Chapter 12
The Logic Programming Paradigm

In this chapter we analyse the other paradigm which, together with functional programming, supports declarative programming. The logic programming paradigm includes both theoretical and fully implemented languages, of which the best known is surely PROLOG. Even if there are big differences of a pragmatic and, for some, of a theoretical nature between these languages, they all share the idea of interpreting computation as logical deduction.

In this chapter, we will therefore examine these concepts while trying to limit the theoretical part. We also adopt the approach that has characterised the rest of the text while examining this paradigm. We do not mean therefore to teach programming in PROLOG, even if we present various examples of real programs, but we do intend to provide enough basis for understanding and, in a short time, mastering this and other logic programming languages.

12.1 Deduction as Computation

A well-known slogan due to R. Kowalski, exactly captures the concepts that underpin the activity of programming: Algorithm = Logic + Control. According to this "equation", the specification of an algorithm, and therefore its formulation in programming languages, can be separated into three parts. On the one side, the logic of the solution is specified. That is, the "what" must be done is defined. On the other, those aspects related to control are specified, and therefore the "how" of finding the desired solution is clarified. The programmer who uses a traditional imperative language must take account of both these components using the mechanisms which we have variously analysed in the chapters that precede this one. Logic programming, on the other hand, was originally defined with the idea of cleanly separating these two aspects. The programmer is only required, at least in principle, to provide a logical specification. Everything related to control is relegated to the abstract machine. Using a computational mechanism based on a particular deduction rule (resolution), the interpreter searches through the space of possible solutions for the one specified by the "logic", defining in this way the sequence of operations necessary to reach the final result.

The basis for this view of computation as logical deduction can be traced back to the work of K. Gödel and J. Herbrand in the 1930s. In particular, Herbrand anticipated, even in an incompletely formal way, some ideas on the process of unification, which, as we will see, forms the basic computational mechanism of logic programming languages.

It was not until the 1960s that a formal definition (due to A. Robinson) of this process appeared and only ten years later (in the early 1970s) it was realised that formal automatic deduction of a particular kind could be interpreted as a computational mechanism. The first programming languages in the logic programming paradigm were created, among which PROLOG (the name is an acronym for PROgramming in LOGic, for more information on the story of this language, see Sect. 13.4).

Today there are many implemented versions of PROLOG and there exist various other languages in this paradigm (as far as applications are concerned, those including constraints are of particular interest). All of these languages (and PROLOG was the first) allow the use of constructs permitting the specification of control for reasons of efficiency. Since these constructs do not have a direct logical interpretation, they make the semantics of the language more complicated, and cause the loss of part of the purely declarative nature of the logic paradigm. This notwithstanding, we are still dealing with logic programming languages, even including these "impure" aspects, which require the programmer to do little more than formulate (or declare) the specification of the problem to be solved. In some cases, the resulting programs are really surprising in their brevity, simplicity and clarity, as we will see in the next section.

Terminological Note Logic programming, which constitutes the theoretical formalism, is distinguished from PROLOG, a language which has different implementations and for which a standard has recently been defined. The concepts we introduce below, if not otherwise stated, are valid for both formalisms. The important differences will be explicitly pointed out. The programming examples which we include (usually in `this font`) are all PROLOG code which could be run on some implementation of the language. The theoretical concepts, on the other hand, even when they are valid for PROLOG, use mathematical concepts and are rendered in *italic characters*. Moreover, we will follow the PROLOG convention that we always write variables as strings of characters beginning with an upper-case letter. Finally note that, as mentioned before, there exist several other languages, different from PROLOG, in the logic programming paradigm. We call these logic programming languages, or logic languages, for short.

12.1.1 An Example

We try to substantiate what we said in the last section with an example which will be fairly informal, at least for now, given that we have introduced neither the syntax nor the semantics of logic languages.

Let us therefore consider the following problem. It is desired to arrange three 1s, three 2s, ..., three 9s in a list (which therefore will consist of 27 numbers) such that, for each $i \in [1, 9]$, there are exactly i numbers between two successive occurrences (in the list) numbered i. Therefore, for example, 1, 2, 1, 8, 2, 4, 6, 2 could be a part of the final solution, while 1, 2, 1, 8, 2, 4, 2, 6 might not be one (because there is only a single number between the last two occurrences of 2). The reader is invited to try to write a program that solves this problem using their preferred imperative programming language. This is not a difficult exercise but does requires some care, because even if the "what" must be done is clear, the "how" of the desired solution is not immediately clear. For example, in a very naive (and completely inefficient) way, it might be thought that all possible permutations of a list of 27 numbers containing three 1s, three 2s, three 3s, ..., three 9s, could be generated and then checked to see if one of them satisfies the required propriety. This solution, too, probably one of the simplest, however, requires the specification in detail of the aspects of control necessary to generate the permutations of a list.

Reasoning in a declarative fashion, on the other hand, we can proceed as follows. First, we need a list which we will call `Ls`. This list will have to contain 27 elements, something that we can specify using a unary predicate (that is, a relation symbol)[1] `list_of_27`. If we write `list_of_27(Ls)`, we mean, therefore, to say that `Ls` must be a list of 27 elements. In other words, `list_of_27` defines a relation formed of all possible lists of 27 elements. To achieve this aim, we can define `list_of_27` as follows:

```
list_of_27(Ls):-
Ls = [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_].
```

where the part to the left of the `:-` symbol denotes what is being defined, while the part to the right of `:-` indicates the definition. The = denotes an equality concept whose definition is left to intuition, for the present (it is different, however, from assignment). Anticipating the PROLOG notation for lists, here we write `[X1, X2, ...,Xn]` to denote the list that contains the n variables `X1`, `X2`, ..., `Xn` (see Sect. 12.4.5 for more details about lists).

In order to satisfy our specification, the list, `Ls`, in addition to being composed of 27 elements, will have to contain a sublist[2] in which the number 1 appears followed by any other number, by another occurrence of the number 1, then another number and finally a last occurrence of the number 1. Such a sublist can be specified as

[1]Recall that unary means that it has a single argument while n-ary means that there are n arguments.

[2]Recall that `Li` is a sublist of `Ls` if `Ls` is obtained by concatenating a (possibly empty) list with `Li` and with another list (itself also possibly empty).

```
[1,X,1,Y,1]
```

where X and Y are variables. More efficiently, we can write:

```
[1,_,1,_,1]
```

where _ is the *anonymous* variable, that is a variable whose name is of no interest and which is distinct from all other variables present, including all other anonymous variables. Assuming that we have available a binary predicate `sublist(X,Y)` whose meaning is that the first argument (X) is a sublist of the second (Y),[3] our requirement therefore can be expressed by writing:

```
sublist([1,_,1,_,1], Ls)
```

Moving on to number 2 and reasoning in a similar fashion, we obtain that the list Ls must also contain the sublist:

```
[2,_,_,2,_,_2]
```

and therefore the following must also be true:

```
sublist([2,_,_,2,_,_,2], Ls)
```

We can repeat this reasoning as far as number 9, hence the `sol` program that we want to produce can be described as follows:

```
sol(Ls) :-                                                              1
  list_of_27(Ls),
  sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ls),             3
  sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ls),
  sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ls),                     5
  sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ls),
  sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ls),                             7
  sublist([4,_,_,_,_,4,_,_,_,_,4], Ls),
  sublist([3,_,_,_,3,_,_,_,3], Ls),                                     9
  sublist([2,_,_,2,_,_,2], Ls),
  sublist([1,_,1,_,1], Ls).                                             11

list_of_27(Ls):-                                                        13
Ls = [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_].
```

where, as before, the part to the right of the `:-` symbol is the definition of what lies to its left. What we have written above says that to find a solution to our problem (`sol(Ls)` in line 1), all the properties described in lines 2–11 must be satisfied. The commas that separate the various predicates are to be interpreted as "and", that is as conjunctions in the logical sense. What we have provided therefore is nothing

[3]The definition of `sublist` is given in Sect. 12.4.5.

more than a specification that, in substance, formally repeats the formulation of the problem and which, as we will see below, can be interpreted in purely logical terms.

This declarative reading, for all of its elegance and compactness, would not be of great of use as the solution to our problem if it were not accompanied by a procedural interpretation and must therefore be translated into a conventional programming language. The important characteristics of the paradigm that we are considering is precisely that the (logical) specifications we write are to all intents and purposes executable programs. The code described above in lines 1–14 (plus the definition of `sublist` in Sect. 12.4.5) is indeed a genuine PROLOG program that can be evaluated by an interpreter to obtain the desired solution.

In other words, our specification can also be read in a procedural fashion, as follows. Line (1) contains the declaration of a procedure called `sol`; it has a single formal parameter, `Ls`. The body of this procedure is defined by lines (2) to (11), where we find the following ten procedure calls. On line (2), procedure `list_of_27` is called with actual parameter `Ls` which is defined as we have seen above and which therefore instantiates[4] its actual parameter into a list of 27 anonymous variables. Therefore, on line (3), we have the call:

```
sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ls)
```

We assume that this call arranges matters so that the list which appears as the first parameter is a sublist of the list which is bound to the second parameter (`Ls`), possibly by instantiating the variable `Ls`. Similarly for the other calls until line (11). Parameter passing occurs in a way similar to call by name and, in PROLOG, the order in which the different procedure calls appear in the text specifies the order of evaluation (in the case of pure logic programs, on the other hand, no order is specified).

Given the preceding definition of procedure `sol`, the call `sol(Ls)` returns `Ls` instantiated with a solution to the problem, such as for example the following:

```
1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7.
```

Successive calls to the same procedure will allow us also to obtain the other solutions to the problem.

With this procedural interpretation, one has therefore a true programming language which allows us to express, in a compact and relatively simple fashion, programs that solve even very complex problems. For this power, the language pays the penalty of efficiency. In the preceding program, despite its apparent simplicity, the computation performed by the language's abstract machine is very complex, given that the interpreter must try the various combinations of possible sublists until it finds the one that satisfies all the conditions. In these search processes, a *backtracking* mechanism is used. When the computation arrives at a point at which it cannot

[4]Even if the terminology is intuitively clear: by instantiation of a variable is meant substituting for it a syntactic object. We will give a precise definition in Sect. 12.3.

proceed, the computation that has been performed is undone so that a decision point can be reached, if it exists, at which an alternative is chosen that is different form the previous one (if this alternative does not exist, the computation terminates in failure). It is not difficult to see that, in general, this search process can have exponential complexity.

12.2 Syntax

Logic programs are sets of logic formulæ of a particular form. We begin therefore with some basic notions for defining the syntax.

The logic of interest here is first-order logic; it is also called *predicate calculus*. The terminology refers to the fact that symbols are used to express (or, in a more old-fashioned terminology, to "predicate") properties of elements of a fixed domain of discourse, $\mathscr{D}$. More expressive logics (second, third, etc., order) also permit predicates whose arguments are more complicated objects such as sets and functions over $\mathscr{D}$ (second order), sets of functions (third order), etc., in addition to elements of $\mathscr{D}$.

12.2.1 The Language of First-Order Logic

If we are to speak of predicate calculus (and therefore of logic programs), we have to define the language. *The language of first-order logic* consists of three components:

1. An *alphabet*.
2. *Terms* defined over this alphabet.
3. *Well-formed formulæ* defined over this alphabet.

Let us look at the various components of this definition in the order in which they appear above.

Alphabet The alphabet is usually a set[5] of symbols. In this case, we consider the set to be partitioned into two disjoint subsets: the set of *logical symbols* (common to all first-order languages) and the set of *non-logical* (or *extra-logical*) *symbols* (which are specific to a domain of interest). For example, all first-order languages will use a (logical) symbol to denote conjunction. If we are considering orderings on a set, we will probably also have the $<$ symbol as one of the non-logical symbols.

The set of *logical symbols* contains the following elements:

- The logical connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\rightarrow$ (implication) and $\leftrightarrow$ (logical equivalence).
- The propositional constants *true* and *false*.

