Introduction to PROLOG

Peter Lucas
Department of Computing Science
University of Aberdeen, Aberdeen

Contents

1 Introduction

PROLOG is a simple, yet powerful programming language, based on the principles of first-order predicate logic. The name of the language is an acronym for the French 'PROgrammation en LOGique'. About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille, influenced by the ideas of R.A. Kowalski concerning programming in the Horn clause subset of first-order predicate logic. The name of PROLOG has since then been connected with a new programming style, known as *logic programming*.

Until the end of the seventies, the use of PROLOG was limited to the academic world. Only after the development of an efficient PROLOG interpreter and compiler by D.H.D. Warren and F.C.N. Pereira at the University of Edinburgh, the language entered the world outside the research institutes. The interest in the language has increased steadily. However, PROLOG is still mainly used by researchers, even though it allows for the development of serious and extensive programs in a fraction of the time needed to develop a C or Java program with similar functionality. The only explanation is that people like waisting their precious time. Nevertheless, there are a large number of fields in which PROLOG has been applied successfully. The main applications of the language can be found in the area of Artificial Intelligence; but PROLOG is being used in other areas in which symbol manipulation is of prime importance as well. Some application areas are:

- Natural-language processing;
- Compiler construction;
- The development of expert systems;
- Work in the area of computer algebra;
- The development of (parallel) computer architectures;
- Database systems.

PROLOG is particularly strong in solving problems characterized by requiring complex symbolic computations. As conventional imperative programs for solving this type of problems tend to be large and impenetrable, equivalent PROLOG programs are often much shorter and easier to grasp. The language in principle enables a programmer to give a formal specification of a program; the result is then almost directly suitable for execution on the computer. Moreover, PROLOG supports stepwise refinement in developing programs because of its modular nature. These characteristics render PROLOG a suitable language for the development of prototype systems.

There are several dialects of PROLOG in use, such as for example, C-PROLOG, SWI-PROLOG, Sicstus-PROLOG, LPA-PROLOG. C-PROLOG, also called Edinburgh PROLOG, was taken as a basis for the ISO standard. C-PROLOG itself is now no longer in use.

The language definition of C-PROLOG is derived from an interpreter developed by D.H.D. Warren, D.L. Bowen, L. Byrd, F.C.N. Pereira, and L.M. Pereira, written in the C programming language for the UNIX operating system. Most dialects only have minor syntactical and semantical differences with the standard language. However, there are a small number of dialects which change the character of the language in a significant way, for example by the necessity of adding data-type information to a program. A typical example is offered by the version of the PROLOG language supported by Visual PROLOG. In recent versions of PROLOG, several features have been added to the ISO standard. Modern PROLOG versions provide a module concept and extensive interfaces to the operating system, as well as tools for the development of graphical user interfaces. As these have not been standardized, we will not pay attention to them here.

2 Logic programming

In more conventional, imperative languages such as C++, Java and Pascal, a program is a specification of a sequence of instructions to be executed one after the other by a target machine, to solve the problem concerned. The description of the problem is incorporated implicitly in this specification, and usually it is not possible to clearly distinguish between the description of the problem, and the method used for its solution. In logic programming, the description of the problem and the method for solving it are explicitly separated from each other. This separation has been expressed by R.A. Kowalski in the following equation:

$$\boxed{\text{algorithm} = \text{logic} + \text{control}}$$

The term 'logic' in this equation indicates the descriptive component of the algorithm, that is, the description of the problem; the term 'control' indicates the component that tries to find a solution, taking the description of the problem as a point of departure. So, the logic component defines *what* the algorithm is supposed to do; the control component indicates *how* it should be done.

A specific problem is described in terms of relevant objects and relations between objects, which are then represented in the clausal form of logic, a restricted form of first-order predicate logic. The logic component for a specific problem is generally called a *logic program*. The control component employs logical deduction or reasoning for deriving new facts from the logic program, thus solving the given problem; one speaks of the *deduction method*. The deduction method is assumed to be quite general, in the sense that it is capable of dealing with any logic program respecting the clausal form syntax.

The splitting of an algorithm into a logic component and a control component has a number of advantages:

- The two components may be developed separately from each other. For example, when describing the problem we do not have to be familiar with how the control component operates on the resulting description; knowledge of the declarative reading of the problem specification suffices.
- A logic component may be developed using a method of stepwise refinement; we have only to watch over the correctness of the specification.
- Changes to the control component affect (under certain conditions) only the efficiency of the algorithm; they do not influence the solutions produced.

An environment for logic programming offers the programmer a deduction method, so that only the logic program has to be developed for the problem at hand.

