The Logic Programming Paradigm: A Tutorial

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
University of Amsterdam

January 25, 2000

1 Introduction

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

The logic programming paradigm substantially differs from other programming paradigms. The reason for it is that it has its roots in automated theorem proving, from which it borrowed the notion of a deduction. What is new is that in the process of deduction some values are computed. When stripped to the bare essentials, this paradigm can be summarized by the following three features:

- values are assigned to variables by means of automatically generated substitutions, called *most general unifiers*,
- the control is provided by a single mechanism: automatic *backtracking*,
- computing takes place over the domain of all terms over some alphabet.

Even such a brief summary shows both the strength and weakness of the logic programming paradigm. Its strength lies in an enormous simplicity and conciseness; its weakness has to do with the restrictions to one control mechanism and one data type.

So this framework has to be modified and enriched to tailor this paradigm to the customary needs of computing, for example by providing the data type of integers with the customary arithmetic operations. This can be done and in fact Prolog and constraint logic programming languages are examples of such a customization of this framework. In what follows we discuss first the logic programming framework and then explain how it can be modified to take into account a computationally meaningful subset of Prolog that deals with arithmetic.

2 Syntax

Syntactic conventions always play an important role in the discussion of any programming paradigm and logic programming is no exception in this matter. In this section we discuss both the syntax of LP and of Prolog.

2.1 Logic Programming Syntax

We begin by introducing an alphabet that 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: , .

We also postulate that 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 $f, g, h, \ldots$.

Finally, *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*. We now 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, all possibly with subscripts. By an *expression* we mean any syntactic entity mentioned so far, e.g., a term, atom, clause or a program.

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 $H \leftarrow A_1, \ldots, A_n$ as $A_1 \wedge \ldots \wedge A_n \rightarrow H$. The use of reversed implication is motivated by the procedural interpretation according to which, given the atom $p(\mathbf{s})$, the clause $p(\mathbf{s}) \leftarrow A_1, \ldots, A_n$ 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.

2.2 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. In particular, Prolog 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. That is why we talk about anonymous variables and not about the anonymous 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 (see Carlsson & Widén (1993)) or ECL^{i}PS^e (see A. Aggoun et al. (1995)) encourage the use of anonymous variables by issuing a warning if a non-anonymous variable is encountered that occurs only once in a clause.

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. In the sequel only the logic programming meaning of "atom" will be used.

Next, a Prolog program is viewed as a *sequence* and not as a set of clauses. Finally, the symbol "%" indicates the beginning of a comment line.

In what follows we adhere to Prolog's notation while presenting Prolog programs but when discussing semantics of programs 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 symbols and for 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.

More importantly, ambivalent syntax together with Prolog's facility to declare binary function and relation symbols as infix operators, allows us to pass queries, clauses and programs as arguments. This facilitates *meta-programming*, that is, writing programs that use other programs as data.

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. This ensures that the built-in relations cannot be redefined. Thus one can rely on their prescribed meaning. In more modern versions of Prolog, like SICStus Prolog and ECL^{i}PS^e, a warning is issued in case such an attempt at redefining a built-in relation is encountered.

2.3 Prolog: How to Run it

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 program read and reports an answer. The system replies are of two forms. If the query succeeds, an answer is

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

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

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

mother_of(sarah,isaac).

daughter_of(X,Y) :- father_of(Y,X), female(X).

son_of(X,Y) :- father_of(Y,X), male(X).
```

Figure 1: The `BIBLICAL FAMILY` program

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). 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 a request 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.

2.4 First Example

As a first example of a Prolog program consider the program given in Figure 1 and taken from Sterling & Shapiro (1986) (in fact ftp-ed from ftp.cwru.edu, directory ArtOfProlog).

Example .1 *Biblical family relationships.*

Here the relation `father_of(X,Y)` is to be read "`X` is the father of `Y`" and similarly for `son_of(X, 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 analogously for the other facts and rules.

Even though we still need to 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_of(haran,lot).

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

no (more) solution.
```