[5] All the sets we consider below are finite or denumerable.

- The quantifiers $\exists$ (exists) and $\forall$ (forall).
- Some punctuation symbols such as brackets "(" and ")" and comma ",".
- An (denumerably) infinite set V of *variables*, written X, Y, Z, ...

The *non-logical* (or *extra-logical*) symbols are defined by a *signature with predicates* (Σ, Π). This is a pair in which the first element, Σ is the *function signature*, that is a set of function symbols, each considered with its own arity.[6] The second element of the pair, Π, is the *predicate signature*, a set of predicate symbols together with their arities. Functions of arity 0 are said to be *constants* and are denoted by the letters $a, b, c, \ldots$. Function symbols of positive arity are, as usual, denoted by f, g, h, ..., while predicate symbols are denoted by p, q, r, ... Let us assume that the sets Σ and Π have an empty intersection and are also disjoint from the other sets of symbols listed above. The difference between function and predicate symbols is that the former must be interpreted as functions, while the latter must be interpreted as relations. This distinction will become clearer when we discuss formulæ.

Terms The concept of term, which is fundamental to mathematical logic and Computer Science, is used implicitly in many contexts. For example, an arithmetic expression is a term obtained by applying (arithmetic) operators to operands. Other types of construct, too, such as strings, binary trees, lists, and so on, can be conveniently seen as terms which are obtained using appropriate constructors.

In the simplest case, a term is obtained by applying a function symbol to as many variables and constants as required by its arity. For example, if a and b are constants, X and Y are both variables and f and g have arity 2, then $f(a, b)$ and $g(a, X)$ are terms. Nothing prevents the use of terms as the arguments to a function, provided that the arity is always respected. We can, for example, write $g(f(a, b), Y)$ or $g(f(a, f(X, Y)), X)$ and so on.

In the most general possible case, we can define terms as follows.

Definition 12.1 (Terms) The terms over a signature Σ (and over the set, V, of variables) are defined inductively[7] as follows:

- A variable (in V) is a term.
- If f (in Σ) is a function symbol of arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

As a particular case of the second point, a constant is a term. According to the letter of the definition, a term which corresponds to a constant must be written with parentheses: $a(), b(), \ldots$. Let us establish, for ease of reading, that in the case of function symbols of arity 0, parentheses are omitted. Terms without variables are said to be *ground* terms. Terms are usually denoted by the letters $s, t, u, \ldots$. Note that predicates do not appear in terms; they appear in formulæ (to express the properties of terms).

[6] Recall that, as stated above, the arity denotes the number of arguments of a function or relation.

[7] See the box on page 153 for inductive definitions.

Formulæ The *well-formed formulæ* (or *formulæ* for short) of the language allow us to express the properties of terms which, from the semantic viewpoint, are properties of some particular domain of interest. For example, if we have the predicate $>$, interpreting it in the usual fashion, writing $>(3, 2)$, we want to express the fact that the term "3" corresponds to a value with is greater than that associated with the term "2." Predicates can then be used to construct complex expressions using logical symbols. For example, the formula $>(X, Y) \wedge >(Y, Z) \rightarrow >(X, Z)$ expresses the transitivity of $>$; it asserts that if $>(X, Y)$ is true and $>(Y, Z)$ is also true then, it is the case that $>(X, Z)$.

Wanting to define formulæ precisely, we have first atomic formulæ (or atoms), constructed by the application of a predicate to the number of terms required by its arity. For example, if p has arity 2, using two the terms introduced above, we can write $p(f(a, b), f(a, X))$. Using logical connectives and quantification, we can construct complex formulæ from atomic ones. As usual, we have an inductive definition (or, equivalently, a free grammar—see Exercise 1).

Definition 12.2 (Formulæ) The (well-formed) formulæ over the signature with terms (Σ, Π) are defined as follows:

1. If $t_1, \ldots, t_n$ are terms over the signature Σ and $p \in \Pi$ is a predicate symbol of arity n, then $p(t_1, \ldots, p_n)$ is a formula.
2. *true* and *false* are formulæ.
3. If F and G are formulæ, then $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$ are formulæ.
4. If F is a formula and X is a variable, then $\forall X.F$ and $\exists X.F$ are formulæ.

12.2.2 Logic Programs

A formula of first-order logic can have a highly complex structure which often determines the effort required to find a proof. In automatic theorem proving, and also in logic programming, particular classes of formulæ, called *clauses*, have been identified which lend themselves to more efficient manipulation, in particular using a special inference rule called *resolution*. A restricted version of the concept of clause, called *definite clauses*, as well as restricted forms of resolution (*SLD resolution*—see page 392) are of interest to us. Using them, the procedure for seeking a proof is not only particularly simple but also allows the explicit calculation of the values of the variables necessary for the proof. These values can be considered as the result of the computation, giving way to an interesting model of computation based on logical deduction. We will see this model in more detail below, for now we will concentrate on syntactic aspects.

Definition 12.3 (Logic Program) Let $H, A_1, \ldots, A_n$ be atomic formulæ. A definite clause (for us simply a "clause") is a formula of the form:

$$H :- A_1, \ldots, A_n.$$

If $n = 0$, the clause is said to be a *unit*, or a fact, and the symbol $:-$ is omitted (but not the final full stop). A logic program is a set of clauses, while a pure PROLOG program is a sequence of clauses. A query (or goal) is a sequence of atoms $A_1, \ldots, A_n$.

Let us clarify some points about this definition. First, the symbol $:-$ which we did not include in our alphabet, is simply a symbol denoting (reversed) implication ($\leftarrow$) and is the same as often encountered in real logic languages.[8]

The commas in a clause or in a query, from the logical viewpoint, should be interpreted as logical conjunction. The notation "$H :- A_1, \ldots, A_n$." is therefore an abbreviation for "$H \leftarrow A_1 \wedge \cdots \wedge A_n$." Note that the full stop is part of the notation and is important because it tells a potential interpreter or compiler that the clause it is working on has terminated.

The part on the left of $:-$ is called the *head* of the clause, while that on the right is called the *body*. A fact is therefore a clause with an empty body. A program is a set of clauses in the case of the theoretical formalism. In the case of PROLOG, on the other hand, a program is considered as a sequence because, as we will see, the order of clauses is relevant. Here, we used the simplified terminology found in many recent texts (for example, [1]). For more precise terminology, see the next box. The set of clauses containing the predicate symbol p in their head is said to be the *definition* of p. Variables occurring in the body of a clause and not in the head are said to be *local* variables.

12.3 Theory of Unification

The fundamental computational mechanism in logic programming is the solution of equations between terms using the unification procedure. In this procedure, substitutions are computed so that variables and terms can be bound (or instantiated). The composition of the different substitutions obtained in the course of a computation provides the result of the calculation. Before seeing the computational model for logic programming in detail, we must analyse unification in a little detail, a task we undertake in this section.

12.3.1 The Logic Variable

Before going into detail on the process of unification, it is important to clarify that the concept of variable we are considering is different from that seen in Sect. 6.2.1. Here, we consider the so-called *logic variable* which, analogously to the variables

[8] $:-$ is used for pragmatic reasons. When logic languages were being introduced, it was much easier to type $:-$ rather than a left arrow.

Clauses

The reader familiar with first-order logic will have recognised the notion of clause given in Definition 12.3 as being a particular case of the one used in logic. A clause, in the general sense, is indeed a formula of the form:

$$\forall X_1, \dots, X_m (L_1 \vee L_2 \vee \cdots \vee L_n)$$

where $L_1 \dots L_n$ are *literals* (atoms or negated atoms) and $X_1, \dots, X_n$ are all the variables that occur in $L_1 \dots L_n$. For greater clarity, we separate the negated atoms from the others and therefore we will see a clause as a formula of the form:

$$\forall X_1, \dots, X_m (A_1 \vee A_2 \vee \cdots \vee A_m, \neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_k).$$

Using a known logical equivalence which allows implications to be written as disjunction, we can express this formula in the following special form (which is equivalent to the previous one);

$$A_1, \dots A_m \leftarrow B_1, \dots, B_k. \tag{12.1}$$

A *program clause*, also called a *definite clauses*, is a clause that has only one unnegated atom. In the form shown in (12.3), a definite clause always has $m = 1$, which is exactly the notion introduced in Definition 12.3. A fact is therefore a definite clause containing no negated atoms. Finally, a *negative clause*, also called a *query* or *goal*, is a clause of the form (12.3) in which $m = 0$.

which we have already spoken of, is an unknown which can assume values from a predetermined set. In our case, this set is that of definite terms over the given alphabet. This fact, together with the use to which logic variables are put in logic programming, gives rise to three important differences between this concept and the modifiable variables in imperative languages:

1. The logic variable can be bound only once, in the sense that if a variable is bound to a term, this binding cannot be destroyed (but the term might be modified, as will be explained below). For example, if we bind the variable X to the constant a in a logic program, the binding cannot later be replaced by another which binds X to the constant b. Clearly, this is possible in imperative languages using assignment. The fact that the binding of a variable cannot be eliminated does not mean, however, that it is impossible to modify the value of the variable. This apparent contradiction merits more consideration; this is done in the next point.
2. The value of a logic variable can be partially defined (or can be undefined), to be specified later. This is because a term that is bound to a variable can contain, in its turn, other logic variables. For example, if the variable X is bound to the term $f(Y, Z)$, successive bindings of the variables Y and Z will also modify the value of the variable X: if Y is bound to a and Z is bound to $g(W)$, the value of X

will become the term $f(a, g(W))$. The process could continue by modifying the value of W. This mechanism for specifying the value of a variable by successive approximations, so to speak, is typical of logic languages and is somewhat different from the corresponding one encountered in imperative languages, where a value assigned to a value cannot be partially defined.

3. A third important difference concerns the bidirectional nature of bindings for logic variables. If X is bound to the term $f(Y)$ and later we are able to bind X to the term $f(a)$, the effect so produced is that of binding the variable Y to the constant a. This does not contradict the first point, given that the binding of X to the term $f(Y)$ is not destroyed, but the value of $f(Y)$ is specified through the binding of Y. Therefore, we can not only modify the value of a variable by modifying the term to which it is bound, but we can also modify this term by providing another binding for that variable. Clearly, this second binding must be consistent with the first, that is if X is bound to the term $f(Y)$, we cannot try to bind X to a term of the form $g(Z)$.

The last point is fundamental. This is the point that allows us to use a single logic program in quite different ways, as will be seen below. Essentially, we are talking about a property that derives from the unification mechanism, which we will discuss shortly. First, however, we must introduce the concept of substitution.

12.3.2 Substitution

The connection between variables and terms is made in terms of the concept of substitution, which, as its name tells us, allows the "substitution" of a variable by a term. A substitution, usually denoted by the greek letters $\vartheta, \sigma, \rho, \ldots$, can be defined as follows.

Definition 12.4 (Substitution) A substitution is a function from variables to terms such that the number of variables which are not mapped to themselves is finite. We denote a substitution ϑ by the notation:

$$\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}$$

where $X_1, \ldots, X_n$ are different variables, $t_1, \ldots, t_n$ are terms and where we assume that t_i is different from X_i, for $i = 1, \ldots, n$.