3 Programming in PROLOG

The programming language PROLOG can be considered to be a first step towards the practical realization of logic programming; as we will see in below, however, the separation between logic and control has not been completely realized in this language. Figure 1 shows the relation between PROLOG and the idea of logic programming discussed above. A PROLOG system consists of two components: a *PROLOG database* and a *PROLOG interpreter*.

A PROLOG program, essentially a logic program consisting of *Horn clauses* (which however may contain some directives for controlling the inference method), is entered into the PROLOG database by the programmer. The PROLOG interpreter offers a deduction method, which is based on a technique called *SLD resolution* (See [3] for details).

Solving a problem in PROLOG starts with discerning the objects that are relevant to the particular problem, and the relationships that exist between them.

Example. In a problem concerning sets, we for instance take constants as separate objects and the set as a whole as another object; a relevant relation between constants and sets is the membership relation. $\diamond$

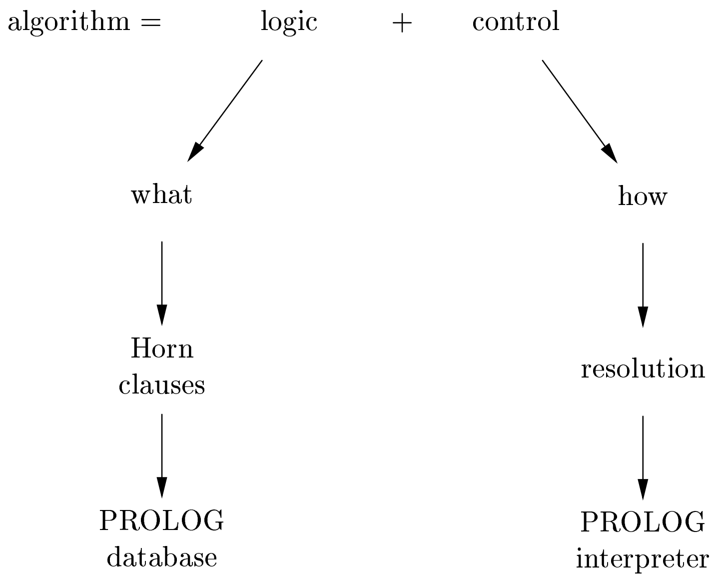

Figure 1: The relationship between PROLOG and logic programming.

When we have identified all relevant objects and relations, it must be specified which *facts* and *rules* hold for the objects and their interrelationships.

Example. Suppose that we are given a problem concerning sets. We may for example have the fact that a certain constant a is a member of a specific set S. The statement 'the set X is a subset of the set Y, if each member of X is a member of Y' is a rule that generally holds in set theory. $\diamond$

When all facts and rules have been identified, then a specific problem may be looked upon as a query concerning the objects and their interrelationships. To summarize, specifying a logic program amounts to:

- Specifying the *facts* concerning the objects and relations between objects relevant to the problem at hand;
- Specifying the *rules* concerning the objects and their interrelationships;
- Posing *queries* concerning the objects and relations.

3.1 The declarative semantics

Information (facts, rules, and queries) is represented in PROLOG using the formalism of *Horn clause logic*. A *Horn clause* takes the following form:

$$B \leftarrow A_1, \ldots, A_n$$

where B, $A_1, \ldots, A_n$, $n \geq 0$, are atomic formulas. Instead of the (reverse) implication symbol, in PROLOG usually the symbol `:-` is used, and clauses are terminated by a dot. An *atomic formula* is an expression of the following form:

$$P(t_1, \ldots, t_m)$$

Formal	Name	In PROLOG	Name
$A \leftarrow$	unit clause	A.	fact
$\leftarrow B_1, \ldots, B_n$	goal clause	?- $B_1, \ldots, B_n$.	query
$A \leftarrow B_1, \ldots, B_n$	clause	A:-$B_1, \ldots, B_n$.	rule

Table 1: Horn clauses and PROLOG.

where P is a *predicate* having m arguments, $m \geq 0$, and $t_1, \ldots, t_m$ are terms. A *term* is either a constant, a variable, or a function of terms. In PROLOG two types of constants are distinguished: numeric constants, called *numbers*, and symbolic constants, called *atoms*. (Note that the word atom is used here in a meaning differing from that of atomic formula, thus deviating from the standard terminology of predicate logic.) Because of the syntactic similarity of predicates and functions, both are called *functors* in PROLOG. The terms of a functor are called its *arguments*. The arguments of a functor are enclosed in parentheses, and separated by commas.