Next, it can be used to compute one or more solutions, like in this query that asks for all sons of `terach`:

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

X = abraham     More? (;)

X = haran     More? (;)

X = nachor
yes
```

"`More?  (;)`" is ECL^{i}PS^e prompt for requesting more solutions.

Further, this program 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 5]: son_of(X, terach), father_of(X,_).

X = abraham     More? (;)

X = haran     More? (;)

X = haran     More? (;)

X = haran     More? (;)

no (more) solution.
```

We shall explain later why repeated answers are produced here. Note that the value of the anonymous variable "_" is not printed. □

2.5 Lists in Prolog

A data structure that supports use of sequences with a single operation on them — an insertion of an element at the front — is usually called a *list*. Lists form a fundamental

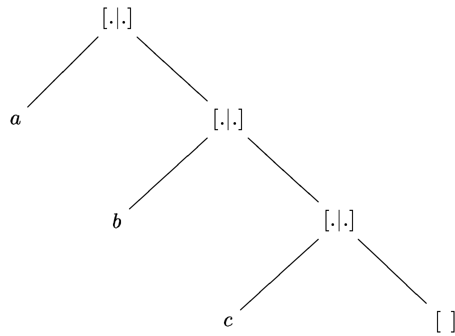

Figure 2: A list

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:

- `[]` is a list, called the *empty list*.
- if `t` is a list, then for any term `h` also `[h | t]` is a list; `h` is called its *head* and `t` is called its *tail*.

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 2 represents the list `[a|[b|[c|[]]]]`.

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

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

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]`.

The following interaction with a Prolog system shows that these simplifications are also carried out internally.

```
[eclipse 1]:  X = [a | [b | c]].

X = [a, b|c]
yes.
[eclipse 2]: [a,b|c]  = [a | [b | c]].

yes.
[eclipse 3]:  X = [a | [b, c | []]].

X = [a, b, c]
yes.
```

```
% 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).
```

Figure 3: The APPEND Program

```
[eclipse 4]: [a,b,c] = [a | [b, c | []]].

yes.
```

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

Example .2 APPEND *program.*

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 the program given in Figure 3.

APPEND can be used not only to concatenate the lists:

```
[eclipse 2]: app([mon, wed], [fri, sun], Zs).

Zs = [mon, wed, fri, sun]     More? (;)

no (more) solution.
```

but also to split a list in all possible ways:

```
[eclipse 3]: app(Xs, Ys, [mon, wed, fri, sun]).

Xs = []
Ys = [mon, wed, fri, sun]     More? (;)

Xs = [mon]
Ys = [wed, fri, sun]     More? (;)

Xs = [mon, wed]
Ys = [fri, sun]     More? (;)
```

```
Xs = [mon, wed, fri]
Ys = [sun]     More? (;)

Xs = [mon, wed, fri, sun]
Ys = []     More? (;)

no (more) solution.
```

□

3 The Computation Process

Let us turn now our attention to the most crucial aspect, that of the computational interpretation of logic programs. 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. This is in contrast to imperative programming languages, such as Pascal or C, in which variables represent known but varying quantities.

The values assigned to variables are terms. These values are assigned by means of substitutions that we now explain.

3.1 Substitutions and Most General Unifiers

Consider a fixed alphabet and consequently a fixed set 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 *simultaneously replaced by* $t_1, \ldots, t_n$, respectively. When $n = 0$, the mapping becomes the empty mapping. The resulting substitution is then called *empty substitution* and is denoted by ϵ.

A substitution θ that is a 1-1 and onto mapping from its domain to itself, is called a *renaming*. For example $\{x/y, y/z, z/x\}$ is a renaming while $\{x/y, y/z, z/y\}$ not.