In the preceding definition, a pair X_i/t_i is said to be a *binding*. In the case in which all the $t_1, \ldots, t_n$ are ground terms, then ϑ is said to be a *ground substitution*. We write ϵ for the empty substitution. For ϑ represented as in Definition 12.4, we define the domain, codomain and variables of a substitution as follows:

$$\mathit{Domain}(\vartheta) = \{X_1, \ldots, X_n\},$$

$$\mathit{Codomain}(\vartheta) = \{Y \mid Y \text{ a variable in } t_i, \text{ for some } t_i, 1 \leq i \leq n\}.$$

A substitution can be applied to a term, or, more generally, to any syntactic expression, to modify the value of the variables present in the domain of the substitution. More precisely, if we consider an expression, E (which could be a term, a literal, a conjunction of atoms, etc), the result of the application of $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ to E, denoted by $E\vartheta$,[9] is obtained by simultaneously replacing every occurrence of X_i in E by the corresponding t_i, for all $1 \le i \le n$. Therefore, for example, if we apply the substitution $\vartheta = \{X/a, Y/f(W)\}$ to the term $g(X, W, Y)$, we obtain the term $g(X, W, Y)\vartheta$, that is $g(a, W, f(W))$. Note that the application is simultaneous. For example, if we apply the substitution $\sigma = \{Y/f(X), X/a\}$ to the term $g(X, Y)$, we obtain $g(a, f(X))$ (and not $g(a, f(a))$).

The *composition*, $\vartheta\sigma$, of two substitutions $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ and $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$ is defined as the substitution obtained by removing from the set

$$\{X_1/t_1\sigma, \dots, X_n/t_n\sigma, Y_1/s_1, \dots, Y_m/s_m\}$$

the pairs $X_i/t_i\sigma$ such that X_i is equal to $t_i\sigma$ and the pairs Y_i/s_i such that $Y_i \in \{X_1, \dots, X_n\}$. Composition is associative and it is not difficult to see that, for any expression, E, it is the case that $E(\vartheta\sigma) = (E\vartheta)\sigma$. The effect of the application of a composition is the same as it is obtained by successively applying the two substitutions that we want to compose.

For example, composing

$$\vartheta_1 = \{X/f(Y), W/a, Z/X\} \quad \text{and} \quad \vartheta_2 = \{Y/b, W/b, X/Z\}$$

we obtain the substitution

$$\vartheta = \vartheta_1\vartheta_2 = \{X/f(b), W/a, Y/b\}.$$

If we apply the latter to the term $g(X, Y, W)$, we obtain the term $g(f(b), b, a)$. The same term is obtained first by applying ϑ_1 to $g(X, Y, W)$, then applying ϑ_2 to the result. Note that, in the result ϑ of the composition, the Y in ϑ_1 is instantiated to b because of the binding occurring in ϑ_2. The X occurring in Z/X is instantiated to Z using the binding X/Z in ϑ_2, after which the binding Z/Z is eliminated from the resulting substitution (because it is the identity). The bindings W/b and X/Z present in ϑ_2 finally disappear from ϑ because both W and Z appear in the domain (or, on the left of a binding) in ϑ_1.

A particular type of substitution is formed from those which simply rename their variables. For example, the substitution $\{X/W, W/X\}$ does nothing more than change the names of the variables X and W. Substitutions like this are called renamings and can be defined as follows.

[9] Note that there exist two opposing versions, both for the notation used for denoting substitutions and for that one used for application of a substitution. The notation for substituting of a term N in place of a variable X used on page 359 is N/X using the conventions adopted in functional programming. Here, on the other hand, such a binding is written X/N (i.e., backwards). In general, we will use the notation most commonly used in logic programming.

Definition 12.5 (Renaming) A substitution ρ is a renaming if its inverse substitution ρ^{-1} exists and is such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$.

Note that the substitution $\{X/Y, W/Y\}$ is not a renaming. Indeed, it not only changes the names of the two variables, but also makes the two variables equal that were previously distinct.

Finally, it will be useful to define a preorder, $\leq$, over substitutions, where $\vartheta \leq \sigma$ is read as: ϑ is more general than σ. Let us therefore define $\vartheta \leq \sigma$ if (and only if) there exists a substitution γ such that $\vartheta\gamma = \sigma$. Analogously, given two expressions, t and t', we define $t \leq t'$ (t is more general t') if and only if there exists a ϑ such that $t\vartheta = t'$. The relation $\leq$ is a preorder and the equivalence induced by it[10] is called the *variance*; t and t' are therefore variants if t is an instance of t' and, conversely, t' is an instance of t. It is not difficult to see that this definition is equivalent to saying that t and t' are variants if there exists a renaming ρ such that t is syntactically identical to $t'\rho$. These definitions can be extended to any expression in an obvious fashion. Finally, if ϑ is a substitution that has as domain the set of variables V, and W is a subset of V, the *restriction* of ϑ to the variables in W is the substitution obtained by considering only the bindings for variables in W, that is the substitution defined as follows:

$$\{Y/t \mid Y \in W \text{ and } Y/t \in \vartheta\}.$$

A comparison with the imperative paradigm can help in better understanding the concepts under consideration. As we saw in Sect. 2.5, in the imperative paradigm, the semantics can be expressed by referring to a concept of state that associates every variable with a value.[11] An expression containing variables is evaluated with respect to a state to obtain a value that is completely defined.

In the logic paradigm, the association of values with variables is implemented through substitutions. The application of a substitution to a term (or to a more complex expression) can be seen as the evaluation of the terms, an evaluation that returns another term, and therefore, in general, a partially defined value.

12.3.3 Most General Unifier

The basic computation mechanism for the logic paradigm is the evaluation of equations of the form $s = t$,[12] where s and t are terms and "=" is a predicate symbol

[10]Given a preorder, $\leq$, the equivalence relation *induced* by $\leq$ is defined as $t = t'$ if and only if $t \leq t'$ and $t' \leq t$.

[11]Wishing to be precise, as we have seen in the box on page 135, in real languages, this association is implemented using two functions, environment and memory. This however does not alter the import of what we are saying.

[12]We draw attention to the notation that, as usual, overloads the "=" symbol. By writing $\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}$, we mean that ϑ is the substitution $\{X_1/t_1, \ldots, X_n/t_n\}$, while writing $s = t$, we mean an equation.

interpreted as syntactic equality over the set of all ground terms; this set is called the *Herbrand Universe*.[13] We will attempt better to clarify this equality.

If we write $X = a$ in a logic program, we mean that the variable X must be bound to the constant a. The substitution $\{X/a\}$ therefore constitutes a solution to this equation since, by applying the substitution to the equation, we obtain $a = a$ which is an equation that is clearly satisfied. The syntactic analogy with assignment in imperative languages should not be allowed to deceive, for it deals with a completely different concept. Indeed, unlike in an imperative language, here we can also write $a = X$ instead of $X = a$ and the meaning does not change (the equality that we are considering is symmetric, as are all equality relations). Also the analogy with the equality of arithmetic expressions can be, in some ways, misleading, as illustrated by the following example.

Let us assume that we have a binary function symbol $+$ which, intuitively, expresses the sum of two natural numbers, and consider the equation $3 = 2 + 1$, where, for simplicity, we use infix notation for $+$ and we represent in the usual fashion the natural numbers. Given that the equation $3 = 2 + 1$ does not contain variables, it can be either true (or, solved) or false (that is, insoluble). Contrary to what arithmetic intuition would suggest to us, in a (pure) logic program this equation cannot be solved. This is because, as we have said, the symbol $=$ is interpreted as syntactic equality over the set of ground terms (the Herbrand universe). It is clear that, from the syntactic viewpoint, the constant 3 is different from the term $2 + 1$ and since they are treated as ground terms (that is completely instantiated terms) there is no way that they can be made syntactically equal. Analogously, the equation $f(X) = g(Y)$ has no solutions (it is not solvable) because however the variables X and Y are instantiated, we cannot make the two different function symbols, f and g, equal. Note that the equation, $f(X) = f(g(X))$, also has no solutions, because the variable X in the left-hand term must be instantiated with $g(X)$ and therefore the possible solution must contain the substitution $\{X/g(X)\}$. The application of this substitution to the term on the right would instantiate X, producing the term $f(g(g(X)))$, so the X in the right-hand term would have to be instantiated to $g(g(X))$ rather than to $g(X)$ and so on, without ever reaching a solution.[14] In general, therefore, the equation $X = t$ cannot be solved if t contains the variable X (and is different from X).

If, on the other hand, we consider the equation $f(X) = f(g(Y))$, it is solvable; one solution is the substitution $\vartheta = \{X/g(Y)\}$ because if this is applied to the two terms in the equation, it makes them syntactically equal. Indeed, $f(X)\vartheta$ is identical to $f(g(Y))$ which is identical to $f(g(Y))\vartheta$. In more formal terms, we can say that the substitution ϑ *unifies* the two terms of the equation and it is therefore called a *unifier*. Note that we have said "a solution" because there are many (an infinite number) of substitutions that are unifiers of X and $g(Y)$. It is sufficient to instantiate Y in the definition of ϑ. So, for example, the substitution $\{X/g(a), Y/a\}$ is also an

[13]The symbol "=" is usually written in infix notation for increased readability. Different logic languages can have different syntactic readings for it and use different equality symbols, each with a different meaning. Here, we refer to "pure" logic programming.

[14]Note that by admitting infinite terms, we can find a solution, however.

unifier, as are $\{X/g(f(Z)), Y/f(Z)\}$, $\{X/g(f(a)), Y/f(a)\}$, and so on. All these substitutions are, however, *less general* than ϑ according to the preorder that we defined above. Each of them, that is, can be obtained by composing ϑ with some other, appropriate, substitution. For example, $\{X/g(a), Y/a\}$ is equal to $\vartheta\{Y/a\}$ (we denote the composition of substitutions with juxtaposition as already seen). In this sense, we say that ϑ is the most general unifier (or m. g. u.) of X and $g(Y)$.

Before moving to general definitions, note one last important detail: the process of solving an equation, and thus a unification, can create bidirectional bindings, that is the direction in which the associations must be realised is not specified. For example, a solution of the equation $f(X, a) = f(b, Y)$ is given by the substitution $\{X/b, Y/a\}$, where a variable on the left and one on the right of the $=$ symbol are bound. Instead, in the equation, $f(X, a) = f(Y, a)$, to find a solution, we can choose whether to bind the variable on the left (using $\{X/Y\}$) or the one on the right (using $\{Y/X\}$) or even both (using $\{X/Z, Y/Z\}$).

This aspect, to which we will return, is important because it allows the implementation of bidirectional parameter-passing mechanisms and, a unique characteristic of the logic paradigm, to use the same program in different ways, turning input arguments into outputs and vice versa, without modifying the program at all.

We now have the formal definition.

Definition 12.6 (M. g. u.) Given a set of equations $E = \{s_1 = t_1, \ldots, s_n = t_n\}$, where $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ are terms, the substitution, ϑ, is a unifier for E if the sequence $(s_1, \ldots, s_n)\vartheta$ and $(t_1, \ldots, t_n)\vartheta$ are syntactically identical. A unifier of E is said to be the *most general unifier* (m. g. u.) if it is more general than any other unifier of E; that is, for every other unifier, σ, of E, σ is equal to $\vartheta\tau$ for some substitution, τ.

The preceding concept of unifier can be extended to other syntactic objects in an obvious fashion. In particular, we say that ϑ is a unifier of two atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ if ϑ is a unifier of $\{s_1 = t_1, \ldots, s_n = t_n\}$.

12.3.4 A Unification Algorithm

An important result, due to Robinson in 1965, shows that the problem of determining whether a set of equations of terms can be unified is decidable. The proof is constructive, in the sense that it provides a *unification algorithm* which, for every set of equations, produces their m. g. u. if the set is unifiable and returns a failure in the opposite case.[15]

[15]Robinson's original algorithm considers the unification of just two terms but this, obviously, is not reductive given that the unification of $\{s_1 = t_1, \ldots, s_n = t_n\}$ can be seen as the unification of $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$.

It can also be proved that a m. g. u. is unique up to renaming. The unification algorithm which we will now see is not Robinson's original but is Martelli and Montanari's from 1982 and it makes use of ideas present in Herbrand's thesis of 1930.