Seen in the light of the discussion from the previous section, the predicate P in the atomic formula $P(t_1, \ldots, t_m)$ is interpreted as the name of the relationship that holds between the objects $t_1, \ldots, t_m$ which occur as the arguments of P. So, in a Horn clause B `:-` $A_1, \ldots, A_n$, the atomic formulas B, $A_1, \ldots, A_n$, denote relations between objects. A Horn clause now is interpreted as stating:

'B (is true) if A_1 and A_2 and ... and A_n (are true)'

$A_1, \ldots, A_n$ are called the *conditions* of the clause, and B its *conclusion*. The commas between the conditions are interpreted as the logical $\wedge$, and the `:-` symbol as the (reverse) logical implication $\leftarrow$.

If $n = 0$, that is, if conditions A_i are lacking in the clause, then there are no conditions for the conclusion to be satisfied, and the clause is said to be a *fact*. In case the clause is a fact, the `:-` sign is replaced by a dot.

Both terminology and notation in PROLOG differ slightly from those employed in logic programming. Table 1 summarizes the differences and similarities. The use of the various syntactic forms of Horn clauses in PROLOG will now be introduced by means of examples.

Example. The PROLOG clause

```
/*1*/      member(X,[X|_]).
```

is an example of a fact concerning the relation with the name `member`. This relation concerns the objects `X` and `[X|_]` (their meaning will be discussed shortly). The clause is preceded by a comment; in PROLOG, comments have to be specified between the delimiters `/*` and `*/`. $\diamond$

If a clause contains one or more conditions as well as a conclusion, it is called a *rule*.

Example. Consider the PROLOG clause

```
/*2*/      member(X,[_|Y]) :- member(X,Y).
```

which is a rule concerning the relation with the name `member`. The conclusion `member(X,[_|Y])` is only subjected to one condition: `member(X,Y)`. $\diamond$

If the conclusion is missing from a clause, then the clause is considered to be a query to the logic program. In case a clause is a query, the sign `:-` is usually replaced by the sign `?-`.

Example. The PROLOG clause

```
/*3*/      ?- member(a,[a,b,c]).
```

is a typical example of a query. ◇

A symbolic constant is denoted in PROLOG by a name starting with a lower-case letter. Names starting with an upper-case letter, or an underscore sign, `_`, indicate *variables* in PROLOG. A relation between objects is denoted by means of a functor having a name starting with a lower-case letter (or a special character, such as &, not having a predefined meaning in PROLOG), followed by a number of arguments, that is the objects between which the relation holds. Recall that arguments are terms, that is, they may be either constants, variables, or functions of terms.

Example. Consider the three clauses from the preceding examples once more. `member` is a functor having two arguments. The names `a`, `b`, and `c` in clause 3 denote symbolic constants; `X` and `Y` are variables. ◇

In PROLOG, a collection of elements enclosed in square brackets denotes a *list*. It is possible to explicitly decompose a list into its first element, the *head* of the list, and the remaining elements, the *tail* of the list. In the notation `[X|Y]`, the part in front of the bar is the head of the list; `X` is a single element. The part following the bar denotes its tail; `Y` itself is a list.

Example. Consider the list `[a,b,c]`. Now, `[a|[b,c]]` is another notation for the same list; in this notation, the head and the tail of the list are distinguished explicitly. Note that the tail again is a list. ◇

Each clause represents a separate piece of knowledge. So, in theory, the meaning of a set of clauses can be specified in terms of the meanings of each of the separate clauses. The meaning of a clause is called the *declarative semantics* of the clause. Knowledge of the declarative semantics of first-order predicate logic helps in understanding PROLOG. Broadly speaking, PROLOG adheres to the semantics of first-order logic. However, there are some differences, such as the use of negation as finite failure which will be discussed below.

Example. Consider the clauses 1, 2 and 3 from the preceding examples once more. Clause 1 expresses that the relation with the name `member` holds between a term and a list of terms, if the head of the list equals the given term. Clause 1 is not a statement concerning specific terms, but it is a general statement; this can be seen from the use of the variable `X` which may be substituted with any term. Clause 2 represents the other possibility that the constant occurs in the tail of the list. The last clause specifies the query whether or not the constant `a` belongs to the list of constants `a`, `b`, and `c`. ◇

3.2 The procedural semantics and the interpreter

In the preceding section we have viewed the formalism of Horn clause logic merely as a formal language for representing knowledge. However, the Horn clause formalism can also be looked upon as a programming language. This view of Horn clause logic is called its *procedural semantics.*

In the procedural semantics, a set of clauses is viewed as a program. Each clause in the program is seen as a *procedure (entry).* In the clause

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

we look upon the conclusion B as the *procedure heading*, composed of a procedure name, and a number of formal parameters; $A_1, \ldots, A_n$ is then taken as the *body* of the procedure, consisting of a sequence of *procedure calls.* In a program all clauses having the same predicate in their conclusion, are viewed as various entries to the same procedure. A clause without any conclusion, that is, a query, acts as the *main program.* Here no strict distinction is made between both types of semantics; it will depend on the subject dealt with, whether the terminology of the declarative semantics is used, or the terminology of procedural semantics is preferred. In the remainder of this section we shall discuss the PROLOG interpreter.

