Part I

Logic programming paradigm

1

Logic programming and pure Prolog

1.1 Introduction

LOGIC PROGRAMMING (LP in short) is a simple yet powerful formalism suitable for computing and for knowledge representation. It provides a formal basis for ***Prolog*** and for ***constraint logic programming***. Other successful applications of LP include ***deductive databases***, an extension of relational databases by rules, a computational approach to machine learning called ***inductive logic programming***, and a computational account of various forms of reasoning, in particular of ***non-monotonic reasoning*** and ***meta-reasoning***.

The ***logic programming paradigm*** substantially differs from other programming paradigms. The reason for this 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 summarised by the following three features:

- any variable can stand for a number or a string, but also a list, a tree, a record or even a procedure or program,

- during program execution variables are constrained, rather than being assigned a value and updated,
- program executions include choice points, where computation can take alternative paths; if the constraints become inconsistent, the program backtracks to the previous open choice point and tries another path.

In this chapter we discuss the logic programming framework and the corresponding small subset of Prolog, usually called ***pure Prolog***. This will allow us to set up a base over which we shall define in the successive chapters a more realistic subset of Prolog supporting in particular arithmetic and various control features. At a later stage we shall discuss various additions to Prolog provided by ECL^{i}PS^e, including libraries that support constraint programming.

We structure the chapter by focussing in turn on each of the above three items. Also we clarify the intended meaning of pure Prolog programs.[1] Consequently, we discuss in turn

- the objects of computation and their syntax,
- the meaning of pure Prolog programs,
- the accumulation of constraints during program execution, and
- the creation of choice points during program execution and backtracking.

1.2 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 the syntax of Prolog.

Full Prolog syntactic conventions are highly original and very powerful. Their full impact is little known outside of the logic programming community. We shall discuss these novel features in Chapters 3 and 4.

By the 'objects of computation' we mean anything that can be denoted by a Prolog variable. These are not only numbers, lists and so on, but also compound structures and even other variables or programs.

Formally, the objects of computation are ***base terms***, which consists of:

- ***variables***, denoted by strings starting with an upper case letter or '_' (the underscore), for example `X3` is a variable,
- ***numbers***, which will be dealt with in Chapter 3,

[1] To clarify the terminology: ***logic programs*** are pure Prolog programs written using the logical and not Prolog notation. In what follows we rather focus on Prolog notation and, as a result, on pure Prolog programs.

- ***atoms***, denoted by sequences of characters starting with a lower case letter, for example `x4` is an atom. Any sequence of characters put between single quotes, even admitting spaces, is also an atom, for example `'My Name'`,

and ***compound terms***, which comprise a ***functor*** and a number of ***arguments***, each of which is itself a (base or compound) ***term***.

By a ***constant*** we mean a number or an atom. Special case of terms are ***ground terms*** which are terms in which no variable appears. In general, the qualification 'ground', which we will also use for other syntactic objects, simply means 'containing no variables'.

In Chapter 4 we shall explain that thanks to the ***ambivalent syntax*** facility of Prolog programs can also be viewed as compound terms. This makes it possible to interpret ***programs as data***, which is an important feature of Prolog.

In the standard syntax for compound terms the functor is written first, followed by the arguments separated by commas, and enclosed in round brackets. For example the term with functor `f` and arguments `a`, `b` and `c` is written `f(a,b,c)`. Similarly, a more complex example with functor `h` and three arguments:

(i) the variable `A`,
(ii) the compound term `f(g,'Twenty',X)`,
(iii) the variable `X`,

is written `h(A,f(g,'Twenty',X),X)`.

Some compound terms can be also written using an infix notation. Next, we define goals, queries, clauses and programs. Here is a preliminary overview.

- A program is made up of procedures.
- A procedure is made up of clauses, each terminated by the period '.'.
- A ***clause*** is either a fact or a rule.
- There is no special procedure such as `main` and a user activates the program by means of a query.

Now we present the details.

