The Logic Programming Paradigm: A Tutorial

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

March 11, 1997

1 Introduction

Logic programming is a simple yet powerful formalism suitable for computing and for knowledge representation. It forms a formal basis for Prolog and for constraint programming. More recent applications of logic programming involve representation of various forms of reasoning and a computational approach to learning. The aim of this tutorial is to provide a simple introduction Prolog via logic programming.

2 Syntax

2.1 Terms

We begin by introducing an alphabet and all terms defined over it.

An *alphabet* consists of the following disjoint classes of symbols:

- *variables*, denoted by $x, y, z, \ldots$ possibly with subscripts,
- *function symbols*,
- *parentheses*, which are: (and),
- *comma*, that is: , .

Each function symbol has a fixed *arity*, that is the number of arguments associated with it. 0-ary function symbols are called

- *constants*, and are denoted by $a, b, c, d, \ldots$.

We denote function symbols of positive arity by $f, g, h, \ldots$. Sometimes, one assumes that function symbols have positive arity and constants are introduced as a separate class of symbols. The approach we take here is slightly more convenient.

Terms are defined inductively as follows:

- a variable is a term,

- if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

In particular every constant is a term. Terms are denoted by $s, t, u, w, \ldots$. A term with no variables is called *ground*. By $Var(t)$ we denote the set of variables occurring in t.

2.2 Queries and Programs

Next, we add to the alphabet

- *relation symbols* denoted by $p, q, r, \ldots$,
- *reversed implication*, that is: $\leftarrow$.

As in the case of function symbols, we assume that each relation symbol has a fixed arity associated with it. When the arity is 0, the relation symbol is usually called a *propositional symbol*.

We now define atoms, queries, clauses and programs as follows:

- if p is an n-ary relation symbol and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is an *atom*,
- a *query* is a finite sequence of atoms,
- a *clause* is a construct of the form $H \leftarrow \mathbf{B}$, where H is an atom and $\mathbf{B}$ is a query; H is called its *head* and $\mathbf{B}$ its *body*,
- a *program* is a finite set of clauses.

We denote atoms by $A, B, C, H, \ldots$, queries by Q, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, clauses by c, and programs by P, possibly with subscripts.

The empty query is denoted by $\square$. It stands for the empty conjunction, so it is considered true. When $\mathbf{B}$ is empty, $H \leftarrow \mathbf{B}$ is written $H \leftarrow$ and is called a *unit clause.*

By the *definition* of a relation symbol p in a given program P we mean the set of all clauses of P which use p in their heads. In mathematical logic it is customary to write $H \leftarrow \mathbf{B}$ as $\mathbf{B} \rightarrow H$. The use of reversed implication is motivated by the procedural interpretation according to which, given the atom $p(\mathbf{s})$, the clause $p(\mathbf{s}) \leftarrow \mathbf{B}$ is viewed as a part of the definition of the relation p.

We adopt here a fixed *universal language* in which there are infinitely many variables, and in which in each arity there are infinitely many function symbols and relation symbols, and assume that all considered queries and programs are written in this universal language. This choice sounds perhaps artificial but it is quite natural if one studies Prolog programs. Indeed, any Prolog manual defines such a universal language in which arbitrary queries and programs can be written. In fact, even in the case of one fixed program arbitrary queries can be posed and in these queries arbitrary constants, function symbols and relation symbols can appear.

2.3 Prolog Syntax

Syntax of Prolog differs in a number of small but important aspects from that of logic programming. All programs presented here are legal Prolog programs so it is perhaps an appropriate moment to discuss these differences.

First, variables are denoted by strings starting with an upper case letter or "_" (underscore) For example `Xs` is a variable. Prolog also allows so-called *anonymous variables*, written as "_" (underscore). These variables have a special interpretation, because each occurrence of "_" in a query or in a clause is interpreted as a *different* variable. Thus by definition each anonymous variable occurs in a query or a clause only once.

Anonymous variables form a simple and elegant device and their use increases the readability of programs in a remarkable way. Modern versions of Prolog, like SICStus Prolog [CW93], encourage the use of anonymous variables by issuing a warning if a non-anonymous variable that occurs only once in a clause is encountered.

Strings starting with a lower-case letter are reserved for the names of function or relation symbols. For example, `f` stands for a constant, function or relation symbol.

Another regrettable, terminological difference between logic programming and Prolog is that the word "atom" has a completely different meaning in the context of Prolog. Namely, in Prolog, an *atom* denotes any non-numeric constant. Throughout this article only the logic programming meaning of "atom" will be used.

In what follows we adhere to Prolog's notation while presenting Prolog programs but when studying these programs from a formal view we shift to the logic programming notation. This will not cause confusion because no entities within logic programs are denoted by upper case letters.

In contrast to first-order logic, in Prolog the *same* name can be used for function or relation symbols of different arity. Even more, the same name with the same arity can be used both for function and relation symbols. This facility is called *ambivalent syntax*. A function or a relation symbol `f` of arity `n` is then referred to as `f/n`. So in a Prolog program we can use both a relation symbol `p/2` and function symbols `p/1` and `p/2` and build syntactically legal terms or atoms like `p(p(a,b),c,p(X))`. In Prolog terminology *relation symbol* is synonymous with *predicate*.