Martelli and Montanari's unification algorithm Given a set of equations

$$E = \{s_1 = t_1, \ldots, s_n = t_n\},$$

the algorithm produces either failure or a set of equations of the following, so called, solved form:

$$\{X_1 = r_1, \ldots, X_m = r_m\},$$

where $X_1, \ldots, X_m$ are distinct variables which not appearing in the terms $r_1, \ldots, r_m$. The set of equations is equivalent to the starting set, E, and from it we can obtain a m. g. u. for E simply interpreting every equality as a binding. Therefore, the m. g. u. that we seek is the substitution:

$$\{X_1/r_1, \ldots, X_m/r_m\}.$$

The algorithm is non-deterministic in the sense that when there are more possible actions, one is chosen in an arbitrary fashion, with no priority between the various actions.[16] The algorithm is given by the following steps.

1. Nondeterministically select one equation from the set E.
2. According to the type of equation chosen, execute, if possible, one of the specific operations as follows (where on the left of the ":" we indicate the type of equation and, on the right, we have the associated action):
   a. $f(l_1, \ldots, l_k) = f(m_1, \ldots, m_k)$: eliminate this equation from the set E and add to E the equations $l_1 = m_1, \ldots, l_k = m_k$.
   b. $f(l_1, \ldots, l_k) = g(m_1, \ldots, m_k)$: it f is different from g, terminate with failure.
   c. $X = X$: eliminate this equation from the set E.
   d. $X = t$: if t does not contain the variable X and this variable appears in another equation in the set E, apply the substitution $\{X/t\}$ to all the other equations in the set E.
   e. $X = t$: if t contains the variable X, terminate with failure.
   f. $t = X$: if t is not a variable, eliminate this equation from the set E and add to E the equation $X = t$.
3. If none of the preceding operations is possible, terminate with success (E contains the solved form). If, on the other hand, an operation different from termination with failure has been executed, go to (1).

This, as can be seen, is a very simple algorithm. We will discuss it in detail, nonetheless.

[16]We will return to the non-determinism briefly when we discuss the operational semantics of logic programs.

The first case is one in which the two terms agree on the function symbol. In this case, in order to unify the two terms, it is necessary to unify the arguments, so we replace the original equation with the equations obtained by equating the arguments in each position. Note that this case also includes equivalence between constants of the form $a = a$ (where a is a function symbol of arity 0) which are eliminated without adding anything.

The second case produces a failure given that, when f and g are different, the two terms cannot be unified (as we have already seen).

The equation $X = X$ is eliminated using the identity substitution which therefore produces no change in the other equations.

The fourth case is the most interesting. An equation $X = t$ is already in solved form (because, by assumption, t does not contain X). In other words, the substitution $\{X/t\}$ is the m. g. u. of this equation. For the effect of such an m. g. u. to be combined with those produced by other equations, we must apply the substitution $\{X/t\}$ to all the other equations in the set E.

On the other hand, in the case in which t contains the variable X, as we have already seen, the equation has no solution and therefore the algorithm terminates with failure. It is important to note that this check, called the *occurs check*, is removed from many implementations of PROLOG for reasons of efficiency. Therefore, many PROLOG implementations use an incorrect unification algorithm!

The last case, finally, serves only to obtain a result form in which the variables appear on the left and the terms on the right of the $=$ symbol.

It is easy to convince oneself that the algorithm terminates, given that the depth of the input terms is finite. It is, moreover, possible to prove that the algorithm produces an m. g. u. which is obtained by the interpretation as substitutions of the final result form of the equations.

By a careful consideration of the algorithm, it can be seen that the computation of the most general unifier occurs in an incremental fashion by solving ever simpler equations until a result form is encountered. It is also possible to express this process in terms of substitution compositions as happens in the operational model of logic languages. We will exemplify this point with an example which also constitutes an example of the application of the unification algorithm that we have just seen. To simplify, we will always choose the leftmost equation (the final result, anyway, does not depend upon such an assumption; any other selection rule would lead to the same result up to renaming).

Let us consider the set of equations:

$$E = \{f(X, b) = f(g(Y), W), h(X, Y) = h(Z, W)\}.$$

Choosing the first equation on the left, using the operation described in (a), the set E is transformed into:

$$E_1 = \{X = g(Y), b = W, h(X, Y) = h(Z, W)\}.$$

Using operation (d), we therefore obtain:

$$E_2 = \{X = g(Y),\ b = W,\ h(g(Y), Y) = h(Z, W)\}.$$

Using (f) and then (d) again on the second equation, we finally obtain:

$$E_3 = \{X = g(Y),\ W = b,\ h(g(Y),Y) = h(Z,b)\}.$$

This already contains the result of the first equation in the set E. In fact, the substitution:

$$\vartheta_1 = \{X/g(Y), W/b\}$$

is an m. g. u. for $f(X,b) = f(g(Y),W)$.

Continuing with the second equation from the original set, suitably instantiated by the substitutions already computed, using operation (a), we obtain:

$$E_4 = \{X = g(Y),\ W = b,\ g(Y) = Z,\ Y = b\};$$

then, by (f), we obtain:

$$E_5 = \{X = g(Y),\ W = b,\ Z = g(Y),\ Y = b\}.$$

Finally, using (d) applied to the last equation, we have:

$$E_4 = \{X = g(b),\ W = b,\ Z = g(b),\ Y = b\},$$

which constitutes the result form for the set E and therefore also provides the m. g. u. of the initial set in the form of the substitution

$$\vartheta = \{X/g(b), W/b, Z/g(b), Y/b\}.$$

There are two important observations to be made. First, note how the value of some variables can be partially specified first, and then later refined. For example, ϑ_1 (m. g. u. of the first equation in E) tells us that X has $g(Y)$ as its value and only by solving the second equation do we see that Y has b as its value. Therefore it can be seen that $g(b)$ is the value of X (as, indeed, it results in the final m. g. u.).

Moreover, we can see that if we consider $\{h(g(X), Y) = h(Z, W)\}\vartheta_1$ (the second equation in E instantiated using the m. g. u. of the first), we obtain the equation

$$h(g(Y),Y) = h(Z,b)$$

for which the substitution

$$\vartheta_2 = \{Z = g(b), Y = b\}$$

is a m. g. u. Using the definition of composition of substitutions, it is easy to check that $\vartheta = \vartheta_1\vartheta_2$. The m. g. u. of the set E can therefore be obtained by composing the first equation's m. g. u. with that of the second (to which the first m. g. u. has already been applied). This, as we have already said, is what, indeed, normally happens in implementations of logic languages, where, instead of accumulating all the equations and then solve them, an m. g. u. is computed on each step of the computation and is composed with the ones that were previously obtained.

12.4 The Computational Model

The logic paradigm, implementing the idea of "computation as deduction", uses a computational model that is substantially different from all the others that we have seen so far. Wishing to synthesise, we can identify the following main differences from the other paradigms:

1. The only possible values, at least in the pure model, are terms over a given signature.
2. Programs can have a declarative reading which is entirely logical, or a procedural reading of an operational kind.
3. Computation works by instantiating the variables appearing in terms (and therefore in goals) to other terms using the unification mechanism.
4. Control, which is entirely handled by the abstract machine (except for some possible annotations in PROLOG) is based on the process of automatic backtracking.

Below, we will try therefore to illustrate the computational model for the logic paradigm by analysing these four points. We will explicitly discuss the differences between logic programming and PROLOG.

12.4.1 The Herbrand Universe

In logic programming, terms are a fundamental element. The set of all possible terms over a given signature is called the *Herbrand Universe* and is the domain over which computation in logic programs is performed. It has some characteristics that must be understood.

- The alphabet over which programs are defined is not fixed but can vary as far as non-logical symbols are concerned.
- As a (partial) consequence of the previous point, unlike what happens in imperative languages, no predefined meaning is assigned to the (non-logical) symbols of the alphabet. For example, a program can use the $+$ symbol to denote addition, while another program can use the same symbol to denote string concatenation. The exceptions are the (binary) equality predicate and some other predefined ("built-in") symbols in PROLOG.[17]
- As a final consequence, no type system is present in logic languages (at least in the classic formalism). The only type that is present is that of terms with which we can represent arithmetic, list, expressions, etc.

From the theoretical stance, the fact that there are no types and that computation occurs in the Herbrand Universe is not limiting, rather it permits the highly elegant and, all considered, simple expression of the formal semantics of logic programs. For example, with only two function symbols, 0 (constant zero) and s (successor,

[17]As well as predefined predicates in constraint languages, a topic we will not consider.

of arity 1), we can express the natural numbers using the terms 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, etc. With a little effort, we can express the normal arithmetic operations in terms of this two-symbol representation.

From the practical view point, on the other hand, the lack of types is a serious problem. In fact, in PROLOG, as in other, more recent, logic languages, some primitive types have been introduced (for example, integers and associated arithmetic operations). The languages of this paradigm are therefore always somewhat lacking as far as types are concerned.

12.4.2 Declarative and Procedural Interpretation

As we have just hinted, a clause, and therefore a logic program, can have two different interpretations: one *declarative* and one *procedural*.

From the *declarative* viewpoint, a clause $H : -A_1, \dots, A_n$ is a formula which, basically, expresses the fact that if A_1 and A_2 and ... and A_n are true, then H is also true. A query (or goal) is also a formula for which we want to prove, provided that it is appropriately instantiated, that it is a logical consequence of the program, and is, therefore, true in all interpretations in which the program is true.[18] This interpretation can be developed using the methods of logic (in particular, some elementary concepts of model theory) in such a way as to give a meaning to a program in purely declarative terms without referring at all to a computational process. For this interpretation, while interesting, we refer the reader to the specialist literature cited at the end of the chapter.

The *procedural* interpretation, on the other hand, allows us to read a clause such as:

$$H : -A_1, \dots, A_n$$

as follows. To prove H, it is necessary first to prove $A_1, \dots, A_n$, or rather to compute H, it is necessary first to compute $A_1, \dots, A_n$. From this, we can view a predicate as the name of a procedure, whose defining clauses constitute its body. In this interpretation, we can read an atom in the body of a clause, or in a goal, as a procedure call. A logic program is therefore a set of declarations and a goal is no more than the equivalent of "main" in an imperative program, given that it contains all the calls to the procedures that we want to evaluate. The comma in the body of clauses and in goals, in PROLOG (but not in other pure logic programming languages), can be read as the analogue of ";" in imperative languages.

Precise correspondence theorems allow us to reconcile the declarative and procedural views, proving that the two approaches are equivalent.

From a formal viewpoint, the procedural interpretation is supported by so-called SLD resolution, a logical inference rule which we will discuss in the box on

[18]Here we will content ourselves with an intuitive idea of this concept. The interested reader can consult any text on logic for more information.

page 392. It is also possible to describe the procedural interpretation in a more informal manner, using only the analogy with procedure calls and parameter passing which we have just outlined. This is the approach we will use below.

12.4.3 Procedure Calls

Let us consider for now a simplified definition of clause in which we assume that, in the head, all the arguments of the predicate are distinct variables. An arbitrary clause of this type therefore has the form:

$$p(X_1, \dots, X_n) :- A_1, \dots, A_m$$

and, as we have anticipated, it can be seen as the declaration of the procedure p with n formal parameters, $X_1, \dots, X_n$. An atom $q(t_1, \dots, t_n)$ can be seen as a call to the procedure q with n actual parameters, $t_1, \dots, t_n$. In the definition of p, therefore, the body is formed from the calls to m procedures which constitute the atoms $A_1, \dots, A_m$.

In accordance with this view, and analogously to what happens in imperative languages, the evaluation of the call $p(t_1, \dots, t_n)$ causes the evaluation of the body of the procedure after parameter passing has been performed. Parameter passing uses a technique similar to call by name, replacing the formal parameter X_i with the corresponding actual parameter t_i. Moreover, given that the variables appearing in the body of the procedure are to be considered as logical variables, they can be considered to be distinct from all other variables. In block-structured languages, this happens implicitly given that the body of the procedure is considered as a block with its own local environment. Here, on the other hand, the concept of block is absent, so, in order to avoid conflicts between variable names, we assume that, before using a clause, all the variables appearing in it are systematically renamed (so that they do not conflict with any others).