When a PROLOG program has been entered into the PROLOG database, the main program is executed by the PROLOG interpreter. The way the given PROLOG clauses are manipulated, will be demonstrated by means of some examples.

Example. The three clauses introduced in Section 3.1 together constitute a complete PROLOG program:

```
/* 1*/     member(X,[X|_]).
/* 2*/     member(X,[_|Y]) :-
              member(X,Y).

/* 3*/     ?- member(a,[a,b,c]).
```

Clauses 1 and 2 are entries to the same `member` procedure. The body of clause 2 consists of just one procedure call. Clause 3 fulfills the role of the main program. ◇

Let us suppose that the PROLOG database initially contains the first two clauses, and that clause 3 is entered by the user as a query to the PROLOG system. The PROLOG interpreter tries to derive an answer to the query using the information stored in the database. To this end, the interpreter employs two fundamental techniques: matching and backtracking.

Matching of clauses

To answer a query, the PROLOG interpreter starts with the first condition in the query clause, taking it as a procedure call. The PROLOG database is subsequently searched for a suitable entry to the called procedure; the search starts with the first clause in the database, and continues until a clause has been found which has a conclusion that can be matched with the procedure call. A *match* between a conclusion and a procedure call is obtained, if there exists a substitution for the variables occurring both in the conclusion and in the procedure call, such that the two become (syntactically) equal after the substitution has been applied to them. Such a match exists

- If the conclusion and the procedure call contain the same predicate, and
- If the terms in corresponding argument positions after substitution of the variables are equal; one then also speaks of a match for argument positions.

Applying a substitution to a variable is called *instantiating* the variable to a term. The most general substitution making the selected conclusion and the procedure call syntactically equal, is called the *most general unifier* (*mgu*) of the two. The algorithmic and theoretical basis of matching is given by *unification* (See [3] for details).

If we have obtained a match for a procedure call, the conditions of the matching clause will be executed. In case the matching clause has no conditions, the next condition from the calling clause is executed. The process of matching (and instantiation) can be examined by means of the special infix predicate `=`, which tries to match the terms at its left-hand and right-hand side and subsequently investigates whether the terms have become syntactically equal.

Example. Consider the following example of the use of the matching predicate `=`. The first line representing a query has been entered by the user; the next line is the system's output.

```
?- f(X) = f(a).
X = a
```

As can be seen, the variable `X` is instantiated to `a`, which leads to a match of the left-hand and right-hand side of `=`. ◇

On first thoughts, instantiation seems similar to the assignment statement in conventional programming languages. However, these two notions differ considerably. An instantiation is a binding of a variable to a value which cannot be changed, that is, it is not possible to overwrite the value of an instantiated variable by some other value (we will see however, that under certain conditions it is possible to create a new instantiation). So, it is not possible to express by instantiation a statement like

$X := X + 1$

which is a typical assignment statement in a language like Pascal. In fact, the 'ordinary' assignment which is usually viewed as a change of the state of a variable, cannot be expressed in standard logic.

A variable in PROLOG has for its lexical scope the clause in which it occurs. Outside that clause, the variable and the instantiations to the variable have no influence. PROLOG does not have global variables. We shall see later that PROLOG actually does provide some special predicates which have a global effect on the database; the meanings of such predicates, however, cannot be accounted for in first-order logic. Variables having a name only consisting of a single underscore character, have a special meaning in PROLOG. These variables, called *don't-care variables*, match with any possible term. However, such a match does not lead to an instantiation to the variable, that is, past the argument position of the match a don't care variable looses its 'binding'. A don't care variable is usually employed at argument positions which are not referred to later in some other position in the clause.