- First we introduce an ***atomic goal*** which is the basic component from which the clauses and queries are built. It has a ***predicate*** and a number of arguments. The predicate has a ***predicate name*** and a ***predicate arity***. The predicate name is, syntactically, an atom. The arguments are placed after the predicate name, separated by commas and surrounded by round brackets. The number of arguments is the arity of the predicate.

An example of an atomic goal is `p(a,X)`. Here the predicate name is `p` and its arity is 2. We often write `p/n` for the predicate with name `p` and arity n.

- A ***query*** (or a ***goal***) is a sequence of atomic goals terminated by the period '.'. A query is called ***atomic*** if it consists of exactly one atomic goal.
- A ***rule*** comprises a ***head***, which is an atomic goal, followed by '`:-`', followed by the ***body*** which is a non-empty query, and is terminated by the period '.'.

  An example of a rule is

  ```
  p(b,Y) :- q(Y), r(Y,c).
  ```

  A rule contributes to the definition of the predicate of its head, so this example rule contributes to the definition of `p/2`. We also call this a rule for `p/2`. Rule bodies may contain some atomic goals with a binary predicate and two arguments written in the infix form, for example `X = a`, whose predicate is `=/2`.
- A ***fact*** is an atomic goal terminated by the period '.'. For example

  ```
  p(a,b).
  ```

  is a fact. This fact also contributes to the definition of `p/2`.
- A sequence of clauses for the same predicate makes a ***procedure***. The procedure provides a ***definition*** for the predicate.

  For example, here is a definition for the predicate `p/2`:

  ```
  p(a,b).
  p(b,Y) :- q(Y), r(Y,c).
  ```

- A ***pure Prolog program*** is a finite sequence of procedures, for example

  ```
  p(a,b).
  p(b,Y) :- q(Y), r(Y,c).
  q(a).
  r(a,c).
  ```

  So a program is simply a sequence of clauses.

In what follows, when discussing specific queries and rules in a running text we shall drop the final period '.'.

Before leaving this discussion of the syntax of pure Prolog, we point out a small but important feature of variable naming. If a variable appears more than once in a query (or similarly, more than once in a program clause), then

each occurrence denotes the *same* object of computation. However, 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. That is why we talk about the anonymous variable*s* and not about the anonymous variable. So by definition each anonymous variable occurs in a query or a clause only once.

Anonymous variables form a simple and elegant device and, as we shall see, their use increases the readability of programs in a remarkable way. ECL^{i}PS^e and many modern versions of Prolog encourage the use of anonymous variables by issuing a warning if a non-anonymous variable is encountered that occurs only once in a clause. This warning can be suppressed by using a normal variable that starts with the underscore symbol '_', for example `_X`.

Prolog has several (about one hundred) built-in predicates, so predicates with a predefined meaning. The clauses, the heads of which refer to these built-in predicates, are ignored. This ensures that the built-in predicates cannot be redefined. Thus one can rely on their prescribed meaning. In ECL^{i}PS^e and several versions of Prolog a warning is issued in case an attempt at redefining a built-in predicate is encountered.

So much about Prolog syntax for a moment. We shall return to it in Chapters 3 and 4 where we shall discuss several novel and powerful features of Prolog syntax.

1.3 The meaning of a program

Pure Prolog programs can be interpreted as statements in the first-order logic. This interpretation makes it easy to understand the behaviour of a program. In particular, it will help us to understand the results of evaluating a query w.r.t. to a program. In the remainder of this section we assume that the reader is familiar with the first-order logic.

Let us start by interpreting a simple program fact, such as `p(a,b)`. It contributes to the definition of the predicate `p/2`. Its arguments are two atoms, `a` and `b`.

We interpret the predicate `p/2` as a relation symbol p of arity 2. The atoms `a` and `b` are interpreted as logical constants a and b. A logical constant denotes a value. Since, in the interpretation of pure Prolog programs, different constants denote different values, we can think of each constant denoting itself. Consequently we interpret the fact `p(a,b)` as the atomic formula $p(a, b)$.