Using more precise terms, we can say that the evaluation of the call $p(t_1, \dots, t_n)$, with the definition of p seen above, causes the evaluation of the m calls:

$$(A_1, \dots, A_m)\vartheta$$

present in the body of p, appropriately instantiated by the substitution

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

which performs parameter passing. In the case in which the body of the clause is empty (or that $m = 0$), the procedure call terminates immediately. Otherwise, the computation proceeds with the evaluation of the new calls. Using logic programming terminology, this can be expressed by saying that the evaluation of the goal $p(t_1, \dots, t_n)$ produces the new goal:

$$(A_1, \dots, A_m)\{X_1/t_1, \dots, X_n/t_n\}$$

which, in its turn, will have to be evaluated. When all the calls generated by this process have been evaluated (provided there has been no failure), the computation terminates with success and the final result is composed of the substitution that associates the values computed in the course of the computation with the variables present in the initial call ($X_1, \dots, X_n$, in our case). This substitution is said to be the *answer computed* by the initial goal in the given program. We will see a more precise definition of this concept below. For now, we will see a simple example. Let us consider the procedure `list_of_2` defined below and identical to the procedure `list_of_27` of Sect. 12.1.1, except for the lesser number of anonymous variables:

```
list_of_2(Ls):- Ls = [_,_].                                    1
```

The evaluation of the call `list_of_2(LXs)` causes the evaluation of the body of the clause, but instantiated by the substitution $\{Ls/LXs\}$, that is:

```
LXs = [_,_]
```

This is a particular call because $=$ is a predefined call which, as we saw in the previous section, is interpreted as syntactic equality over the Herbrand Universe and, operationally, corresponds to the unification operation. The previous call therefore reduces to the attempt (performed by the language interpreter) to solve the equation using unification. In our case, clearly, this attempt succeeds and produces the m. g. u. $\{LXs/[\_,\_]\}$ which is the result of the computation. The previous substitution is therefore the answer computed by the goal `list_of_2` in the logic program defined by the single line (1) above.

Evaluation of a non-atomic goal The view presented in the previous subsection must be generalised and made more precise to clarify the logic-programming computational model. Below, in order to conform with current terminology, we will talk of atomic goals and goals rather than procedure calls and sequences of calls. The analogy with the conventional paradigm remains valid, though.

In the case in which a non-atomic goal must be evaluated, the computational mechanism is analogous to the one seen above, except that now we must choose one of the possible calls using some *selection rule*. While in the case of pure logic programming no such rule is specified, PROLOG adopts the rule that the leftmost atom is always chosen. It is however possible to prove that whatever rule is adopted, the results that are computed are always the same (see also Exercises 13 and 14 at the end of the chapter).

Assuming, for simplicity, that we adopt the PROLOG rule, we can describe the progress of the evaluation process. Let

$$B_1, \dots, B_k$$

with $k \geq 1$, be the goal to be evaluated. We distinguish the following cases according to the form of the selected atom, B_1:

1. If B_1 is an equation of the form $s = t$, try to compute a m. g. u. (using the unification algorithm). There are two possibilities:
   a. If the m. g. u. exists and is the substitution σ, then the result of the evaluation is the goal:

   $$(B_2, \ldots, B_k)\sigma$$

   obtained from the previous one by eliminating the chosen atom and applying the m. g. u. thus computed. If $k = 1$ (and therefore $(B_2, \ldots, B_k)\sigma$ is empty), the computation terminates in success.
   b. If the m. g. u. does not exist (or the equation has no solutions), then we have failure.
2. If, on the other hand, B_1 has the form $p(t_1, \ldots, t_n)$, we have the following two cases:
   a. If, in the program, there exists a clause of the form:

   $$p(X_1, , \ldots, X_n) : -A_1, \ldots, A_m$$

   (which we consider renamed to avoid variable capture), then the result of evaluation is a new goal:

   $$(A_1, \ldots, A_m)\vartheta, B_2, \ldots, B_k$$

   where $\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}$. If $k = 1$ (then we have an atomic goal) and $m = 0$ (the body of the clause is empty), then the computation terminates with success.
   b. If, in the program, there exists no clause defining the predicate p, we have failure.

To be able exactly to define the results of the computation (the answers computed), we need to clarify some aspects of control, which we will do in Sect. 12.4.4.

Heads with arbitrary terms So far, we have assumed that the heads of clauses contain just distinct variables. We have made this choice to preserve the similarity with procedures in traditional languages. However, real logic programs also use arbitrary terms as arguments to predicates in heads, as we have seen in the example in Sect. 12.1.1. The box on page 392 provides the evaluation rule for such a general case. Note, however, that our assumption is in no way limiting (apart, perhaps, from textual convenience). Indeed, as seems clear from what was said in the box and from the preceding treatment, a clause of the form:

$$p(t_1, \ldots, t_n) : -A_1, \ldots, A_m$$

can be seen as an abbreviation of the clause:

$$p(X_1, \ldots, X_n) : -X_1 = t_1, \ldots, X_n = t_n, A_1, \ldots, A_m.$$

In the following examples, we will often use the notation with arbitrary terms in heads.

SLD Resolution

Definite clauses allow a natural procedural reading based on *resolution*, an inference rule which is complete for sets of clauses and which was introduced by Robinson and used in automated deduction. In logic programming, we have *SLD resolution*, that is linear resolution guided by a clause selection rule (SLD is an acronym for Selection rule-driven Linear resolution for Definite clauses).

This rule can be described as follows. Let G be the goal $B_1, \ldots, B_k$ and let C be the (definite) clause $H : -A_1, \ldots, A_n$. We say that G' is *derived* from G and C using ϑ or, equivalently, G' is a *resolvent* of G and C if (and only if) the following conditions are met:

1. B_m, with $1 \leq m \leq k$, is a *selected* atom from those in G.
2. ϑ is the m. g. u. of B_m and H.
3. G' is the goal $(B_1, \ldots, B_{m-1}, A_1, \ldots, A_n, B_{m+1}, \ldots B_k)\vartheta$.

Note that, unlike Sect. 12.4.3, here it is necessary also to apply ϑ to the other atoms occurring in the goal G because the heads of the clauses contain arbitrary terms and therefore ϑ could instantiate variables that are also in the goal.

Given a goal, G, and a logic program, P, an *SLD derivation* of $P \cup G$ consists of a (possibly infinite) sequence of goals $G_0, G_1, G_2, \ldots$, of a sequence $C_1, C_2, \ldots$ of clauses in P which have been renamed in such a way as to avoid variable capture and a sequence $\vartheta_1, \vartheta_2, \ldots$, of m. g. u. s such that G_0 is G and, for $i \geq 1$, every G_i, is derived from G_{i-1} and C_i using ϑ_i. An *SLD refutation* of $P \cup G$ is a finite SLD derivation of $P \cup G$ which has the empty clause as the last resolvent of the derivation. If $\vartheta_1, \vartheta_2, \ldots$ are the m. g. u. s used in the refutation of $P \cup G$, we say that the substitution $\vartheta_1\vartheta_2 \cdots \vartheta_n$ restricted to the variables occurring in G is the *answer substitution computed* by $P \cup G$ (or for the goal G in the program P).

Classic results, due to K. L. Clark, show that this rule is sound and complete with respect to the traditional first-order logical interpretation.

Indeed, it can be proved that if ϑ is the *answer substitution computed* by the goal G in program P, then $G\vartheta$ is a logical consequence of P (soundness). Moreover, if $G\vartheta$ is a logical consequence of P, then no matter which selection rule is used, there exists a SLD refutation of $P \cup G$ with computed answer σ such that $G\sigma$ is more general than $G\vartheta$ (strong completeness).

12.4.4 Control: Non-determinism

In the evaluation of a goal, we have two degrees of freedom: the selection of the atom to evaluate and the choice of the clause to apply.

For the first, we have said that we can fix a selection rule, without influencing the final results of the computation that terminate in success.

For the selection of clauses, on the other hand, the matter is more delicate. Given that a predicate can be defined by more than one clauses, and we have to use only one of them at a time, we could think about fixing some rule for choosing clauses, analogous to the one used to choose atoms. The following reveals however a problem. Let us consider the following program, which we call `Pa`:

```
p(X):- p(X).                                                    1
p(X):- X=a.                                                     2
```

and let us assume that we choose clauses from top to bottom, following the textual order of the program. It is easy to see that by adopting this rule, the evaluation of `p(Y)` never terminates and therefore we do not have an answer computed by the program. Indeed, using clause (1) and the substitution $\{X/Y\}$, the initial goal, after one computation step, becomes the goal `p(Y)`, which is again the starting goal. According to the clause-selection rule, we must choose clause (1) and apply it to this (new) goal, and so on. It is however clear that using clause (2) we would immediately obtain a computation which terminates, producing the substitution $\{Y/a\}$ as result. Note that to fix another order, for example, from bottom to top, would not in general solve this problem.

In the light of this example, let us carefully reconsider the evaluation rule for a goal seen in the previous subsection and, in particular, let us fix on point 2(a), where we wrote "if there exists a clause" without specifying how this can be chosen. By doing this, we have therefore introduced into the computation model a form of non-determinism: in the case in which there is more than one clause for the same predicate, we can choose one in a non-deterministic fashion, without adopting any particular rule. This form of non-determinism is called "*don't know*" non-determinism because we do not know which the "right" clause will be that allows the termination of the computation with success. The theoretical model of logic programming keeps this non-determinism when it considers all possible choices of clause and therefore all possible results of the various computations that are produced as a consequence of this choice. The result of the evaluation of a goal G in the program P is therefore a set of computed answers, where each of these answers is the substitution obtained by the composition of all the m. g. u. s which are encountered in a specific computation (with specific choices of clauses), restricted to the variables present in G. For a more precise definition of this idea, as well as of the whole process of evaluating a goal, see the box on page 392.

Turning to our previous example, the only answer computed for the goal `p(Y)` in program `Pa` is the substitution $\{Y/a\}$ while, for the same program, the goal `p(b)` has no computed answer given that all its computations either terminate with failure (when the second clause is used) or do not terminate (when the first clause is used).

Backtracking in PROLOG When moving from the theoretical model to an implemented language, such as PROLOG, non-determinism must, at some level, be transformed into determinism, given that the physical computing machine is deterministic. This can obviously be done in various ways, and in principle does not cause loss of solutions. For example, we could think of starting k parallel computations when there are k possible clauses for a predicate[19] and therefore consider the results of all the possible computations.

In PROLOG, however, for reasons of simplicity and implementation efficiency, the strategy we first saw is employed: clauses are used according to the textual order in which they occur in the program (top-to-bottom). We saw in the previous example that this strategy is *incomplete*, given that it does allow us to find all possible computed answers. This limit, however, is adjustable by the programmer who, knowing this property of PROLOG, can order the clauses in the program in the most convenient way (typically putting first those relating to terminal cases and then the inductive steps). Note, though, that this trick, at least in principle, eliminates some of the declarative nature of the language, because the programmer has specified an aspect of control.

In addition to infinite computations, there is a second, more important aspect to be considered by adopting the deterministic model of PROLOG and deals with the handing of failures. Let us first see an example. Let us consider the program `Pb`:

```
p(X):- X=f(a).
p(X):- X=g(a).
```

Let us consider the evaluation of the goal `p(g(Y))`. By the PROLOG strategy, clause (1) is chosen, which, using the substitution $\{X/g(Y)\}$, produces the new goal `g(Y) = f(a)`. This fails, given that the two terms in the equation cannot be unified. Moreover, given that there is still one clause to use, it would not be acceptable to terminate the computation by returning a failure. "Backtracking" then occurs, returning to the choice of the clause for `p(g(Y))` and therefore continues the computation by trying clause (2). In this way, the computation achieves success with the computed answer $\{Y/a\}$.