In the presence of ambivalent syntax the distinction between function symbols and relation symbols and, consequently, between terms and atoms, disappears but in the context of queries and clauses it is clear which symbol refers to which syntactic category. The ambivalent syntax facility allows us to use the same name for naturally related function or relation symbols.

Next, a Prolog program is viewed as a *sequence* and not as a set of clauses.

Finally, in Prolog "`:-`" is used instead of "$\leftarrow$" ("`<-`" would have been a better choice), each query and clause ends with the period "`.`" and in the unit clauses "`:-`" is omitted. Unit clauses are called *facts* and non-unit clauses are called *rules*. Of course, queries and clauses can be broken over several lines.

Prolog has several (around one hundred) of built-in relations, so relations that are internally defined. The clauses, the heads of which refer to these built-in relations, are ignored. In more modern versions of Prolog, like SICStus Prolog, a warning is issued in case such an attempt at redefining a built-in relation is encountered.

2.4 Interaction with a Prolog System

Even though we did not yet explain how computing in Prolog takes place, it is useful to get an idea how one interacts with a Prolog system. For a more complete description the reader is referred to a language manual. The interaction starts by typing `sicstus` for SICStus Prolog or `eclipse` for the ECL^{i}PS^e system. There are some small differences in interaction with these two systems which we shall disregard. The SICStus Prolog system replies by the prompt "`| ?-`" while the prompt of the ECL^{i}PS^e system is "`[eclipse 1]:`". Now the program can be read in by typing `[file-name]` followed by the period ".". Assuming that the program is syntactically correct, the system replies with the answer "yes" followed by the prompt. Now a query to the program can be submitted, by typing it with the period "." at its end.

The idea is that the system evaluates the query w.r.t. the read in program and reports an answer. The system replies are of two forms. If the query succeeds, an answer is printed followed by "`yes`". The answer shows, informally, for what values of the variables the query becomes true. (A precise definition will be given in the next section). If the query fails, the answer "`no`" is printed. At this point typing the return key terminates the computation, whereas typing ";" (followed by the return key for the SICStus Prolog) is interpreted as the request to produce the next answer. If the query fails, or if the requests to produce the next answer fails, the answer "`no`" is printed.

Below, we use queries both to find *one* answer and to find *all* answers. Finally, typing "`halt.`" finishes the interaction with the system.

2.5 First Example

As a first example of a Prolog program consider the following example from Sterling and Shapiro [SS86] (in fact ftp-ed from ftp.cwru.edu, directory ArtOfProlog).

Example 2.1 *Biblical family relationships*

```
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).
father(terach,abraham).
father(terach,haran).
father(terach,nachor).

female(milcah).
female(sarah).
female(yiscah).

male(abraham).
male(haran).
male(isaac).
male(lot).
male(nachor).
male(terach).

mother(sarah,isaac).

daughter(X,Y) :- father(Y,X), female(X).

son(X,Y) :- father(Y,X), male(X).
```

Here the relation `father(X,Y)` is to be read "`X` is the father of `Y`", so the last rule should be read as "`X` is a son of `Y` if `Y` is the father of `X` and `X` is a male" and similarly with the other facts and rules.

Even though we did not yet explain how computing in Prolog takes place, it is useful to get an idea how this program can be used. So first of all, it can used for testing:

```
[eclipse 2]: father(haran,lot).

yes.
[eclipse 3]: mother(haran,lot).

no (more) solution.
```

Next, it can be used to compute one or more solutions:

```
[eclipse 4]: son(X,terach).

X = abraham     More? (;)

X = haran     More? (;)

X = nachor
yes
```

Further, it can be used to compute answers to more complex queries like this one which asks which of the sons of `terach` are fathers themselves:

```
[eclipse 18]: son(X, terach), father(X,_).

X = abraham     More? (;)

X = abraham     More? (;)

X = haran     More? (;)

X = haran     More? (;)

X = haran     More? (;)

X = haran     More? (;)

X = haran     More? (;)

X = haran     More? (;)

no (more) solution.
```

We shall explain later why several repeated answers are produced. □

2.6 Lists in Prolog

A data structure that supports a single operation on sequences — an insertion of an element at the front — is usually called a *list*. Lists form a fundamental data structure and in Prolog special, built-in notational facilities for them are available. In particular, the pair consisting of a constant `[]` and a binary function symbol `[.|..]` is used to define them. Formally, lists in Prolog are defined inductively as follows:

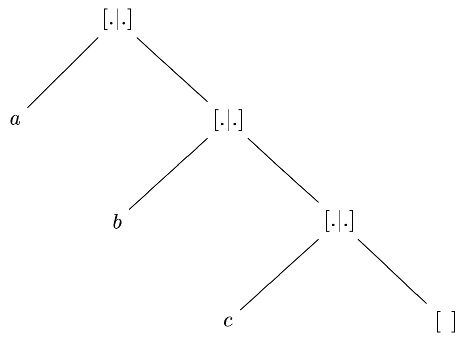

Figure 1: A list

- `[]` is a list,
- if `xs` is a list, then `[x | xs]` is a list; `x` is called its *head* and `xs` is called its *tail*.