The arguments of facts may also be variables and compound terms. Consider for example the fact `p(a,f(b))`. The interpretation of the compound term `f(b)` is a logical expression, in which the unary function f is applied to the logical constant b. Under the interpretation of pure Prolog programs, the denotations of any two distinct ground terms are themselves distinct.[2] Consequently we can think of ground terms as denoting themselves, and so we interpret the fact `p(a,f(b))` as the atomic formula $p(a, f(b))$.

The next fact has a variable argument: `p(a,Y)`. We view it as a statement that for all ground terms t the atomic formula $p(a, t)$ is true. So we interpret it as the universally quantified formula $\forall Y.\, p(a, Y)$.

With this interpretation there can be no use in writing the procedure

```
p(a,Y).
p(a,b).
```

because the second fact is already covered by the first, more general fact.

Finally we should mention that facts with no arguments are also admitted. Accordingly we can assert the fact `p`. Its logical interpretation is the proposition p.

In general, we interpret a fact by simply changing the font from `teletype` to *italic* and by preceding it by the universal quantification of all variables that appear in it.

The interpretation of a rule involves a logical ***implication***. For example the rule

```
p :- q.
```

states that if q is true then p is true.

As another example, consider the ground rule

```
p(a,b) :- q(a,f(c)), r(d).
```

Its interpretation is as follows. If $q(a, f(c))$ and $r(d)$ are both true, then $p(a, b)$ is also true, i.e., $q(a, f(c)) \wedge r(d) \rightarrow p(a, b)$.

Rules with variables need a little more thought. The rule

```
p(X) :- q.
```

states that if q is true, then $p(t)$ is true for any ground term t. So logically this rule is interpreted as $q \rightarrow \forall X.\, p(X)$. This is equivalent to the formula $\forall X.\, (q \rightarrow p(X))$.

If the variable in the head also appears in the body, the meaning is the same. The rule

[2] This will no longer hold for arithmetic expressions which will be covered in Chapter 3.

```
p(X) :- q(X).
```

states that for any ground term t, if $q(t)$ is true, then $p(t)$ is also true. Therefore logically this rule is interpreted as $\forall X.\,(q(X) \rightarrow p(X))$.

Finally, we consider rules in which variables appear in the body but not in the head, for example

```
p(a) :- q(X).
```

This rule states that if we can find a ground term t for which $q(t)$ is true, then $p(a)$ is true. Logically this rule is interpreted as $\forall X.\,(q(X) \rightarrow p(a))$, which is equivalent to the formula $(\exists X.\,q(X)) \rightarrow p(a)$.

Given an atomic goal A denote its interpretation by $\tilde{A}$. Any ground rule H `:-` $B_1, \ldots, B_n$ is interpreted as the implication $\tilde{B}_1 \wedge \ldots \wedge \tilde{B}_n \rightarrow \tilde{H}$. In general, all rules H `:-` $B_1, \ldots, B_n$ have the same, uniform, logical interpretation. If $\mathbf{V}$ is the list of the variables appearing in the rule, its logical interpretation is $\forall \mathbf{V}.\,(\tilde{B}_1 \wedge \ldots \wedge \tilde{B}_n \rightarrow \tilde{H})$.

This interpretation of ',' (as $\wedge$) and ':-' (as $\rightarrow$) leads to so-called ***declarative interpretation*** of pure Prolog programs that focusses – through their translation to the first-order logic – on their semantic meaning.

The computational interpretation of pure Prolog is usually called ***procedural interpretation***. It will be discussed in the next section. In this interpretation the comma ',' separating atomic goals in a query or in a body of a rule is interpreted as the semicolon symbol ';' of the imperative programming and ':-' as (essentially) the separator between the procedure header and body.

1.4 Computing with equations

We defined in Section 1.2 the computational objects over which computations of pure Prolog programs take place. The next step is to explain how variables and computational objects become constrained to be equal to each other, in the form of the answers. This is the closest that logic programming comes to assigning values to variables.