An expression $E\theta$ is called an *instance* of E and if θ is a renaming, then $E\theta$ is called a *variant* of E. For example $p(x, z)$ is a variant of $p(z, y)$ because $p(z, y)\{x/y, y/z, z/x\} = p(x, z)$ and $\{x/y, y/z, z/x\}$ is a renaming.

Further, given a substitution $\theta := \{x_1/t_1, \ldots, x_n/t_n\}$, we denote by $Dom(\theta)$ the set of variables $\{x_1, \ldots, x_n\}$ and by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$.

We now define the result of *applying a substitution θ to an expression E*, written as $E\theta$, as the result of the simultaneous replacement of each occurrence in E of a variable from $Dom(\theta)$ by the corresponding term in $Range(\theta)$.

Example .3 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

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

□

Next, we define the composition of two substitutions.

Definition .4 Let θ and η be substitutions. Their *composition*, written as $\theta\eta$, is defined as follows. We put for a variable x

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

In other words, $\theta\eta$ assigns to a variable x the term obtained by applying the substitution η to the term $x\theta$. Clearly, for $x \notin Dom(\theta) \cup Dom(\eta)$ we have $(\theta\eta)(x) = x$, so $\theta\eta$, limited to the variables on which it is not an identity, is a finite mapping from variables to terms, i.e. $\theta\eta$ uniquely identifies a substitution. □

For example, for $\theta = \{u/z, x/3, y/f(x, 1)\}$ and $\eta = \{x/4, z/u\}$ we can check that

$$\theta\eta = \{x/3, y/f(4, 1), z/u\}.$$

Next, we introduce the following notion.

Definition .5 Let θ and τ be substitutions. We say that θ is *more general than* τ if for some substitution η we have $\tau = \theta\eta$. □

For example, taking the above substitutions we note that $\{u/z, x/3, y/f(x, 1)\}$ is more general than $\{x/3, y/f(4, 1), z/u\}$, since we have $\{u/z, x/3, y/f(x, 1)\}\{x/4, z/u\} = \{x/3, y/f(4, 1), z/u\}$.

Also, the substitution $\{x/y\}$ is more general than $\{x/a, y/a\}$ since $\{x/y\}\{y/a\} = \{x/a, y/a\}$.

Note that θ is more general than τ if τ can be obtained from θ by applying to it some substitution η. Since η can be chosen to be the empty substitution ϵ, we conclude that every substitution is more general than itself.

We conclude this string of definitions by introducing the crucial notion of a most general unifier.

Definition .6 Consider a pair of atoms A and B. A substitution θ is called a *unifier* of A and B if $A\theta = B\theta$. If a unifier θ of A and B exists, then we say that A and B are *unifiable*.

A unifier θ of A and B is called a *most general unifier* (in short *mgu*) of A and B if it 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 of A and B iff every unifier of A and B is of the form $\theta\gamma$ for some substitution γ.

Consider some examples.

Example .7

(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 $p(y, b)$.

(ii) Consider the atoms $p(g(x, a), z)$ and $p(g(x, b), b)$. They have no unifier as for no substitution θ we have $a\theta = b\theta$.

(iii) Finally consider the atoms $p(x, a)$ and $p(f(x), a)$. They have no unifier either because for any substitution θ the term $x\theta$ is a proper substring of $f(x)\theta$. □

The problem of deciding whether a pair of atoms has a unifier is called the *unification problem*. The following result was established by Robinson (1965).

Theorem .8 (Unification) *An algorithm exists that for a given pair of atoms determines whether they are unifiable. Additionally, if yes, it produces an mgu.* □

3.2 The Computation Process of Logic Programs

Unification, i.e., the process of computing most general unifiers, 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. Further, these most general unifiers are automatically generated during the computation process, so — in contrast to the imperative programming— the assignment of values to variables takes place implicitly.

This operational semantics of logic programs is called a *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 approach 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).