`[]` is called the *empty list*.

For example, `[s(0)|[]]` and `[0|[X|[]]]` are lists, whereas `[0|s(0)]` is not, because `s(0)` is not a list. Note that the elements of a list need not to be ground. The tree depicted in Figure 1 represents the list `[a|[b|[c|[]]]]`.

The list notation is not very readable and even short lists become then difficult to parse. So the following shorthands are carried out internally in Prolog for $\mathtt{n} \geq \mathtt{1}$:

- $[\mathtt{s_0}|[\mathtt{s_1}, ..., \mathtt{s_n}|\mathtt{t}]]$ abbreviates to $[\mathtt{s_0}, \mathtt{s_1}, ..., \mathtt{s_n}|\mathtt{t}]$,
- $[\mathtt{s_0}, \mathtt{s_1}, ..., \mathtt{s_n}|[\ ]]$ abbreviates to $[\mathtt{s_0}, \mathtt{s_1}, ..., \mathtt{s_n}]$.

Thus for example, `[a|[b|c]]` abbreviates to `[a,b|c]`, and the depicted list `[a|[b,c|[]]]` abbreviates to a more readable form, namely `[a,b,c]`.

To enhance the readability, one often uses in programs the names ending with "`s`" to denote variables which are meant to be instantiated to lists.

Example 2.2 To illustrate the use of lists we now discuss the perhaps most often cited Prolog program. This program concatenates two lists. The inductive definition of concatenation is as follows:

- the concatenation of the empty list `[]` and the list `ys` yields the list `ys`,
- if the concatenation of the lists `xs` and `ys` equals `zs`, the concatenation of the lists `[x | xs]` and `ys` equals `[x | zs]`.

This translates into following program `APPEND`.

```
% app(Xs, Ys, Zs) :- Zs is the result of concatenating the lists Xs and Ys.
app([], Ys, Ys).
app([X | Xs], Ys, [X | Zs]) :- app(Xs, Ys, Zs).
```

`APPEND` can be used not only to concatenate the lists:

```
| ?- app([a,b], [a,c], Zs).

Zs = [a,b,a,c]
```

but also to split a list in all possible ways:

```
| ?- app(Xs, Ys, [a,b,a,c]).

Xs = []
Ys = [a,b,a,c] ;

Xs = [a]
Ys = [b,a,c] ;

Xs = [a,b]
Ys = [a,c] ;

Xs = [a,b,a]
Ys = [c] ;

Xs = [a,b,a,c]
Ys = [] ;

no
```

□

3 Computational Process — Procedural Interpretation

From the computational point of view a query $A_1, \ldots, A_n$ should be viewed as a request for finding values for the variables $x_1, \ldots, x_k$ such that the conjunction $A_1 \wedge \ldots \wedge A_n$ becomes true.

In turn, a clause $H \leftarrow B_1, \ldots, B_n$ should be interpreted as a statement "to prove H prove each of B_i's. The reverse implication can thus be read as "if" and "," as "and". The order in which these B_i' are to be proved is of importance and will be discussed below.

In logic programming variables represent unknown values, very much like in mathematics. The values assigned to variables are terms. These values are assigned by means of certain substitutions, called most general unifiers. The process of computing most general unifiers is called unification. So unification forms a basic mechanism by means of which logic programs compute.

The use of unification to assign values to variables forms a distinguishing feature of logic programming and is one of the main differences between logic programming and other programming styles. Unification was defined in Robinson [Rob65] in the context of automated theorem proving. Its use for computing is due to Kowalski [Kow74].

This computational interpretation of logic programs is called *procedural interpretation*. It explains *how* logic programs compute. While Prolog differs from logic programming, it can be naturally introduced by defining the computation mechanism of logic programs first and then by explaining the differences.

This process has the additional advantage of clarifying certain design decisions (like the choice of the search mechanism) and of shedding light on the resulting dangers (like the possibility of divergence).

3.1 Substitutions

We begin now the explanation of the computational interpretation by defining substitutions.

Substitutions bind variables to terms. They are the only means of assigning values to variables within the logic programming framework. The relevant substitutions are automatically generated during the computation process, so — in contrast to the imperative programming — the assignment of values to variables takes place implicitly.

More precisely, consider now a fixed language of terms. A *substitution* is a finite mapping from variables to terms which assigns to each variable x in its domain a term t different from x. We write it as

$$\{x_1/t_1, \ldots, x_n/t_n\}$$

where

- $x_1, \ldots, x_n$ are different variables,
- $t_1, \ldots, t_n$ are terms,
- for $i \in [1, n]$, $x_i \neq t_i$.