In general, therefore, when arriving at a failure, the PROLOG abstract machine "backtracks" to a previous choice point at which there are other choices; that is, where there are other clauses to test. In this backtracking process, the bindings that might have been computed in the previous computation must be undone. Once a choice point has been reached, a new clause is tried and the computation continues in the way we have seen. If the previous choice point contains no possibilities for computations, an older choice point is sought and, if there are no more, the computation terminates in failure. Note that all of this is handled directly by the PROLOG abstract machine and is completely invisible to the programmer (except the use of particular constructs such as the cut which we will introduce in Sect. 12.5.1).

It is also easy to see how this procedure, which uses a search corresponding to a depth-first search through a tree representing the possible computations, can be computationally demanding. The solution to the problem seen in Sect. 12.1.1, for example, requires extensive backtracking and is fairly wasteful in computation time.

Let us now see another example. Consider the program `Pc`:

[19]Clearly, on a machine with one processor, the k parallel computations must be appropriately "scheduled" so they can be executed in a sequential manner, analogous to what happens with processes in a multitasking operating system.

```
p(X):- X=f(Y), q(X).                                                    1
p(X):- X=g(Y), q(X).                                                    2
q(X):- X=g(a).                                                          3
q(X):- X=g(b).                                                          4
```

Let us analyse the goal `p(Z)`. Using clause (1), we obtain the goal `Z=f(Y),q(Z)`. The evaluation of the equation produces the m. g. u. $\{Z/f(Y)\}$ and we obtain the goal `q(f(Y))`. Using clause (3), we obtain the goal `f(Y) = g(a)`, which fails. At this point we must turn to the last choice point, that is to the point where the choice of clause for the predicate `q` occurred. In this case, there are no bindings to undo, and therefore we try clause (4), obtaining therefore the goal `f(Y)=g(b)` which also fails. We return again to the choice point for `q` and we see that there are no other possible clauses and therefore we return to the previous choice point (the one for predicate `p`). Doing this, we have to undo the binding $\{Z/f(Y)\}$ calculated by clause (1) and therefore we return to the initial situation, where variable `Z` is not instantiated. Using clause (2), we obtain the goal `Z=g(Y),q(Z)` and therefore by the evaluation of the equation, we obtain the m.g.u. $\{Y/a\}$ and the new goal `q(g(Y))`. At this point, using clause (3), we obtain the goal `g(Y)=g(a)` which succeeds and produces the m. g. u. $\{Y/a\}$. Given that there remain no more goals to evaluate, the computation terminates with success. The result of the computation is produced by composing the computed m. g. u. s $\{Z/g(Y)\}\{Y/a\}$ and restricting the substitution $\{Z/g(a),\ Y(a)\}$ obtained by this composition to the single variable present in the initial goal. Therefore, the computed answer obtained is $\{Z/g(a)\}$. Note that there is another computed answer, $\{Z/g(b)\}$, which we can obtain using clause (4) rather than (3). In PROLOG implementations, answers subsequent to the first can be obtained using the ";" command. Finally, the reader can easily check that the goal `p(g(c))` in program `Pc` terminates with failure, a result that is obtained after having proved all four of the possible clause combinations.

12.4.5 Some Examples

In this subsection, we will focus on the PROLOG language, and we will make use of its syntax. The notation `[h|t]` is used to denote the list which has `h` as its head and `t` as its tail. Let us remember that the head is the first element in the list, while the tail is what remains of the list when the head has been removed. The empty list is written `[]`, while `[a,b,c]` is an abbreviation for `[a | [b | [c |[] ]]]` (the list composed of elements `a`, `b` and `c`). Note that in PROLOG, as in pure logic programming language, there exists no list type, for which the binary function symbol `[|]` can be used for terms that are not lists. For example, we can also write `[a | f(a)]` which is not a list (because `f(a)` is not a list).

As the first example, let us consider the following program `member` which checks whether an element belongs to a given list:

```
member(X, [X | Xs]).
member(X, [Y | Xs]) :- member(X, Xs).
```

The declarative reading of the program is immediate: Clause (1) consists of the terminal case in which the element that we are looking for (the first argument of the `member` predicate) is the head of the list (the second argument to the `member` predicate). Clause (2) instead provides the inductive case and tells us that `X` is an element of the list `[Y|Xs]` if it is an element of the list `Xs`.

Formulated in this way, the `member` program is similar to the one that we could write in any language that supports recursion. However, let us note that, unlike in the imperative and functional paradigms, this program can be used in two different ways.

The more conventional mode is the one in which it is used as a test. In a PROLOG system, once the previous program has been input, we have:

```
?- member(hewey, [dewey, hewey, louie]).
Yes
```

where `?-` is the abstract machine's prompt, to which we have added the goal whose evaluation we require. The next line contains the interpreter's answer. In this case, we have a simple "boolean" answer which expresses the existence of a successful computation of our goal. Moreover, as we know, we can also use the program to compute. For example, we can ask for the evaluation of:

```
?- member(X, [dewey, hewey, louie]).
X = dewey
```

The abstract machine returns $\{X/dewey\}$ as the computed answer. We can also obtain more answers using the ";" command. When there are no more answer, the system replies "no".

Finally, even if it is used in a rather unnatural fashion, we can use the first argument to instantiate the list that appears as the second argument. For example:

```
?- member(dewey, [X, hewey, louie]).
X = dewey
```

This possibility of using the same arguments as inputs or as outputs according to the way in which they are instantiated is unique to the logic paradigm and is due to the presence of unification in the computational model.

We can clarify this point further by considering the following `append` program which allows us to concatenate lists:

```
append([], Ys, Ys).                                   1
append([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).    2
```

In this case, too, the declarative reading is immediate. If the first list is empty, the results is the second list (clause(1)). Otherwise, (inductively), if `Zs` is the result of the concatenation of `Xs` and `Ys`, the result of the concatenation of `[X|Xs]` and `Ys` is obtained by adding `X` to the head of the list `Zs`, as indicated by `[X|Zs]`.

The normal use of the this program is that illustrated by the following goal:

```
?- append([dewey, hewey], [louie, donald], Zs).
Zs = [dewey, hewey, louie, donald]
```

Moreover, we can use append also to know how to subdivide a list into sublists, something that is not possible in a functional or imperative program:

```
?- append(Xs, Ys, [dewey, hewey]).
Xs = []
Ys = [dewey, hewey];
Xs = [dewey]
Ys = [hewey];
Xs = [dewey, hewey]
Ys = [];
no
```

Here, as before, the ";" command causes the computation of new solutions.

As a third example, we give the definition of the sublist predicate (we used this in Sect. 12.1.1). If Xs is a sublist of Ys, then there exist another two (possibly empty) lists As and Bs such that Ys is the concatenation of As, Xs and Bs. We note that this means that Xs is the suffix of a prefix of Ys. Hence using append, we can define sublist as follows:

```
sublist(Xs, Ys) :- append(As, XsBs, Ys), append(Xs, Bs, XsBs).
```

This program, combined with the append program and the sequence program of Sect. 12.1.1 allow us to solve the problem stated in Sect. 12.1.1. Note the conciseness and simplicity of the resulting program. Clearly other definitions of sublist are possible (see Exercise 8).

As a last example, let us consider a program that solves the classical problem of the Towers of Hanoi. We have a tower (in Computer Science terms, we would say a stack) composed of *n* perforated discs (of different diameters), arranged on a pole in order of decreasing diameter and we have 2 free poles. The problem consists of moving the tower to another pole, recreating the initial order of the discs. The following rules apply. The disks can be moved only from one pole to another. Only one disc at a time can be moved from one pole to another and the top disc must be taken from a pole. A disc cannot be put onto a smaller disc.

According to legend, this problem was assigned by the Divinity to the monks of a monastery near to Hanoi. There were three poles and 64 discs of gold. The solution to the problem would have signalled the end of the world. Given that the optimal solution requires time exponential in the number of discs, even if the legend comes true, we can still remain calm for a while: 2^{64} is a large enough number.

The following program solves the problem for an arbitrary number of disc, N, and three poles called A, B and C. It uses the coding of the natural numbers in terms of 0 and successor that we saw above.

```
hanoi(s(0), A, B, C [move(A,B)]).
hanoi(s(N), A, B, C, Moves):-
  hanoi(N, A, C, B, Moves1),
  hanoi(N, C, B, A, Moves2),
  append(Moves1, [move(A,B)|Moves2], Moves).
```

The call `hanoi(n, A, B, C, Move)`, where `n` is a (term which represents a) natural number, solves the problem of moving the tower of discs from `A` to `B` using `C` as an auxiliary pole. The solution is contained in the `Move` variable which, when the computation terminates, is instantiated with the list containing move which contribute to the solution. Every move is represented by a term of the form `move(X,Y)` to indicate the move of the top disc of pole `X` to pole `Y`.

This amazingly simple program amply shows the power of logic programming and also recursive reasoning. As usual, we give its logical interpretation. The first clause is clear: if we have just one disc, all we have to do is to move it from `A` from `B`. The reading of the second clause also is quite intuitive. If `Moves1` is the list of moves that solves the problem of moving a tower of `N` discs from `A` to `C` using `B` as an auxiliary pole and `Moves2` is the list of moves needed to move the tower of `N` disc from `C` to `B` using `A` as an auxiliary pole, to solve our problem in the case of `N+1` (or `s(N)`) discs, clearly we would have to do the following: first, execute all the moves in `Moves1`, then move from `A` to `B` (with the move `move(A,B)`) the `N+1`st disc in `A` which, being the largest, must be the last (at the bottom) in `B` as well. Finally, execute all the moves in `Moves2`. This is exactly what is done by the `append` predicate which therefore will use the variable `Moves` to provide the solutions to this problem.

12.5 Extensions

Until now, we have seen pure logic programming and some, highly partial, aspects of PROLOG. In this last section, we will briefly outline each of the numerous extensions to the pure formalism, each of which would require an entire chapter to do it justice. In the bibliographical notes we have included some references for those wishing to learn more.

12.5.1 Prolog

PROLOG is a language that is much richer than it might appear from what has been said so far. Let us recall that, as already seen, this language differs from logic programming in its adoption of precise rules for selecting the next atom to be rewritten (left to right) and for selecting the next clause to use (top to bottom, according to the program text).

In addition to these, there are other important differences with the theoretical formalism. We will list some of these without pretending to be exhaustive.

Arithmetic A real language cannot allow itself the luxury of using a completely symbolic arithmetic in which natural numbers are represented as successors of zero and there are no predefined operators.

In PROLOG, there exist therefore integers and reals (as floating point) as predefined data structures and various operators for manipulating them. These include:

- The usual arithmetic operators $+$, $-$, $*$ and $//$ (integer division).
- Arithmetic comparison operators such as $<$, $<=$ $>=$, $=:=$ (equal), $= \backslash =$ (different).
- An evaluation operator called `is`.

Unlike other symbols (function and predicate) used in PROLOG, these symbols use infix notation for ease of use. There are however numerous delicate aspects that require a deep knowledge to avoid simple errors which the programmer used to imperative languages can easily cause.

First, comparison operators always require that their operands are ground arithmetic expressions (they cannot contain variables). Thus, while we have:[20]

```
?- 3*2 =:= 1+5
Yes
? 4 > 5+2
no
```

if we use terms that are not arithmetic expressions or which contain variables, error will be signalled, as in:

```
?- 3 > a
error in arithmetic expression: a is not a number
?- X =:= 3+5
instantiation fault.
```

The last example is particularly disturbing. What we want to say is that the evaluation of the arithmetic expression `3+5` produces the value `8` which will therefore be bound to the variable `X`. This cannot be done using `=:=`, as seen above, nor can it be done using equality between terms as considered in the previous paragraph. If we write `X = 3+5`, indeed, the effect is that of binding `X` to the term `3+5` (rather than the value `8`). To avoid this problem, the expression evaluation operator `is` is introduced. This operator allows us to obtain the effect we desire; `s is t` indeed *unifies* `s` with the *value* of the ground arithmetic expression `t` (if `t` is not a ground arithmetic expression, we have an error). The following are some examples of the use of `is`:

```
?- X is 3+5;
X = 8
?- 8 is 3 + 5
```

[20] As said above, the line with `?-` returns the query that we want to evaluate and the following one is the PROLOG interpreter's reply.

```
Yes
?- 6 is 3 * 3
no
?- X is Y* 2
error in arithmetic expression: Y*2 is not a number
```

The use of `is`, necessary to be able to evaluate an expression, makes the use of arithmetic in PROLOG rather complicated. For example, given the following program:

```
evaluate(0,0).
evaluate(s(X), Val+1) :- evaluate(X, Val)
```

The goal `evaluate(s(s(s(0))), X)` does not compute the value `3` for `X`, as perhaps would have been expected but the term `0+1+1+1` (see Exercise 9 at the end of this chapter, too).