To define the procedural interpretation we now consider pairs of the form $\langle Q \ ; \ \theta\rangle$, where Q is a query with, if it is non-empty, one atom selected in it 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 the " $\rightarrow$ " 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 and a clause c from P. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint with $\langle \mathbf{A}, B, \mathbf{C} \; ; \; \theta \rangle$, that is such that

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

where $Var(E)$ denotes the set of variables occurring in an expression E and $Var(\theta)$ the set of all variables used in θ.

Suppose now 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

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

(The origin of the "SLD" abbreviation will be explained at the end of the chapter.) So $\langle \mathbf{A}, \mathbf{B}, \mathbf{C} \; ; \; \theta\eta \rangle$ is obtained from $\langle \mathbf{A}, B, \mathbf{C} \; ; \; \theta \rangle$ by

- replacing B by $\mathbf{B}$,
- applying η to θ.

By repeating the " $\rightarrow$ " steps we obtain derivations that are tantamount to computations. More precisely, consider a program P and a query Q. By an *SLD-derivation* of Q w.r.t. P we mean a maximal sequence of pairs $\langle Q_i \; ; \; \theta_i \rangle$, where $i \geq 0$, such that for all $j \geq 0$ we have $\langle Q_j \; ; \; \theta_j \rangle \rightarrow \langle Q_{j+1} \; ; \; \theta_{j+1} \rangle$.

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 " $\rightarrow$ " (i.e., " $\rightarrow^*$ " 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.

3.3 The Computation Process of 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, that we call *computation 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 a computation tree

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

In the computation tree the backtracking search for computed answer substitutions takes place. If in this search a leaf with the empty query is encountered, then the associated computed answer substitution is printed as a system of equations and the search is suspended. The request for more solutions (";" of Subsection 2.3) results in a resumption of the search from the last visited leaf until a new leaf with the empty query is visited. If the tree has no (respectively, no more) leaves with the empty query, then failure is reported, by printing the answer "`no`" (respectively, `‘‘no (more) solution’’`).

Figure 4 depicts the backtracking search over such a tree.

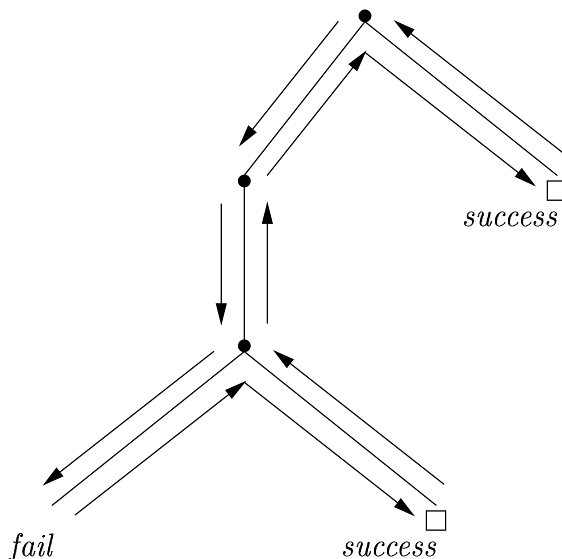

Figure 4: Backtracking over a computation tree

The reader can now manually simulate (though it is a tedious exercise) the results of the Prolog computations for the queries considered in Subsections 2.4 and 2.5. In particular, the query `son_of(X, terach), father_of(X,_)` succeeds in three possible ways because the direct descendant

$$\langle \texttt{father_of(haran,_)} \;;\; \{\texttt{X/haran}\} \rangle$$

of

$$\langle \texttt{son_of(X,terach),father_of(X,_)} \;;\; \epsilon \rangle$$

succeeds in three possible ways, since `terach` has three sons. Prolog reports each success separately disregarding the possibility that some of the answers are identical.

3.4 Power of Backtracking

The power of Prolog lies in automatic support for backtracking. To illustrate its use we consider Example 2.11 of the Principles of Constraint Programming Lecture Notes. We provide here a solution in Prolog to the problem of finding all legal labellings of the cube scene given there in Figure 2.7.