Informally, it is to be read the variables $x_1, \ldots, x_n$ are *bound to* $t_1, \ldots, t_n$, respectively. A pair x_i/t_i is called a *binding*. When $n = 0$, the mapping becomes the empty mapping. The resulting substitution is then called *empty substitution* and is denoted by ϵ. If θ is a 1-1 and onto mapping from its domain to itself, then θ is called a *renaming*. For example $\{x/y, y/z, z/x\}$ is a renaming.

Given a substitution $\theta := \{x_1/t_1, \ldots, x_n/t_n\}$ we put $Dom(\theta) = \{x_1, \ldots, x_n\}$ and $Var(\theta) = \{x_1, \ldots, x_n\} \cup \bigcup_{i=1}^{n} Var(t_i)$. So $Var(\theta)$ denotes the set of all variables that occur somewhere in θ. We call $Dom(\theta)$ the domain of θ.

We now define the result of *applying a substitution* θ *to a term* s, written as $s\theta$, as the result of the *simultaneous* replacement of each occurrence in s of a variable from the domain of θ by the corresponding term in its range.

Example 3.1 Consider a language allowing us to build arithmetic expressions in prefix form. It contains two binary function symbols, "+" and "·" and infinitely many constants: 0, 1, ... Then $s = +(\cdot(x, 7), \cdot(4, y))$ is a term and for the substitution $\theta = \{x/0, y/+(z, 2)\}$ we have

$$s\theta = +(\cdot(0, 7), \cdot(4, +(z, 2))).$$

□

The term $s\theta$ is called an *instance* of s and if θ is a renaming, then $s\theta$ is called a *variant* of s. For example $f(x, z)$ is a variant of $f(z, y)$ because $f(z, y)\{x/y, y/z, z/x\} = f(x, z)$ and we already note that $\{x/y, y/z, z/x\}$ is a renaming. An instance is called *ground* if it contains no variables. Finally, s is called *more general than* t if t is an instance of s.

Next, we define the *composition* of substitutions θ and η, written as $\theta\eta$, as follows. We put for a variable x

$$(\theta\eta)(x) := (x\theta)\eta.$$

In other words, $\theta\eta$ is the substitution that assigns to a variable x the term obtained by applying the substitution η to the term $x\theta$.

A more useful for specific calculations, equivalent, definition is as follows. The composition $\theta\eta$ is the outcome of the following procedure:

- remove from the sequence

$$x_1/t_1\eta, \ldots, x_n/t_n\eta, y_1/s_1, \ldots, y_m/s_m$$

  the bindings $x_i/t_i\eta$ for which $x_i = t_i\eta$ and the bindings y_j/s_j for which $y_j \in \{x_1, \ldots, x_n\}$,

- form from the resulting sequence of bindings a substitution.

The next lemma shows that the composition of substitutions is associative and that when writing a sequence of substitutions the parentheses can be omitted.

Lemma 3.2 (Associativity)

(i) $(A\theta)\eta = A(\theta\eta)$.

(ii) $(\theta\eta)\gamma = \theta(\eta\gamma)$. □

The notions of an (ground) instance, of a variant and the notation $Var(\ldots)$can be defined without any difference for *arbitrary* strings of symbols in a given language. In the sequel, we shall use these notions, the notation and the results for other syntactic constructs of interest.

3.2 Most General Unifiers

We already mentioned at the beginning of this section that unification is a fundamental concept that is crucial for logic programming.

Informally, unification is the process of making atoms identical by means of certain substitutions. For example, the atoms $p(a, y, z)$ and $p(x, b, z)$, with a, b constants and x, y, z variables, can be made identical by applying to them the substitution $\{x/a, y/b\}$: both sides then become $p(a, b, z)$. But the substitution $\{x/a, y/b, z/a\}$ also makes these two atoms identical. Such substitutions are called unifiers. The first unifier is preferable because it is "more general" — the second one is a "special case" of the first one. More precisely, the first unifier is a *most general unifier* of $p(a, y, z)$ and $p(x, b, z)$ while $\{x/a, y/b, z/a\}$ is not.

We begin the formal presentation with the following definition needed to define the notion of a most general unifier.

Definition 3.3 Let θ and τ be substitutions. We say that θ is *more general than* τ if for some substitution η we have $\tau = \theta\eta$. □

Thus θ is more general than τ if τ can be obtained from θ by applying to it some substitution η.

Example 3.4 The substitution $\{x/y\}$ is more general than $\{x/a, y/a\}$, because $\{x/y\}\{y/a\} = \{x/a, y/a\}$. □

The following notion is the key concept of this section.

Definition 3.5

- θ is called a *unifier* of As and B if $A\theta = B\theta$. If a unifier of A and B exists, we say that A and B are *unifiable*.

- θ is called a *most general unifier* (*mgu* in short) of A and B if it is a unifier of A and B that is more general than all unifiers of A and B. □

Intuitively, an mgu is a substitution which makes two atoms equal but which does it in a "most general way", without unnecessary bindings. So θ is an mgu if for every unifier η for some substitution γ we have $\eta = \theta\gamma$.