1.4.1 Querying pure Prolog programs

The computation process of pure Prolog involves a program P against which we pose a query Q. This can lead to a successful, failed or diverging computation (which of course we wish to avoid).

A successful computation yields an *answer*, which specifies constraints on the query variables under which the query is true. In this subsection we

describe how these constraints are accumulated during query processing. In the next subsection we will describe how the constraints are checked for consistency and answers are extracted from them.

The constraints accumulated by Prolog are equations whose conjunction logically entails the truth of the query. To clarify the discussion we will use the following simple program:

```
p(X) :- q(X,a).
q(Y,Y).
```

Let us first discuss the answer to the atomic query

```
q(W,a).
```

The definition of the predicate `q/2` comprises just the single fact `q(Y,Y)`. Clearly the query can only succeed if `W = a`.

Inside Prolog, however, this constraint is represented as an equation between two atomic goals: `q(W,a) = q(Y1,Y1)`. The atomic goal `q(W,a)` at the left-hand side of the equation is just the original query. For the fact `q(Y,Y)`, however, a new variable `Y1` has been introduced. This is not important for the current example, but it is necessary in general because of possible variable clashes. This complication is solved by using a different variable each time. Accordingly, our first query succeeds under the constraint `q(W,a) = q(Y1,Y1)`.

Now consider the query

```
p(a).
```

This time we need to use a rule instead of a fact. Again a new variable is introduced for each use of the rule, so this first time it becomes:

```
p(X1) :- q(X1,a).
```

To answer this query, Prolog first adds the constraint `p(a) = p(X1)`, which constrains the query to match the definition of `p/1`. Further, the query `p(a)` succeeds only if the body `q(X1,a)` succeeds, which it does under the additional constraint `q(X1,a) = q(Y1,Y1)`. The complete sequence of constraints under which the query succeeds is therefore `p(a) = p(X1), q(X1,a) = q(Y1,Y1)`. Informally we can observe that these constraints hold if all the variables take the value `a`.

Consider now the query:

```
p(b).
```

Reasoning as before, we find the query would succeed under the constraints: `p(b) = p(X1), q(X1,a) = q(Y1,Y1)`. In this case, however, there are no possible values for the variables which would satisfy these constraints. `Y1` would have to be equal both to `a` and to `b`, which is impossible. Consequently the query fails.

Next consider a non-atomic query

```
p(a), q(W,a).
```

The execution of this query proceeds in two stages. First, as we already saw, `p(a)` succeeds under the constraints `p(a) = p(X1), q(X1,a) = q(Y1,Y1)`, and secondly `q(W,a)` succeeds under the constraint `q(W,a) = q(Y2,Y2)`. The complete sequence of constraints is therefore: `p(a) = p(X1), q(X1,a) = q(Y1,Y1), q(W,a) = q(Y2,Y2)`. Informally these constraints are satisfied if all the variables take the value `a`.

A failing non-atomic query is:

```
p(W), q(W,b).
```

Indeed, this would succeed under the constraints `p(W) = p(X1), q(X1,a) = q(Y1,Y1), q(W,b) = q(Y2,Y2)`. However, for this to be satisfied `W` would have to take both the value `a` (to satisfy the first two equations) and `b` (to satisfy the last equation), so the constraints cannot be satisfied and the query fails.

Operationally, the constraints are added to the sequence during computation, and tested for consistency immediately. Thus the query

```
p(b), q(W,b).
```

fails already during the evaluation of the first atomic query, because already at this stage the accumulated constraints are inconsistent. Consequently the second atomic query is not evaluated at all.

1.4.2 Most general unifiers

The Prolog system has a built-in algorithm which can detect whether a sequence of equations between atomic goals is consistent or not. In general, two outcomes are possible. Either the algorithm returns a failure, because some atomic goals cannot be made equal (for example `p(X,a) = p(b,X)`), or it yields a positive answer in the form of a ***substitution***, which is a set of equations of the form *Variable* = *Term* such that each variable occurs at most once at a left-hand side of an equation. The resulting set of equations is called a ***unifier***.