Example. In our `member` example, the interpreter tries to obtain a match for the following query:

```
/*3*/     ?- member(a,[a,b,c]).
```

The first clause in the database specifying the predicate `member` in its conclusion, is clause 1:

```
/*1*/     member(X,[X|_]).
```

The query contains at its first argument position the constant `a`. In clause 1 the variable `X` occurs at the same argument position. If the constant `a` is substituted for the variable `X`, then we have obtained a match for the first argument positions. So, `X` will be instantiated to the constant `a`. As a consequence, the variable `X` at the second argument position of the conclusion of clause 1 has the value `a` as well, since this `X` is the same variable as at the first argument position of the same clause. We now have to investigate the respective second argument positions, that is, we have to compare the lists `[a,b,c]` and `[a|_]`. Note that the list `[a,b,c]` can be written as `[a|[b,c]]`; it is easily seen that we succeed in finding a match for the second argument positions, since the don't care variable will match with the list `[b,c]`. So, we have obtained a match with respect to the predicate name as well as to all argument positions. Since clause 1 does not contain any conditions, the interpreter answers the original query by printing `yes`:

```
/*3*/      ?- member(a,[a,b,c]).
yes
```

◇

Example. Consider again the clauses 1 and 2 from the preceding example. Suppose that, instead of the previous query, the following query is entered:

```
/*3*/      ?- member(a,[b,a,c]).
```

Then again, the interpreter first tries to find a match with clause 1:

```
/*1*/      member(X,[X|_]).
```

Again we have that the variable `X` will be instantiated to the constant `a`. In the second argument position of clause 1, the variable `X` also has the value `a`. We therefore have to compare the lists `[b,a,c]` and `[a|_]`: this time, we are not able to find a match for the second argument positions. Since the only possible instantiation of `X` is to `a`, we will never find a match for the query with clause 1. The interpreter now turns its attention to the following entry of the `member` procedure, being clause 2:

```
/*2*/      member(X,[_|Y]) :-
              member(X,Y).
```

When comparing the first argument positions of the query and the conclusion of clause 2 respectively, we infer that the variable `X` will again be instantiated to the constant `a`. For the second argument positions we have to compare the lists `[b,a,c]` and `[_|Y]`. We obtain a match for the second argument positions by instantiating the variable `Y` to the list `[a,c]`. We have now obtained a complete match for the query with the conclusion of clause 2. Note that all occurrences of the variables `X` and `Y` within the scope of clause 2 will have been instantiated to `a` and `[a,c]`, respectively. So, after instantiation we have

```
member(a,[_|[a,c]]) :-
   member(a,[a,c]).
```

Since, clause 2 contains a condition, its conclusion may be drawn only if the specified condition is fulfilled. The interpreter treats this condition as a new query:

```
?- member(a,[a,c]).
```

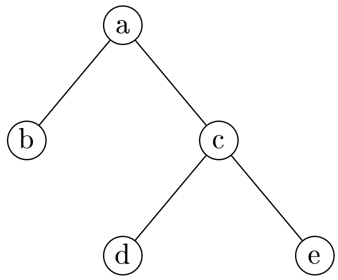

Figure 2: A binary tree.

This query matches with clause 1 in the same way as has been described in the previous example; the interpreter returns success. Subsequently, the conclusion of clause 2 is drawn, and the interpreter prints the answer `yes` to the original query. ◇

Backtracking

When after the creation of a number of instantiations and matches the system does not succeed in obtaining the next match, it systematically tries alternatives for the instantiations and matches arrived at so far. This process of finding alternatives by undoing previous work, is called *backtracking*. The following example demonstrates the process of backtracking.

Example. Consider the following PROLOG program:

```
/*1*/     branch(a,b).
/*2*/     branch(a,c).
/*3*/     branch(c,d).
/*4*/     branch(c,e).
/*5*/     path(X,X).
/*6*/     path(X,Y) :-
             branch(X,Z),
             path(Z,Y).
```

The clauses 1–4 inclusive represent a specific binary tree by means of the predicate `branch`; the tree is depicted in Figure 2. The symbolic constants `a`, `b`, `c`, `d` and `e` denote the vertices of the tree. The predicate `branch` in `branch(a,b)` has the following intended meaning: 'there exists a branch from vertex `a` to vertex `b`'.

The clauses 5 and 6 for `path` specify under which conditions there exists a path between two vertices. The notion of a path has been defined recursively: the definition of a path makes use of the notion of a path again.

A *recursive definition* of a relation generally consists of two parts: one or more *termination criteria*, usually defining the basic states for which the relation holds, and the actual recursion describing how to proceed from a state in which the relation holds to a new, simpler state concerning the relation.

The termination criterion of the recursive definition of the `path` relation is expressed above in clause 5; the actual recursion is defined in clause 6. Note that the definition of the `member` relation in the preceding examples is also a recursive definition.