Cut PROLOG provides various constructs for interaction with the abstract PROLOG machine's interpreter so that the normal flow of control can be modified. Among these, one of the more important (and most discussed) is *cut*. This is an argument-free predicate, written as an exclamation mark, which allows the programmer to eliminate some of the possible alternatives produced during evaluation, with the aim of increasing execution efficiency. It is used when we are sure that, when a condition is satisfied, the other clauses in the program are no longer useful. For example, the following program computes the minimum of two values, if a comparison condition is true, the other is necessarily false, so we can use `!` to express the fact that once the first clause has been used, there is no need to consider the second.

```
minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y) :- X > Y.
```

Alternatively, if we are interested only in testing whether a value appears at least once in a list, we can use the following modification to the `member` program:

```
member(X, [X | Xs]) :- !.
member(X, [Y | Xs]) :- member(X, Xs).
```

In general, the meaning of cut is the following. If we have *n* clauses to define the predicate *p*

```
p(S1) :- A1.
...
p(Sk) :- B, !, C.
...
p(Sn) :- An.
```

if, during the evaluation of the goal `p(t)`, we find the *k*th clause in the list being used, we have the following cases:

1. If an evaluation of `B`[21] fails, then we proceed by trying the $k+1$st clause.
2. If, instead, the evaluation of `B` succeeded, then `!` is evaluated. It succeeds (it always does) and the evaluation proceeds with `C`. In the case of backtracking, however, all the alternative ways of computing `B` are eliminated, as well as are eliminated all the alternatives provided by the clauses from the kth to the nth to compute `p(t)`.

There is no need to say that cut, in addition to not being easy to use, eliminates a good part of the declarativeness of a program.

Disjunction If we want to express the disjunction of two goals, `G1` and `G2`, that is the fact that it is sufficient that at least one of the two succeeds, we can use two clauses of the form:

```
p(X) :- G1.
p(X) :- G2.
```

with the goal `p(X)`. The same effect can be obtained writing `G1;G2`, where we have used the predefined predicate ";", which represents disjunction.

If-then-else The traditional construct from imperative languages:

```
If B then C1 else C2
```

is provided in Prolog as a built-in with syntax `B -> C1;C2`. The `if then else` can be implemented using the cut as follows:

```
if_then_else(B, C1, C2) :- B, !, C1.
if_then_else(B, C1, C2) :- C2.
```

In effect, this is the internal definition of the `B -> C1;C2` construct.

Negation Thus far, we have seen that in the body of a clause, only "positive" atomic formulæ can be used; in other words, they cannot be negated.[22] Often, however, it can also be useful to use negated atomic formulæ. For example, let us consider the following program, `flights`:

```
direct_flight(bologna, paris).
direct_flight(bologna, amsterdam).
direct_flight(paris, bombay).
direct_flight(amsterdam, moscow).
flight(X, Y):- direct_flight(X, Y).
flight(X, Y):- direct_flight(X, Z), flight(Z, Y).
```

[21]Here, `B`, just like `Ai` and `C`, is considered as an arbitrary goal and not necessarily an atomic one.

[22]Here, we refer, clearly, to the $H : -A_1, \ldots, A_n$ notation. If, instead, we consider a clause as a disjunction of literals, the atoms in the body are negated, given that the preceding representation is equivalent to $\neg A_1 \vee \cdots \vee \neg A_n \vee H$.

Here, we have a series of facts defining the `direct_flight` predicate, which denotes the existence of a connection without stopover between two destinations. Then, we have the `flight` predicate which defines a flight, possibly with intermediate stops, between two locations. This will be a direct flight (clause 5) or rather a direct flight for a stopover different from the one that is the destination, followed by a flight from this stopover to the destination (clause 6). With this program, we can check the existence of a flight, or we can ask which destinations can be reached from a given airport:

```
?- flight(bologna, bombay)
Yes
?- direct_flight(bologna, moscow)
no
?- flight(bologna, X)
X = paris
```

However, we fail to express the fact that there exists only non-direct flights because we do not know how to express the fact that there *does not* exist a direct flight. To do this, we require negation. If we write:

```
indirect_flight(X, Y) :- flight(X, Y), not direct_flight(X, Y).
```

we mean to say that there exists an indirect flight between `X` and `Y` if there exists a flight (between `X` and `Y`) and there exist no direct flight. With this definition, we have:

```
?- indirect_flight(bologna, bombay)
Yes
?- indirect_flight(bologna, paris)
no
```

These results can be explained as follows. The PROLOG interpreter evaluates a goal such as `not G` by trying to evaluate the un-negated goal `G`. If the evaluation of this goal terminates (possibly after backtracking) with failure, then the goal `not G` succeeds. If, on the other hand, the goal `G` has a computation that terminates with success, then that of `not G` fails. Finally, if the evaluation of `G` does not terminate, then `not G` fails to terminate. This type of negation is called "negation as failure",[23] in that, exactly, it interprets the negation of a goal in terms of failure of the un-negated goal. Note that, because of infinite computations,[24] this type of negation differs from the negation in classical logic, given that the lack of success of `G` is not equivalent to the success of the negated version `not G`. For example, neither the goal `p` nor the goal `not p` succeed in the program `p :- p`.

[23] To be more precise, negation as failure, as defined in the theoretical model of logic programming, has a behaviour that is slightly different from that described because of the incompleteness of the PROLOG interpreter.

[24] And also non-ground goals, which we will not consider here.

12.5.2 Logic Programming and Databases

The `flight` program above indirectly indicates a possible application of logic programming in the context of databases.

A set of unit clauses, such as those defining the `direct_flight` predicate, is indeed, to all effects, the explicit (or extensional) definition of the relation denoted by this predicate. In this sense, this set of unit clauses can be seen as the analogue of a relation in the relational database model. To express a query, while in the relational model we would use relational algebra with the usual operations of selection, projection, join, etc., or more conveniently, a data manipulation language such as SQL, in the logic paradigm we can use the usual computational mechanisms that we have already seen. For example, to know if there exists a direct flight that arrives at Paris, we can use the query `direct_flight(X,paris)`. The predicates defined by the non-unit clauses such as `flight`, define relations in an implicit (or intensional) manner. They, indeed, allow us to compute new relations which are not explicitly stored, through the mechanism of inference as we have seen. Using database terminology, we will say that these predicates define "views" (or virtual relations).

Note, moreover, how in the `flights` program, function symbols are not used. Indeed, if we want only to manipulate relations, as is the case in relational algebra, function symbols are not needed.

These considerations were crystallised in the definition of *Datalog*, a logic language for databases. In its simplest form, it is a simplified version of logic programming in which:

1. There are no function symbols.
2. Extensional and intensional predicates are distinguished, as are comparison predicates.
3. Extensional predicates cannot occur in the head of clauses with a non-empty body.
4. If a variable occurs in the head, it must occur in the body of a clause.
5. A comparison predicate can occur only in the body of a clause. The variables occurring in such a predicate must occur also in another atom in the body of the same clause.

The last two conditions deal with the specific evaluation rule which Datalog uses[25] and will here will be ignored. The distinction between extensional and intensional predicates corresponds to the intuitive idea that has already been discussed. Extensional predicates, as condition (3) implies, can be defined only by facts.

The `flights` program can therefore be considered as a Datalog program, to all intents and purposes.

The reader who knows SQL or relational algebra will have no difficulty in understanding how the presence of recursion increases the expressive power of Datalog beyond that of these other formalisms. The same program, `flights`, provides us

[25]Datalog adopts a kind of bottom-up evaluation mechanism for goals. This differs from the top-down one we saw for logic programming. The results obtained are, however, the same.

with an example in this sense: the query `flight(bologna, X)` allows us to find all the destinations that can be reached from Bologna with an arbitrary number of intermediate stops. This query is not expressible in relational algebra or in SQL (at least, in its initial version) since the number of joins that we must create depends on the number of intermediate stops and, in general, not knowing how many of them there are (we are making no assumptions about how relations are structured), we cannot define such a number *a priori*.

12.5.3 Logic Programming with Constraints

To conclude, we will briefly discuss logic programming with constraints, or CLP (*Constraint Logic Programming*), an extension of logic programming which adds sophisticated constraint-satisfaction mechanisms to the mechanisms we have already seen. The resulting paradigm is certainly interesting for practical application and is today used in different application areas.

A constraint is nothing more than a particular first-order logic formula (normally we use a conjunction of atomic formulæ) which uses only predefined predicates. An example of constraint has already been given: if we write in a logic programming: `X=t, Y= f(a)`, we have used constraints, where the predicate symbol, `=`, as we know, is interpreted as syntactic equality over the Herbrand Universe,[26] while the comma indicates logical conjunction.

The idea of constraint logic programming is that of replacing the Herbrand Universe with another computational domain which usually is symbolic, but can also be arithmetic in some cases. Such a domain is suitable for the specification of the application area of interest. Constraints, rather than relations between ground terms, define relations over the values in the new domain under consideration. Correspondingly, the basic computational mechanism will no longer be the solution of equations between terms (using unification) but will be composed of a constraint-solving mechanism, that is an appropriate algorithm for determining solutions to the constraints to be solved. For example, we could consider the real numbers as the domain of computation, conjunctions of linear equations as constraints and use the Gauss-Jordan method as the constraint solver.

The principal advantage of this approach is clear. We can integrate into the logic paradigm (in a semantically clean fashion) very powerful computational mechanisms that were developed in other contexts (such as linear programming, operation research, etc.). Particularly interesting domains for the practical application of constraints are, as mentioned, the reals[27] and, above all, finite domains in which variables can take a finite number of values. In this last case, we also speak of *Constraint*

[26]Recall that this is the set of ground terms.

[27]Clearly, as always, when dealing with computers, what is represented is, in effect, a subset of the reals.

Satisfaction Problem (CSP). There exist many specific algorithms for the solution of this type of problems.

To obtain an idea of the possibilities of constraint logic programming, consider the problem of defining the amount of the instalments of a loan. The variables involved in this problem are the following: `F` is the requested financing, or the initial sum loaned, `NumR` is the number of the instalment, `Int` is the amount of interest, `Rate` is the amount of a single instalment and, finally, `Deb` is the remaining debt. The relation between these variables is intuitively the following. In the case in which there is no amount paid back (or that `NumR = 0`), clearly the debt is equal to the initial financing:

```
Deb = Fin.
```

In the case of the payment of a single instalment, we have the following relation:

```
Deb = Fin + Fin*Int - Rate
```

Here, the remaining debt is the initial sum to which we have added the interest accrued and have subtracted the amount repaid in an instalment. In the case of two instalments, matters are more complicated because interest is calculated on the remaining debt and not on the initial sum. The remaining debt, then, becomes the new financing. Using the `NuFin1` and `NuFin2` variables to indicate this new value of financing, we then have therefore the relation:

```
NuFin1 = Fin + Fin*Int - Rate
NuFin2 = NuFin1 + Fin*Int - Rate
Deb = NuFin2
```

In the general case, we can reason recursively. We have therefore the following constraint program, which we will call `loan`:

```
loan(Fin, NumR, Int, Rate, Deb):-
  NumR = 0,
  Deb = Fin.

loan(Fin, NumR, Int, Rate, Deb):-
  NumR >= 1,
  NuFin = Fin+Fin*Int-Rate,
  NuNumR = NumR-1,
  loan(NuFin, NuNumR, Int, Rate, Deb).
```

Understanding this program, given what we have already said, should not be too difficult.