Example 3.6
(i) Consider the atoms $p(g(x,a),z)$ and $p(y,b)$. Then $\{x/c, y/g(c,a), z/b\}$ is one of their unifiers and so is $\{y/g(x,a), z/b\}$ which is more general than the first one, since $\{x/c, y/g(c,a), z/b\} = \{y/g(x,a), z/b\}\{x/c\}$.

Actually, one can show that $\{y/g(x,a), z/b\}$ is an mgu of $p(g(x,a),z)$ and $p(y,b)$. In particular, we have

$$\{x/c, y/g(c,a), z/b\} = \{y/g(x,a), z/b\}\{x/c, y/g(c,a), z/b\}.$$

(ii) Consider the atoms $p(g(x,a),z)$ and $p(g(x,b),b)$. They are not unifiable, since for no substitution θ we have $a\theta = b\theta$.

(iii) Finally consider the atoms $p(x,a)$ and $p(f(x),a)$. They are not unifiable either because for any substitution θ the atom $x\theta$ has more symbols than $p(x)\theta$. □

The problem of deciding whether two atoms are unifiable is called the *unification problem.* This problem was solved in Robinson [Rob65] where the following result was established.

Theorem 3.7 (Unification) *There exists an algorithm that given a pair of two atoms determines whether they unify and if they do it produces an mgu of them, and otherwise it terminates with failure.*

3.3 SLD derivations

To define the computational process we now consider pairs of the form $\langle Q, \theta \rangle$, where Q is a query with one atom selected in it if it is non-empty and where θ is a substitution,

Intuitively, $\langle Q, \theta \rangle$ denotes the query Q in the "environment" θ, that is, the variables of Q should be "interpreted" as in θ. We do not identify $\langle Q, \theta \rangle$ with $Q\theta$ for technical reasons we prefer not to elaborate upon. It suffices to note that the former contains more information than the latter: $Q\theta$ can be computed from $\langle Q, \theta \rangle$ but not vice versa. The role of the selected atom will be clear in a moment.

Consider a program P. We now define a relation

$$\langle Q, \theta \rangle \rightarrow \langle Q', \theta' \rangle$$

as follows. Consider a pair $\langle \mathbf{A}, B, \mathbf{C},\ \theta \rangle$ with B the selected atom of $\mathbf{A}, B, \mathbf{C}$ and a clause c from P. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint with $\mathbf{A}, B, \mathbf{C}$, that is such that

$$Var(H \leftarrow \mathbf{B}) \cap (\,Var(\mathbf{A}, B, \mathbf{C}) \cup Var(\theta)) = \emptyset.$$

Suppose that $B\theta$ and H unify and let η be an mgu of $B\theta$ and H. Then we call the pair $\langle \mathbf{A}, \mathbf{B}, \mathbf{C},\ \theta\eta \rangle$ an *SLD-resolvent of* $\langle \mathbf{A}, B, \mathbf{C},\ \theta \rangle$ *and* c and write

$$\langle \mathbf{A}, B, \mathbf{C},\ \theta \rangle \rightarrow \langle \mathbf{A}, \mathbf{B}, \mathbf{C},\ \theta\eta \rangle.$$

Consider now a program P and a query Q. By an *SLD-derivation* of Q w.r.t. P we mean a maximal sequence of pairs $\langle Q_i, \theta_i \rangle$ such that for $i \geq 0$ we have $\langle Q_i, \theta_i \rangle \rightarrow \langle Q_{i+1}, \theta_{i+1} \rangle$.

An SLD-derivation is called *successful* if it is finite and the query in the last pair is empty. An SLD-derivation is called *failed* if it is finite, the query in the last pair is non-empty and the last pair has no $\rightarrow$ successor for all clauses c from P.

If

$$\langle Q, \epsilon \rangle \rightarrow^* \langle \Box, \tau \rangle,$$

where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$ (so $\rightarrow^*$ is the outcome of iterating $\rightarrow$ zero or more times), then we call $Q\tau$ a *computed instance* of Q w.r.t. P and the restriction of τ to the variables occuring in Q, a *computed answer substitution* for Q w.r.t. P.

Intuitively, $\langle Q, \epsilon \rangle \rightarrow^* \langle \Box, \tau \rangle$ means that the query Q has been proved with the answer τ.

3.4 Computation Process in Prolog

In Prolog always the first atom from the left is selected. The resulting SLD-derivations starting in a pair $\langle Q, \epsilon \rangle$ can be arranged in a tree in which the direct descendants of every node are ordered according to the order the clauses used appear in the program. More precisely, in this tree

- the branches are SLD-derivations starting in $\langle Q, \epsilon \rangle$ such that in each non-empty query the the leftmost atom is selected.
- every node $\langle Q', \theta' \rangle$ has exactly one descendant for every clause c from P that can be used to form a successor in the $\rightarrow$ relation.

In this tree a depth-first search for computed answer substitutions takes place. If in this search a node with the empty query is encountered, then the associated computed answer substitution is printed and the search is suspended. The request for more solutions (";") results in a resumption of the search from the last visited node until a new node with the empty query is visited. If the tree has no (more) nodes with the empty query, then failure is reported, by printing the answer "`no`".

