinite precision numbers as in the prolog4 mode. Furthermore, the numbers entered in this way are not considered as real numbers by Prolog IV's solver.

  ```
  ?- X  = 1.00000000000000000000000000001.

  X = 1.0.

  ?- X = 1.1, real(X).

  false.
  ```

- The notation $name/arity$, which is often used in ISO Prolog for rule management, is not available in prolog4 mode (where the slash is interpreted as a division in the linear solver). In the following example, we get a failure, because $toto$ is not a number:

```
>> X = toto/4.

false.
```

1.3.3 Prolog IVś I/O primitives

Some primitives are available in Prolog IV which allow programmers to handle input and output. There are primitives for:

- opening a file, in read mode or write mode (`open`),
- reading terms from an open file (`read`),
- writing terms and characters in an open file (`write`, `nl`),
- closing a file (`close`),
- get information on open files.

Like in many programming systems, files are not handled directly. Let's introduce the notion of *stream*. An input stream is a « place » (like a file, for instance), where characters are available, and an output stream is a « place » where we can deposit characters. These streams are handled in Prolog IV through *stream descriptors*. These descriptors are Prolog entities which allow us to reference an open file from a program. It is of course some kind of a term[14], but it should be used only in straight unification and as argument to I/O primitives.

Let's give a short example. Let's assume that there is a file named `myfile.p4` in the current working directory, containing the following text:

```
toto(X,h(u),X).
[a,b,3333, 44444444444444,
555555555555, X].
```

This file contains two terms (each one is terminated by a period). It is important to note that each term must be terminated by a period, immediately followed by a space or a carriage return. If we want to read the terms in this file, we need to do as follows:

```
>> open('myfile.p4', read, S), read(S,T1), read(S, T2), close(S).

A ex
T2 ~ [a,b,3333,44444444444444,555555555555,tree],
T1 = toto(A,h(u),A),
S = 4443940,
A ~ tree.

>>
```

In this query, we first open the file by a call to `open`; the arguments are the name of the file[15], the mode in which it is to be opened (here, the mode is

[14] This is implementation-dependant, and could change at any moment, without prior warning. In Prolog IV, the stream descriptors currently are numbers.

[15] In Prolog IV's current implementation, file names must be speciied as atoms; since the name used in the example contains a non-alphanumerical character, it needs to be quoted.

read, and a variable which is unified with the descriptor of the new stream. Next, we have two calls to the read primitive; the arguments describe the stream from which to read, and the variable with which the term read will be unified. Finally, we have a call to close, which closes the stream; its only argument is the stream descriptor corresponding to the stream to be closed.

After running the query, we get the two terms in T1 and T2. The variable S contains the stream descriptor, and its value is of no use, since the close primitive has been called. Furthermore, this value is most likely different in your session.

Let's now try to write some characters and terms in a file called wri.p4: Essayons maintenant d'écrire des caractères ou des termes dans un fichier ecr.p4. On peut faire :

```
>> S ex open('ecr.p4', write, S),
      write(S,'Hi everybody, here is a term :-)'), nl(S),
      write(S, toto(h,10/20)), nl(S),
      close(S).

true.

>>
```

This time, we have explicitly stated that the value of S was something we were not interested in. Next, we have the call to open; this time, the second argument's value is write. We then use write to write a sequence of characters (in fact, this is a quoted atom), which we follow by a new-line (using the nl command). Of course, both of these calls include one argument which is the stream descriptor. We of course terminate the query by a call to close.

If we look at the wri.p4 file in an editor, it will now contain the two folowing lines:

```
Hi everybody, here is a term :-)
toto(h,1/2)
```

1.3.4 Entering rules

All along the present tutorial, we have not insisted much on the input of rules into the rule database, since we were rather presenting the features of the language through its predefined constraints. We have however encountered a few instances of the consult and reconsult primitives. When writing Prolog IV programs, you will need to write many rules and « feed » them to Prolog IV. We will describe in the present section the most common ways to do this. But first, let's start by recalling a few basics about Prolog rules.

Rule blocks

We call name and arity of a rule the name and arity of the head of the rule. We can then define a *rule block* as being a set of rules which have the same name and arity. Rule blocks are therefore identified by a couple $(name, arity)$. Usually, in Prolog, such rules are grouped together rather than split within a file or across a file[16]. The name of a rule block will often be denoted by $name/arity$.

[16] There are exceptions to this rule, and the ISO standard provides some options to deal with these exceptions.

Compiling and consulting

There are in Prolog IV three different ways to enter rules:

- by compiling a file,
- by consulting a file (or the console, which simply is a special file),
- by asserting rules (represented as terms) during the execution of a Prolog IV program.

There are in Prolog IV two distinct types of encoding, because of contradictory technical needs:

- The compiled form (or P-Code) presents the advantage of a good execution speed. Its major drawback is the impossibility to access the rules once they are compiled, to decompile them or to add rules to a block.
- The interpreted form (or asserted from) allows the user to access the rules at any time, either to decompile them or to add rules to a block. Of course, this results in a reduced execution speed, as well as an increase in memory usage.

In the present chapter, we have privileged the ease of use for very small programs. Therefore, we have used `consult`, which allowed us to input programs on the console. However, in Prolog IV's daily use, we strongly recommend the use of `compile`, which leads to a drastic performance increase, both in terms of execution time and in terms of memory consumption. It is indeed quite uncommon to need a dynamic handling of rules.