Such a program can be used, just like logic programs, in many ways. For example, if we take out a loan of 1000 Euros and having paid 10 instalments of 150 Euros each at 10% interest, we can use it to find out what the remaining debt is, The query to solve this problem is:

```
loan(1000, 10, 10/100, 150, Deb)
```

This is evaluated and produces the result `Deb = 203.13`.

We can also use the same program in an inverse fashion. That is, rather finding out what the requested funding was, knowing that we have paid the same 10 instalments at 150 Euros each, again at 10% interest, but with residual debt of 0. The goal in this case is:

```
loan(Fin, 10, 10/100, 150, 0)
```

This yields the result `Fin = 921.685`.

Finally, we can also leave more than one variable in the goal, for example to determine the relation that holds between financing, instalment and debt, knowing that we pay 10 instalments at 10% interest. We can therefore formulate the query:

```
loan(Fin, 10, 10/100, Rate, Deb)
```

With appropriate assumptions on the constraint solver[28] we obtain the result:

```
Fin = 0.3855 * Deb + 6.1446 * Rata.
```

That is, and this is unique to the various paradigms that we have seen so far, it allows us to obtain a relation on the numerical domains as a result.

12.6 Advantages and Disadvantages of the Logic Paradigm

The few examples given in this chapter must however be sufficient to show that logic languages require a programming style that is significantly different from that of traditional languages and they also have a different expressive power. Clearly, given that every language (or almost) which have been named in this text is a Turing-complete formalism, it is not true that PROLOG or CLP or any other logic language allows the "computation of more things" than a common imperative language. The expressive power to which we refer is of a pragmatic kind, as we will make clearer in this section.

First, it is worth repeating that logic languages, in a way analogous to functional languages, allow us express solutions even to very complex problems in a "purely declarative" way, and therefore in an extremely simple and compact way. In our specific case, these aspects are further reinforced by three unique properties of the

[28]The `loan` program is written using the syntax of CLP(R). The adaptation to other CLP systems, even if they are defined over the reals, can require fairly substantial modifications even if the basic idea is always the same. In particular, the possibility of obtaining the results just discussed depends a great deal on the power of the constraint solver adopted by the CLP language. The results given here are derived from [9].

logic paradigm: (i) the ability to use a program in more than one way by transforming input arguments into outputs and viceversa; (ii) the possibility of obtaining a complex relationship between variables as a result, which implicitly expresses an infinite number of solutions (given by all the values of the variables that satisfy the relation); (iii) the possibility of reading a program directly as a logical formula.

The first characteristic derives from the fact that the basic computational mechanism, unification, is intrinsically bi-directional. This does not happen either with assignment in imperative languages, or with pattern matching in functional languages because both of these mechanisms presuppose a direction in variable bindings. This flexibility clearly allows us to use every program to its best, avoiding the duplication of code for modifications that are often only of a syntactic nature.

The second property of the logic paradigm is clear in the case of constraint logic languages, as we saw in the last program of the last section. This aspect is particularly important to all application areas in which a single solution, understood as a set of specific values of interesting variables, either cannot be obtained (because the data do not contain enough information), or is not interesting, or is particularly difficult to obtain computationally.

The third aspect, finally, at least in principle, makes it easier to verify the correctness of logic programs than imperative ones, where a logical reading of programs is possible but requires much more expressive (and complicated) tools. As already emphasised in Sect. 11.5, this aspect is essential to be able to obtain formal correctness-verifying tools that are easily usable on programs of significant size.

These three aspects, which we can summarise in the principle of "computation as deduction", allow us to express in a very natural fashion forms of reasoning that are typical of the problems that we encounter in artificial intelligence and knowledge representation (for example, in the context of the Semantic Web). The fact (in those languages that have been implemented) that control is handled entirely by the abstract machine through backtracking, allows us, for example, extremely easily to express problems that involve aspects of search in a space of solutions.

Logic languages can also profitably be used as tools for the rapid prototyping of systems. Using PROLOG, it is possible to describe in a very short period and in a compact way, even very complex systems, with the advantage, as far as specification languages are concerned, that it can be executed.

The use of sophisticated constraint solvers and also optimisers (for example, based on the Simplex algorithm) makes constraint logic programs competitive with respect to other formalisms in many commercial applications where solutions to combinatorial problems, solutions to optimisations and more generally solutions to problems expressible as relations over appropriate domains are required. Examples in this sense, include scheduling problems and electrical circuit diagnosis; commercial mathematics and financial planning; civil engineering and others (for other examples, the reader should consult the text cited in the Bibliographical Notes at the end of this chapter).

Thus far, advantages. Beside them, however, we can record various negative aspects of logic languages, at least in the classical versions that currently exist.

First, as already noted in this chapter, the control mechanism based on backtracking is of limited efficiency. Various devices can be used to improve this aspect, be it at the level of constructs that can be used in programs, or at the implementation level. For example, abstract machines specific to logic languages such as the WAM (Warren Abstract Machine) have been defined and have been variously optimised. However, the efficiency of a PROLOG program remains fairly limited.

The absence of types or modules is a second bad point, even if in this sense some more recent languages have proposed partial solutions (for example, the logic language, Mercury). From what was said in Chap. 8, it should be clear that the lack of an adequate type system makes program correctness checking relatively difficult, despite the possibility of declarative reading.

Also, the arithmetic constructs and, in general, built-ins in PROLOG certainly do not facilitate the correctness of programs. This is because their use is not easy due to various semantic subtleties. This is particularly evident in the control constructs, which frequently force us to choose between a clear program that is not very efficient and an efficient program that is, however, relatively hard to understand.

Finally, and for many reasons (not just a limited commercial interest), unfortunately logic languages almost always have a programming environment that is rather deficient, where there are few of the characteristics available, for example, in the sophisticated environments for object-oriented programming.

12.7 Chapter Summary

In this chapter, we have presented the primary characteristics of the logic programming paradigm, a relatively recent paradigm (which has developed since the mid-1970s), which implements the old aspiration of seeing computation as a process that is entirely governed by logical laws. With only one chapter available to us, we have had to limit ourselves to introductory aspects, however what has been presented has enough formal precision to illustrate the computational model for logic programming languages. In particular, we have seen:

- Some syntactic concepts in first-order logic which are necessary for the definition of clauses and therefore logic programs.
- The process of unification which comprises the basic computational mechanism. This required a few concepts relating to terms and substitutions. We have also seen a specific algorithm for unification.
- How computation in a logic language works using a rule of deduction called SLD Resolution. In the text, we have chosen an approach that was inspired by a procedural reading of the clauses that demonstrate the similarity with "normal" procedures in imperative languages. Two boxes more formally introduced the concepts.

After some examples, we saw some important extensions to the pure formalism, in particular:

- Some specific characteristics of the PROLOG language. We did not cover meta-programming, higher-order programming or constructs that manipulate the program at runtime, all of which are very important.

- The idea of the use of logic languages in connection with databases.
- Constraint logic languages (but only through a single example).

For more information on those aspects of PROLOG that we have omitted or for a better introduction to Datalog or to constraint-logic languages, the reader is recommended to the literature listed below.

12.8 Bibliographical Notes

Herbrand's ideas on unification are in his doctoral dissertation from 1930 [6]. The definition of the resolution rule and the first formalisation of the unification algorithm are in A. Robinson's work [11]. The unification algorithm that we have presented was introduced in [10]. Major details of theory of unification can be obtained from various articles and texts, among which is [5].

The "historic" paper by R. Kowalski which introduced SLD Resolution and therefore the theory of logic programming is [7]. K.L. Clark's results on correctness and completion are in [3].

There are many texts both on the theory of logic programming and on programming in PROLOG. Among the former, there is the text by Lloyd [8] and the more recent and detailed treatment by K. Apt [1]. For programming in PROLOG and for numerous (non-trivial) examples of programming, we advise the reader to consult the classic text by L. Sterling and E. Shapiro [12], from which we have taken the program that solves the problem of the Towers of Hanoi; there is also the book by H. Coelho and J.C. Cotta [4], from which we have taken the program in the example presented in Sect. 12.1.1.

Finally, as for Datalog and, more generally, the use of logic programs with databases, [2] can be consulted, while [9] provides a good introduction to languages with constraints.

12.9 Exercises

1. State a context-free grammar which defines propositional formulæ (that is those obtained by considering only predicates of arity 0 and without quantifiers).
2. Assume that we have to represent the natural numbers using `0` for zero and `s(n)` for the successor of `n`. State what is the answer computer by the following logic program for a generic goal `p(s,t,X)`, where `s` and `t` are terms that represent natural numbers:

```
p(0, X, X).
p(s(Y), X, s(Z)):- p(Y, X, Z).
```

3. State what are the answers computed by the following logic program for the goal `p(X)`:

```
p(0).
p(s(X)):- q(X).
q(s(X):- p(X).
```

4. Given the following logic program:

```
member(X, [X| Xs]).
member(X, [Y| Xs]):- member(X, Xs).
```

   State what is the result of evaluating the goal:

```
member(f(X), [1, f(2), 3] ).
```

5. Given the following PROLOG program (recall that `X` and `Y` are variables, while `a` and `b` are constants):

```
p(b):- p(b).
p(X):- r(b).
p(a):- p(a).
r(Y).
```

   State whether the goal `p(a)` terminates or not; justify your answer.
6. Consider the following logic program:

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

   State whether the goal `p(b)` terminates or not; justify your answer.
7. Assuming that the natural numbers are represented using `0` for zero and `s(n)` for the successor of `n` and using a primitive `write(x)` that writes the term `t`, write a logic program that prints all the natural numbers.
8. Define the `sublist` predicate in a direct fashion without using `append`.
9. Write in PROLOG a program that computes the length (understood as the number of elements) of a list and returns this value in numeric form. (Hint: consider an inductive definition of length and use the `is` operator to increment the value in the inductive case.)
10. List the principle differences between a logic program and a PROLOG program.
11. If in a logic program, the order of the atoms in the body of a clause is changed, is the semantics of the program altered? Justify your answer.
12. If in PROLOG, the clause-selection rule were changed (for example, always selecting the lowest instead of the highest) would it change the semantics of the language? Justify your answer.
13. Give an example of a logic program, P, and of a goal, G, such that the evaluation of G in P produces a different effect when two different selection rules are used. (Suggestion: given that we have seen that for computed answers, there is no difference in the use of different selection rules, consider what happens to computations that do not terminate and that fail.)

14. Informally describe a selection rule that allows us to obtain the least possible number of computations that do not terminate. (Suggestion: computations that do not terminate, by changing the selection rule, would become finite failures. Consider a rule that guarantees that all atoms in the goal are evaluated.)

References

1. K. Apt. *From Logic Programming to Prolog*. Prentice Hall, New York, 1997.
2. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, Berlin, 1989.
3. K. L. Clark. Predicate logic as a computational formalism. Technical Report Res. Rep. DOC 79/59, Imperial College, Dpt. of Computing, London, 1979.
4. H. C. Coelho and J. C. Cotta. *Prolog by Example*. Springer, Berlin, 1988.
5. E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.
6. J. Herbrand. *Logical Writings*. Reidel, Dordrecht, 1971. Edited by W.D. Goldfarb.
7. R. A. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.
8. J. W. Lloyd. *Foundations of Logic Programming*, 2nd edition. Springer, Berlin, 1987.
9. K. Marriott and P. Stuckey. *Programming with Constraints*. MIT Press, Cambridge, 1998.
10. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
11. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
12. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, 1986.