The reader can now manually simulate (though it is a tedious exercise) the results of the Prolog computations for the queries considered in Subsections 2.5 and 2.6.

4 Arithmetic in Prolog

4.1 Operators

Prolog provides integers as a built-in data structure, with various operations on them. In the case of SICStus Prolog these operations include the following binary operations:

- addition, written as `+`,
- subtraction, written as `-`,
- multiplication, written as `*`,
- integer division, written as `//`,
- remainder of the integer division, written as `mod`,

and the following unary operations:

- negation of a natural number, written as `-`,

- absolute value, written as `abs`.

We call the above operations *arithmetic operators*.

According to the usual notational convention of logic programming and Prolog, the relation and function symbols are written in a *prefix form*, that is in front of the arguments. In contrast, in accordance with the usage in arithmetic, the binary arithmetic operators are written in the *infix form*, that is between the arguments, while the unary arithmetic operators are written in the prefix form. On the other hand, negation of a natural number can be written in the *bracketless prefix form*, that is without brackets surrounding its argument.

Recall that the integer division is defined as the integer part of the usual division outcome and given two integers `x` and `y` such that $\texttt{y} \neq \texttt{0}$, `x mod y` is defined as `x - y*(x//y)`.

The use of the infix and bracketless prefix form for arithmetic operators leads to well-known ambiguities. For example, `4+3*5` could be interpreted either as `(4+3)*5` or `4+(3*5)` and `-3+4` could be interpreted either as `(-3)+4` or `-(3+4)`. Further, `12//4//3` could be interpreted either as `(12//4)//3` or `12//(4//3)`, etc.

Such ambiguities are resolved in Prolog in a way that also allows for the presence of other function symbols written in the infix or bracketless prefix form. To this end Prolog provides a means to declare an *arbitrary* function symbol as an infix binary symbol or as a bracketless prefix unary symbol, with a fixed *priority* that determines its binding power and a certain *mnemonics* that implies some (or no) form of associativity. Function symbols that are declared in such a way are called *operators*.

The priority and mnemonics information allows us to associate with each term written using the infix or bracketless prefix notation, a unique term written in the customary prefix notation, that serves as the *interpretation* of the original one.

In SICStus Prolog priority is a natural number between 1 and 1200 inclusive. Informally, the higher the priority the lower the binding power.

There are seven mnemonics. We list them together with (if any) the associativity information each of them implies. For the binary function symbols these are

- `xfx` (no associativity),
- `xfy` (right associativity),
- `yfx` (left associativity),

and for the unary function symbols these are

- `fx`,
- `fy`,
- `xf`,
- `yf`.

The mnemonics `yfy` is not allowed, as it would imply both left and right associativity and would thus permit the interpretation of a term of the form `s f t f u` *both* as `(s f t) f u` and `s f (t f u)`. Consequently, it would not provide a unique interpretation to the term.

The *declaration of an operator* `g` is a statement of the form

```
:- op(pr, mn, g).
```

written in the program before the first use of `g`; `pr` is the priority of `g` and `mn` is the mnemonic of `g`.

Formally, in presence of operator declarations, terms are defined inductively as follows, where with each term we associate a priority in the form of a natural number between 0 and 1200 inclusive and an interpretation in the sense mentioned above:

- a variable is a term with priority 0 and itself as its interpretation,
- if t is a term with interpretation $i(t)$, then (t) is a term with priority 0 and interpretation $i(t)$ (that is, bracketing reduces the priority to 0),
- if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms with respective interpretations $i(t_1), \ldots, i(t_n)$, then $f(t_1, \ldots, t_n)$ is a term with priority 0 and interpretation $f(i(t_1), \ldots, i(t_n))$,
- if f is a binary operator with priority pr and s and t are terms with respective priorities $pr(s)$ and $pr(t)$ and interpretations $i(s)$ and $i(t)$, then sft is a term with priority pr and interpretation $f(i(s), i(t))$, according to the table below and subject to the corresponding conditions:

mnemonics	conditions
`xfx`	$pr(s) < pr$, $pr(t) < pr$
`xfy`	$pr(s) < pr$, $pr(t) \leq pr$
`yfx`	$pr(s) \leq pr$, $pr(t) < pr$

- if f is a unary operator with priority pr and s is a term with priority $pr(s)$ and interpretation $i(s)$, then the following is a term with priority pr and interpretation $f(i(s))$, according to the table below and subject to the corresponding condition:

term	mnemonics	condition
fs	`fx`	$pr(s) < pr$
fs	`fy`	$pr(s) \leq pr$
sf	`xf`	$pr(s) < pr$
sf	`yf`	$pr(s) \leq pr$

The arithmetic operators are disambiguated by declaring them internally as follows (the declarations are the ones used in SICStus Prolog):

```
:- op(500, yfx, [+, -]).
:- op(500, fx, -).
:- op(400, yfx, [*, //]).
:- op(300, xfx, mod).
```

Here a list notation is used to group together the declarations of the operators with the same mnemonics and priority.