The program `CUBE` given in Figure 5 is practically a verbatim formalization of the constraints used. The query `cube(AC, AE, AB, BF, BD, CD, DG, EF, FG)` computes all legal labellings of the cube. In total there are 4 solutions. The first one, listed below, corresponds to the one given in Figure 2.10 of the lecture notes.

```
[eclipse 3]: cube(AC, AE, AB, BF, BD, CD, DG, EF, FG).

AC = l
AE = r
AB = +
BF = +
BD = +
CD = l
DG = l
EF = r
FG = r
```

4 Arithmetic in Prolog

4.1 Arithmetic Operators

Prolog provides integers and floating point numbers as built-in data structures. So we have infinitely many integer constants:

$$0,\ -1,\ 1,\ -2,\ 2,\ \ldots$$

and infinitely many floating point numbers, such as

$$0.0,\ -1.72,\ 3.141,\ -2.0,\ \ldots$$

Floating point numbers are the computer representable "reals" and the number of digits allowed after the obligatory decimal point "." depends on the implementation. In what follows by a *number* we mean either an integer constant or a floating point number.

Prolog provides various operations on integers and floating point numbers. These operations include the following binary operations:

```
% the L constraint
l(r,l).
l(l,r).
l(+,r).
l(l,+).
l(-,l).
l(r,-).

% the fork constraint
fork(+,+,+).
fork(-,-,-).
fork(l,r,-).
fork(-,l,r).
fork(r,-,l).

% the T constraint
t(r,l,r).
t(r,l,l).
t(r,l,+).
t(r,l,-).

% the arrow constraint
arrow(l,r,+).
arrow(+,+,-).
arrow(-,-,+).

% the edge constraint
edge(+,+).
edge(-,-).
edge(r,l).
edge(l,r).

% the cube relation computes legal labelling of the cube
% scene given in Figure 2.7.
cube(AC, AE, AB, BF, BD, CD, DG, EF, FG) :-
    arrow(AC,AE,AB), fork(BA,BF,BD),  l(CA,CD),
    arrow(DG,DC,DB), l(EF,EA), arrow(FE,FG,FB),
    l(GD,GF),
    edge(AB,BA), edge(AC,CA), edge(CD,DC),
    edge(BD,DB), edge(AE,EA), edge(EF,FE),
    edge(BF,FB), edge(FG,GF), edge(DG,GD).
```

Figure 5: The CUBE program

- 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. Moreover, 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 $y \neq 0$, `x mod y` is defined as `x - y*(x//y)`.

The arithmetic operators and the above introduced set of numbers uniquely determine a set of terms. We call terms defined in this language *arithmetic expressions* and introduce the abbreviation *gae* for ground arithmetic expressions.

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 the following six *arithmetic comparison relations* (in short *comparison relations*):

- "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` that in Prolog denotes the "is unifiable" relation (internally defined by the single clause `X = X.` .)

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

```
[eclipse 7]: 5*2 > 3+4.

yes

[eclipse 8]: 7 > 3+4.

no (more) solution.
```

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

```
[eclipse 9]: [] < 5.
undefined arithmetic expression _59 is [] in module eclipse
```

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 program in Figure 6 that checks whether a list is an ordered one.

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

Figure 6: The `ORDERED` Program

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 at certain stage the comparison relation `=<` was applied to an argument that is not a number.

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 (1962). 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 by means of the program in Figure 7, 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.

Figure 7: The `QUICKSORT` Program

For example:

```
[eclipse 4]: qs([7,9,8,1,5], Ys).

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

no (more) solution.
```

However, we also have

```
[eclipse 5]: qs([3,X,0,1], Ys).
instantiation fault in 3 > X
```

because during the computation the comparison relation $>$ is applied to non-gae arguments.