Rule input primitives

`consult` : reads a sequence of rules on the current input stream. If a rule block is read whose name and arity correspond to an already existing block, an error occurs, signaling the redefinition attempt. See also `reconsult`.

`consult(`*`file`*`)` : works exactly as `consult` without argument, except from the fact that it reads the rules from *file* (which must be an atom containing the name of the file).

`reconsult` : reads a sequence of rules on the current input stream. It works like `consult`, except that it allows the redefinition of user-input rules. When a rule block is read whose name and arity correspond to an already existing block, no error occurs. Of course, the user is not allowed to redefine the predefined primitives and relations.

`reconsult(`*`file`*`)` : works exactly as `reconsult` without argument, except from the fact that it reads the rules from *file* (which must be an atom containing the name of the file).

`compile(`*`file`*`)` : works exactly like `consult(`*`file`*`)`, except that it encodes rules in the compiled form.

`recompile(`*`file`*`)` : works exactly like `reconsult(`*`file`*`)`, except that it encodes rules in the compiled form.

In all cases above :

- The end of the file or the term « `end_of_file.` » end the input of the rules.

- The rules are encoded in memory.
- Trying to redefine predefined primitives or constraints invariably leads to an error.

Note that it is not possible to compile rules in the console; it is necessary to use a file. However, when using Unix and a `tty` terminal (like an xterm or a cmdtool), it is possible to use the pseudo-file `/dev/tty`, like in `compile('/dev/tty')`, who will wait for rules in the terminal, until an end of file is reached (usually by typing `CTRL-D`).

1.3.5 Factorisation des sorties

In the standard Prolog IV output, there is little factorisation performed:

```
>> record(q__factorize, 0).

true.

>> X = f(Y), Y = f(X).

X = f(X),
Y = f(Y).

>> X = f(g(2),g(2)).

X = f(g(2),g(2)).

>>
```

There exists an option which will try to factorise subterms which occur more than once, even if that leads to the introduction of existential variables:

```
>> record(q__factorize, 1).

true.

>> X = f(Y), Y = f(X).

X = Y,
Y = f(Y).

>> X = f(g(2),g(2)).

A ex
X = f(A,A),
A = g(2).

>>
```

By default, the `q_factorize` option is set to 0 (no factorisation).

1.3.6 Possible problems

We describe in the present section some classical problems and pitfalls which often occur.

Mixing numerical constraints

In the following query, we try to compute the domain of X, by giving a numerical formula:

```
>> X = cc(1,2).*.3 - exp(1).

X ~ real.
```

The answer is not at all satisfactory, since we are expecting a domain smaller than the set of all real numbers. In fact, by looking closely at the query, we can see that the symbol−, which is the short cut for $minuslin$ (the difference in the linear solver), has been used in a constraint in the interval solver, which is asking for trouble. Since the two solvers only communicate when variables reach a fixed value, we here get no solution. In short, **it is strongly recommended NOT to mix linear constraints and constraints over real numbers (interval solver)**.

This problem may easily be fixed by using the `.-.` symbol, which is the shortcut for the $minus$ relation (which works on the interval solver):

```
>> X = cc(1,2).*.3.-.exp(1).

X ~ oo(`>0.281718`,`>3.2817182`).
```

Here is another common mistake about mixing solvers:

```
>> X = -pi.

X ~ real.
```

Once again, the result is not satisfactory. This is exactly the main problem, since the unary minus which was used is the one from the linear solver. The correct version is:

```
>> X = .-.pi.

X ~ oo(-`>3.1415927`,-`>3.1415925`).
```

Now, that looks like $-\pi$.

Overflow in « unions » mode

The goal of the present section is to show that some constraints about integers or periodical functions may lead to problems when used in the « unions of intervals » mode. Let's show an example:

```
>> set_prolog_flag(interval_mode, union).

true

>> X ~ int, X ~ cc(0,10).

error: heap_overflow
>>
```

We have asked for integers between 0 and 10, and that led us to a « heap overflow » error, which means that the main workspace of Prolog IV has been saturated. What happened?

The problem comes from the order of the constraints. Prolog IV first started to build the union made of all integers, *i.e.* the union of all integer values which may be represented as floating-point numbers. There are many such numbers (several millions), and this union simply does not fit in memory[17].

Since the problem comes from the order of the constraints, it is of course easy to fix, by reordering the constraints in order to trestrict the variable's domain before to constrain it as being integer:

```
>> X ~ cc(0,10), X ~ int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

Prolog IV provides another way to solve this problem. When analysing a complex pseudo-expression, the compiler generates some code which will minimize the run-time problems. We can therefore write:

```
>> X ~ int n cc(0,10).

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

In pseudo-expressions, the order does not matter.

Which are the « hungry » relations?

They are the constraints which deal with integers, or with periodical functions (such as the trigonometric functions). The full list is: int, $nint$, $bint$, $bnint$, $floor$, $ceil$, sin, cos, tan and cot.

We must insist on the fact that these overflow problems *only* occur when unions of intervals are used.

1.3.7 Program debugging

The Prolog IV system includes an interactive debugger which may be used when writing programs. It implements the box model which has been introduced for Prolog, with of course some improvements due to the nature of Prolog IV. This debugger can work in a « poor » textual environment, but it reaches its full power when used in conjunction with Prolog IV's graphical environment. You then get a source-oriented debugger, with which you can follow the execution of your program in the source code.

In order to get a description of the debugging facilities included in Prolog IV, please refer to the chapters « Environment » and « Graphical environment. »

[17] It may on some particularly powerful computers, and it most likely will in a few years.