Returning to our original examples of possibly ambiguous arithmetic terms, we now see that `4+3*5` is a term with priority 500 and interpretation `+(4, *(3,5))`, `-3+4` is a term with priority 500 and interpretation `+(-(3), 4)` and `12//4//3` is a term with priority 400 and interpretation `//(//(12,4), 3)`. In addition, note that the declaration of negation of a natural number with the mnemonics `fx` implies that `- - 3` is not a (legal) term. In contrast, `-(-3)` is a (legal) term.

The list of arithmetic operators introduced at the beginning of this section together with infinitely many integer constants: 0, -1, 1, -2, 2, ... determines a language of terms in the sense just defined. We call terms defined in this language *arithmetic expressions* and introduce the abbreviation *gae* for ground arithmetic expressions.

It is worthwhile mentioning that Prolog built-in operators, in particular arithmetic operators, can also be written in the customary prefix notation. (The comma "," though requires a slightly different treatment: in SICStus Prolog it has to be written then as ',', like for instance in `','(A,B)`.) In particular each arithmetic expression can also be written as a term that is its interpretation in the sense discussed above. In the case of ground arithmetic expressions both forms are equal. For example, we have

```
| ?- 4+3*5 =:= +(4, *(3,5)).

yes
```

4.2 Arithmetic Comparison Relations

With each gae we can uniquely associate its *value*, computed in the expected way. Prolog allows us to compare the values of gaes by means of *arithmetic comparison relations* (in short *comparison relations*). The following six comparison relations are provided:

- "less than", written as `<`,
- "less than or equal", written as `=<`,
- "equality", written as `=:=`,
- "inequality", written as `=\=`,
- "greater than or equal", written as `>=`,
- "greater than", written as `>`.

The equality relation `=:=` should not be confused with the "is unifiable with" relation `=/2`, another Prolog's built-in written using the infix notation and defined internally by a single clause:

```
% X = Y :- X and Y are unifiable.
X = X.
```

By definition the built-in relations cannot be redefined, so clauses in which the head refers to the comparison relations are ignored.

The comparison relations work on gaes and produce the outcome expected to anyone familiar with the basics of arithmetic. So for instance, $>$ compares the values of two gaes and succeeds if the value of the first argument is larger than the value of the second and fails otherwise. Thus, for example

```
| ?- 5*2 > 3+4.

yes

| ?- 7 > 3+4.

no
```

However, when one of the arguments of the comparison relations is not a gae, the computation *ends in an error*. For example, we have

```
| ?- [] < 5.

! Error in arithmetic expression: [] is not a number

no
```

Such type of errors are called *run-time errors*, because they happen during the program execution. As a simple example of the use of the comparison relations consider the following program that checks whether a list is an ordered one.

```
% ordered(Xs) :- Xs is an =<-ordered list of integers.
ordered([]).
ordered([_]).
ordered([X, Y | Xs]) :- X =< Y, ordered([Y| Xs]).
```

We now have

```
[eclipse 2]: ordered([1,1,2,3]).

yes.
```

but also

```
[eclipse 3]: ordered([1,X,1]).
instantiation fault in 1 =< X
```

Here a run-time took place here because the arithmetic comparison relation was applied to an argument that is not an integer. On the other hand, we also have

```
[eclipse 4]: ordered([X]).

X = X     More? (;)
yes.
```

which shows that programs that use the arithmetic comparison relations are quite subtle.

Quicksort

One of the most fundamental operations on the lists is sorting. The task is to sort a list of integers. We present here Prolog's version of the *quicksort* procedure proposed by Hoare [Hoa62]. According to this sorting procedure, a list is first partitioned into two sublists using an element `X` of it, one consisting of elements smaller than `X` and the other consisting of elements larger or equal than `X`. Then each sublist is quicksorted and the resulting sorted sublists are appended with the element `X` put in the middle. This can be expressed in Prolog as follows where `X` is chosen to be the first element of the given list:

```
% qs(Xs, Ys) :- Ys is an ordered permutation of the list Xs.
qs([], []).
qs([X | Xs], Ys) :-
```

```
        part(X, Xs, Littles, Bigs),
        qs(Littles, Ls),
        qs(Bigs, Bs),
        app(Ls, [X | Bs], Ys).

% part(X, Xs, Ls, Bs) :- Ls is a list of elements of Xs which are < X,
%                        Bs is a list of elements of Xs which are >= X.
part(_, [], [], []).
part(X, [Y | Xs], [Y | Ls], Bs) :- X > Y, part(X, Xs, Ls, Bs).
part(X, [Y | Xs], Ls, [Y | Bs]) :- X =< Y, part(X, Xs, Ls, Bs).
```

augmented by the `APPEND` program.