Now, suppose that after the above given program is entered into the PROLOG database, we enter the following query:

```
/*7*/      ?- path(a,d).
```

The interpreter first tries to obtain a match with clause 5, the first clause in the database specifying the predicate `path` in its conclusion:

```
/*5*/      path(X,X).
```

For a match for the respective first argument positions, the variable `X` will be instantiated to the constant `a`. Matching the second argument positions fails, since `a`, the instantiation of `X`, and the constant `d` are different from each other. The interpreter therefore tries the next clause for `path`, which is clause 6:

```
/*6*/      path(X,Y) :- branch(X,Z),path(Z,Y).
```

It will now find a match for the query: the variable `X` occurring in the first argument position of the conclusion of clause 6 is instantiated to the constant `a` from the first argument position of the query, and the variable `Y` is instantiated to the constant `d`. These instantiations again pertain to the entire matching clause; in fact, clause 6 may now be looked upon as having the following instantiated form:

```
path(a,d) :- branch(a,Z),path(Z,d).
```

Before we may draw the conclusion of clause 6, we have to fulfill the two conditions `branch(a,Z)` and `path(Z,d)`. The interpreter deals with these new queries from left to right. For the query

```
?- branch(a,Z).
```

the interpreter finds a match with clause 1

```
/*1*/      branch(a,b).
```

by instantiating the variable `Z` to `b`. Again, this instantiation affects all occurrences of the variable `Z` in the entire clause containing the query; so, we have:

```
path(a,d) :- branch(a,b),path(b,d).
```

The next procedure call to be handled by the interpreter therefore is

```
?- path(b,d)
```

No match is found for this query with clause 5. The query however matches with the conclusion of clause 6:

```
/*6*/      path(X,Y) :- branch(X,Z),path(Z,Y).
```

The interpreter instantiates the variable `X` to `b`, and the variable `Y` to `d`, yielding the following instance of clause 6:

```
path(b,d) :- branch(b,Z),path(Z,d).
```

Note that these instantiations for the variables `X` and `Y` are allowed; the earlier instantiations for variables `X` and `Y` concerned *different* variables since they occurred in a different clause and therefore within a different scope. Again, before the query `path(b,d)` may be answered in the affirmative, we have to check the two conditions of the instance of clause 6 obtained. Unfortunately, the first condition

```
?- branch(b,Z).
```

does not match with any clause in the PROLOG program (as can be seen in Figure 2, there is no outgoing branch from the vertex `b`).

The PROLOG interpreter now cancels the last match and its corresponding instantiations, and tries to find a new match for the originating query. The match of the query `path(b,d)` with the conclusion of clause 6 was the last match found, so the corresponding instantiations to `X` and `Y` in clause 6 are cancelled. The interpreter now has to try to find a new match for the query `path(b,d)`. However, since clause 6 is the last clause in the program having the predicate `path` in its conclusion, there is no alternative match possible. The interpreter therefore goes yet another step further back.

The match of `branch(a,Z)` with clause 1 will now be undone by cancelling the instantiation of the variable `Z` to `b`. For the query

```
?- branch(a,Z).
```

the interpreter is able to find an alternative match, namely with clause 2:

```
/*2*/      branch(a,c).
```

It instantiates the variable `Z` to `c`. Recall that the query `branch(a,Z)` came from the match of the query `path(a,d)` with clause 6:

```
path(a,d) :- branch(a,Z),path(Z,d).
```

The undoing of the instantiation to `Z`, and the subsequent creation of a new instantiation again influences the entire calling clause:

```
path(a,d) :- branch(a,c),path(c,d).
```

Instead of the condition `path(b,d)` we therefore have to consider the condition `path(c,d)`. By means of successive matches with the clauses 6, 3 and 5, the interpreter derives the answer `yes` to the query `path(c,d)`. Both conditions to the match with the original query `path(a,d)` are now fulfilled. The interpreter therefore answers the original query in the affirmative. ◇

This example illustrates the modus operandi of the PROLOG interpreter, and, among other things, it was demonstrated that the PROLOG interpreter examines clauses in the order in which they have been specified in the database. According to the principles of logic programming, a logic program is viewed as a set of clauses; so, their respective order is of no consequence to the derived results. As can be seen from the previous example, however, the order in which clauses have been specified in the PROLOG database may be important. This is a substantial difference between a logic program and a PROLOG program: whereas logic programs are purely declarative in nature, PROLOG programs tend to be much more procedural. As a consequence, the programmer must bear in mind properties of the PROLOG interpreter when developing a PROLOG program. For example, when imposing some order on the clauses in the database, it is usually necessary that the clauses acting as a termination criterion for a recursive definition, or having some other special function, are specified before the clauses expressing the general rule.