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. 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 and ECL^{i}PS^e:

```
:- 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,

    ```
    [eclipse 11]: 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,

    ```
    [eclipse 12]: X is 3+4.

    X = 7
    yes.
    ```

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

    Then the arithmetic evaluator fails if `s` is a number and otherwise a run-time error arises. For example,

    ```
    [eclipse 13]: 8 is 3+4.

    no (more) solution.

    [eclipse 14]: 3+4 is 3+4.
    number expected in +(3, 4, 3 + 4)
    ```

- `t` is not a gae.

  Then a run-time error arises. For example,

  ```
  [eclipse 15]:  X is Y+1.
  instantiation fault in +(Y, 1, X)
  ```

As an example of the use of an arithmetic evaluator consider the program `LENGTH` that computer the length of a list. (Actually `length` is a built-in of Prolog.)

```
% length(Xs, N) :- N is the length of the list Xs.
length([], 0).
length([_ | Ts], N) :- length(Ts, M), N is M+1.
```

Figure 8: The `LENGTH` program

In turn, in Figure 9 we listed a more involved program that by means of backtracking generates 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).
```

Figure 9: The `BETWEEN` program

For example:

```
[eclipse 17]: between(10, 14, Z).

Z = 10     More? (;)

Z = 11     More? (;)

Z = 12     More? (;)

Z = 13     More? (;)

Z = 14     More? (;)

no (more) solution.
```

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 $X + Y = 3$. 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.

5 Operators (Optional)

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 and ECL^{i}PS^e 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 and ECL^{i}PS^e):

```
:- 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.

It is worthwhile mentioning that Prolog built-in operators, in particular arithmetic operators, can also be written in the customary prefix notation. 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

```
[eclipse 6]: 4+3*5 =:= +(4, *(3,5)).

yes
```

6 Bibliographic Remarks

Use of unification for computing is due to Kowalski (1974). The definition of SLD-derivations we used here is due to T. Nipkow (unpublished). *SLD*-resolution stands for *S*election rule driven *L*inear resolution for *D*efinite clauses. Linearity means that each resolvent depends only on the previous one, so that derivations become sequences. Definite clauses are clauses in our terminology.

The elegant program `CUBE` given in Figure 5 is taken from By (1997). We discussed here the subject of Prolog only very briefly. Two most often used books on Prolog are Bratko (1986) and Sterling & Shapiro (1986). The book of Clocksin & Mellish (1984) explains various subtle points of the language and the book of O'Keefe (1990) discusses in depth the efficiency and pragmatics of programming in Prolog. In Apt (1997) a systematic introduction to Prolog via logic programming is offered. Another, very concise, book on Prolog is Clocksin (1997).

References

A. Aggoun et al. (1995), *ECL^{i}PS^e 3.5 User Manual*, Munich, Germany.

Apt, K. R. (1997), *From Logic Programming to Prolog*, Prentice-Hall, London, U.K.

Bratko, I. (1986), *PROLOG Programming for Artificial Intelligence*, International Computer Science Series, Addison-Wesley.

By, T. (1997), Line labelling by meta-programming, Technical Report CS-97-07, University of Sheffield.

Carlsson, M. & Widén, J. (1993), *SICStus Prolog User's Manual*, P.O. Box 1263, S-164 28 Kista, Sweden.

Clocksin, W. & Mellish, C. (1984), *Programming in Prolog*, second edn, Springer-Verlag, Berlin.

Clocksin, W. F. (1997), *Clause and Effect*, Springer-Verlag, Berlin.

Hoare, C. (1962), 'Quicksort', *BCS Computer Journal* **5**(1), 10–15.

Kowalski, R. (1974), Predicate logic as a programming language, *in* 'Proceedings IFIP'74', North-Holland, pp. 569–574.

O'Keefe, R. (1990), *The Craft of Prolog*, MIT Press.

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

Sterling, L. & Shapiro, E. (1986), *The Art of Prolog*, MIT Press.