For example:

```
[eclipse 23]: qs([7,9,8,1,5], Ys).

Ys = [1, 5, 7, 8, 9]     More? (;)

no (more) solution.
```

However, we also have

```
[eclipse 20]: qs([3,X,0,1], Ys).
instantiation fault in 3 > X
```

and

```
[eclipse 5]: qs([X], Ys).

X = X
Ys = [X]     More? (;)
yes.
```

4.3 Evaluation of Arithmetic Expressions

So far we have presented programs that use ground arithmetic expressions, but have not yet presented any means of evaluating them. The advantages of evaluating gaes will become clear in a moment. For example, no facilities have been introduced so far to evaluate `3+4`. All we can do at this stage is to check that the outcome is `7` by using the comparison relation `=:=` and the query `7 =:= 3+4`. But using the comparison relations it is not possible to *assign* the value of `3+4`, that is 7, to a variable, say `X`. Note that the query `X =:= 3+4` ends in an error.

To overcome this problem the *arithmetic evaluator* `is/2` is incorporated into Prolog. `is/2` is defined internally as an infix operator with the following declaration in the case of SICStus Prolog:

```
:- op(700, xfx, is).
```

Consider the call `s is t`. Then `t` has to be a ground arithmetic expression (gae). The call of `s is t` results in the unification of the *value* of the gae `t` with `s`. If `t` is not a gae then a run-time error arises.

Thus, the following possibilities arise.

- `t` is a gae.

  Let `val(t)` be the value of `t`.

  - `s` is identical to `val(t)`.

    Then the arithmetic evaluator succeeds and the empty computed answer substitution is produced. For example,

    ```
    | ?- 7 is 3+4.

    yes
    ```

  - `s` is a variable.

    Then the arithmetic evaluator also succeeds and the computed answer substitution {`s`/`val`(`t`)} is produced. For example,

    ```
    | ?- X is 3+4.

    X = 7
    ```

  - `s` is not identical to `val(t)` and is not a variable.

    Then the arithmetic evaluator fails. For example,

    ```
    | ?- 8 is 3+4.

    no
    | ?- 3+4 is 3+4.

    no
    ```

- `t` is not a gae.

  Then a run-time error arises. For example,

  ```
  | ?- X is Y+1.

  ! Error in arithmetic expression: not a number
  ```

As an example of the use of an arithmetic evaluator consider the following program that allows us to generate all the integer values in a given range.

```
% between(X, Y, Z) :- X, Y are gaes and Z is an integer between X and Y inclusive.
between(X, Y, Z) :- X =< Y, Z is X.
between(X, Y, Z)  :- X < Y, X1 is X+1, between(X1, Y, Z).
```

For example:

```
| ?- between(10, 14, Z).

Z = 10 ;

Z = 11 ;
```

```
Z = 12 ;

Z = 13 ;

Z = 14 ;

no
```

Note the use of a local variable `X1` in the arithmetic evaluator `X1 is X+1` to compute the increment of `X`. Such use of a local variable is typical for computing using integers in Prolog.

4.4 Concluding Remarks

We introduced in this section the comparison relations on ground arithmetic expressions and the arithmetic evaluator `is`. Let us now try to assess these Prolog features.

4.4.1 Comparison Relations

Because of the possibility of errors, the use of arithmetic expressions in Prolog is quite cumbersome and can easily lead to problems. Suppose for example that we wish to consider natural numbers in the range `[1,100]`. One way to do this is by listing all the relevant facts, so `small_num(1), small_num(2)`, etc. This is hardly a meaningful way of programming. It is more natural to define what constitutes a desired number by means of arithmetic expressions. Thus, we naturally define

```
small_num(X) :- 1 =< X, X =< 100.
```

Unfortunately, these two definitions are not equivalent. For example, with the first definition of `small_num` the query `small_num(X), X < 10` produces all numbers smaller than `10` whereas with the second definition an error arises, because of the improper use of the built-in relation `=<`. In fact, one needs to use here the more complicated program `BETWEEN`, defined in the previous subsection.

As another example of the complications suppose that we wish to produce all pairs of natural numbers `X, Y` such that $\mathtt{X} + \mathtt{Y} = 3$. Both queries `X >= 0, Y >= 0, X + Y =:= 3` and `X + Y =:= 3, X >= 0, Y >= 0` are incorrect and we actually need to use a more complicated and artificial query `between(0,3,X), Y is 3-X`.

4.4.2 Arithmetic Evaluator

Ground arithmetic expressions can be evaluated only using the arithmetic evaluator `is`. However, its use can also easily cause a run-time error. Moreover, the appropriate use of `is` in specific programs, like `BETWEEN`, is quite subtle because it relies on the introduction of fresh variables for holding intermediate results. This proliferation of local variables makes an understanding of such programs more difficult. In imperative programming languages the reuse of the same variables in computation can be seen in such circumstances as an advantage. In functional programming the corresponding functions can be programmed in a much more natural way.

We conclude that arithmetic facilities in Prolog are quite subtle and require good insights to be properly used.

References

[CW93] M. Carlsson and J. Widén. *SICStus Prolog User's Manual.* P.O. Box 1263, S-164 28 Kista, Sweden, January 1993.

[Hoa62] C.A.R. Hoare. Quicksort. *BCS Computer Journal*, 5(1):10–15, 1962.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574. North-Holland, 1974.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.