4 Overview of the PROLOG language

Until now, all predicates discussed in the examples have been defined on purpose. However, every PROLOG system offers a number of predefined predicates, which the programmer may utilize in programs as desired. Such predicates are usually called *standard predicates* or *built-in predicates* to distinguish them from the predicates defined by the programmer.

In this section, we shall discuss several standard predicates and their use. Only frequently applied predicates will be dealt with here. A complete overview is usually included in the documentation concerning the particular PROLOG system. This discussion is based on SWI-PROLOG.

4.1 Reading in programs

By means of the predicate `consult` programs can be read from file and inserted into the PROLOG database. The predicate `consult` takes one argument which has to be instantiated to the name of a file before execution.

Example. The query

```
?- consult(file).
```

instructs the interpreter to read a PROLOG program from the file with the name `file`. ◇

It is also possible to insert into the database several programs from different files. This may be achieved by entering the following clause:

?- consult(file$_1$),...,consult(file$_n$).

PROLOG offers an abbreviation for such a clause; the required file names may be specified in a list:

?- [file$_1$,...,file$_n$].

4.2 Input and output

Printing text on the screen can be done by means of the predicate `write` which takes a single argument. Before execution of the procedure call `write(X)`, the variable `X` must be instantiated to the term to be printed.

Example. The clause

```
?- write(output).
```

prints the term `output` on the screen. Execution of the call

```
?- write('This is output.').
```

results in

```
This is output.
```

When the clause

```
?- create(Output),write(Output).
```

is executed, the value to which `Output` is instantiated by a call to some user-defined predicate `create` will be printed on the screen. If the variable `Output` is instantiated to a term containing uninstantiated variables, then (the internal representation of) the variables will be shown as part of the output. ◇

The predicate `nl` just prints a new line, causing output to start at the beginning of the next line. `nl` takes no arguments.

We also have some means for input. The predicate `read` reads terms entered from the keyboard. The predicate `read` takes only one argument. Before executing the call `read(X)`, the variable `X` has to be uninstantiated; after execution of the `read` predicate, `X` will be instantiated to the term that has been entered. A term entered from the keyboard has to end with a dot, followed by a carriage return.

4.3 Arithmetical predicates

PROLOG provides a number of arithmetical predicates. These predicates take as arguments arithmetical expressions; arithmetical expressions are constructed as in usual mathematical practice, that is, by means of infix operators, such as `+`, `-`, `*` and `/`, for addition, subtraction, multiplication, and division, respectively. Generally, before executing an arithmetical predicate all variables in the expressions in its left-hand and right-hand side have to be instantiated to terms only containing numbers and operators; the arguments will be evaluated before the test specified by means of the predicate is performed. For example, in a condition `X < Y` both `X` and `Y` have to be instantiated to terms which upon evaluation yield numeric constants, before the comparison is carried out. The following arithmetical relational predicates are the ones most frequently used:

```
X > Y.
X < Y.
X >= Y.
X =< Y.
X =:= Y.
X =\= Y.
```

The last two predicates express equality and inequality, respectively. Note that the earlier mentioned matching predicate `=` is not an arithmetical predicate; it is a more general predicate the use of which is not restricted to arithmetical expressions. Furthermore, the predicate `=` does not force evaluation of its arguments.

Besides the six arithmetical relational predicates shown above, we also have in PROLOG an infix predicate with the name `is`. Before executing

```
?- X is Y.
```

only the right-hand side `Y` has to be instantiated to an arithmetical expression. Note that the `is` predicate differs from `=:=` as well as from the matching predicate `=`; in case of `=:=` both `X` and `Y` have to be instantiated to arithmetical expressions, and in case of the matching predicate neither `X` nor `Y` has to be instantiated. If in the query shown above `X` is an uninstantiated variable, it will after execution of the query be instantiated to the value of `Y`. The values of both left-hand and right-hand side are subsequently examined upon equality; it is obvious that this test will always succeed. If, on the other hand, the variable `X` is instantiated to a number (or the left-hand side itself is a number), then the condition succeeds if the result of evaluating the right-hand side of `is` equals the left-hand side, and it fails otherwise. All other uses of the predicate `is` lead to a syntax error.

Example. Consider the following queries and answers which illustrate the differences and similarities between the predicates `=`, `=:=`, and `is`:

```
?- 3 = 2+1.
no

?- 3 is 2+1.
yes

?- 3 =:= 2+1.
yes

?- 3+1 = 3+1.
yes

?- 3+1 =:= 3+1.
yes

?- 3+1 is 3+1.
no

?- 1+3 = 3+1.
no

?- 1+3 =:= 3+1.
yes
```

The following examples illustrate the behaviour of these predicates in case the left-hand side is an uninstantiated variable. PROLOG returns by showing the computed instantiation:

```
?- X is 2+1.
X = 3

?- X = 2+1.
X = 2+1
```

We have left out the example `?- X =:= 2+1`, since it is not permitted to have an uninstantiated variable as an argument to `=:=`. ◇

The predicates `=:=` and `is` may only be applied to arithmetical arguments. The predicate `=` however, also applies to non-arithmetical arguments, as has been shown in Section 3.2.

Example. Execution of the query

```
?- X = [a,b].
```

leads to the instantiation of the variable `X` to the list `[a,b]`. In case the predicate `=:=` or the predicate `is` would have been used, the PROLOG interpreter would have signaled an error. ◇

4.4 Examining instantiations

A number of predicates is provided which can be used to examine a variable and its possible instantiation. The predicate `var` taking one argument, investigates whether or not its argument has been instantiated. The condition `var(X)` is fulfilled if `X` at the time of execution is uninstantiated; otherwise, the condition fails. The predicate `nonvar` has a complementary meaning.

By means of the predicate `atom`, also taking one argument, it can be checked whether the argument is instantiated to a symbolic constant. The predicate `atomic`, which also takes a single argument, investigates whether its argument is instantiated to a symbolic or numeric constant. The one-argument predicate `integer` tests if its argument is instantiated to an integer.

Example. Consider the following queries specifying the predicates mentioned above, and answers of the PROLOG interpreter:

```
?- atomic([a]).
no

?- atomic(3).
yes

?- atom(3).
no

?- atom(a).
yes

?- integer(a).
no
```

◇

4.5 Controlling backtracking

PROLOG offers the programmer a number of predicates for explicitly controlling the backtracking behaviour of the interpreter. Note that here PROLOG deviates from the logic programming idea.

The predicate `call` takes one argument, which before execution has to be instantiated to a procedure call; `call` takes care of its argument being handled like a procedure call by the PROLOG interpreter in the usual way. Note that the use of the `call` predicate allows for 'filling in' the program during run-time.

The predicate `true` takes no arguments; the condition `true` always succeeds. The predicate `fail` also has no arguments; the condition `fail` never succeeds. The general application of the predicate `fail` is to enforce backtracking, as shown in the following example.

Example. Consider the following clause:

```
a(X) :- b(X),fail.
```

When the query `a(X)` is entered, the PROLOG interpreter first tries to find a match for `b(X)`. Let us suppose that such a match is found, and that the variable `X` is instantiated to some term. Then, in the next step `fail`, as a consequence of its failure, enforces the interpreter to look for an alternative instantiation to `X`. If it succeeds in finding another instantiation for `X`, then again `fail` will be executed. This entire process is repeated until no further instantiations can be found. This way all possible instantiations for `X` will be found. Note that if no side-effects are employed to record the instantiations of `X` in some way, the successive instantiations leave no trace. It will be evident that in the end the query `a(X)` will be answered by `no`. ◇

The predicate `not` takes a procedure call as its argument. The condition `not(P)` succeeds if the procedure call to which `P` is instantiated fails, and vice versa. Contrary to what one would expect in case of the ordinary logical negation, PROLOG does not look for facts `not(P)` in the database (these are not even allowed in PROLOG). Instead, negation is handled by confirming failed procedure calls. This form of negation is known as *negation as (finite) failure*; for a more detailed discussion of this notion the reader is referred to [2].

The *cut*, denoted by `!`, is a predicate without any arguments. It is used as a condition which can be confirmed only once by the PROLOG interpreter: on backtracking it is not possible to confirm a cut for the second time. Moreover, the cut has a significant side effect on the remainder of the backtracking process: it enforces the interpreter to reject the clause containing the cut, and also to ignore all other alternatives for the procedure call which led to the execution of the particular clause.

Example. Consider the following clauses:

```
/* 1 */     a :- b,c,d.
/* 2 */     c :- p,q,!,r,s.
/* 3 */     c.
```

Suppose that upon executing the call `a`, the successive procedure calls `b`, `p`, `q`, the cut and `r` have succeeded (the cut by definition always succeeds on first encounter). Furthermore, assume that no match can be found for the procedure call `s`. Then as usual, the interpreter tries to find an alternative match for the procedure call `r`. For each alternative match for `r`, it again tries to find a match for condition `s`. If no alternatives for `r` can be found, or similarly if all alternative matches have been tried, the interpreter normally would try to find an alternative match for `q`. However, since we have specified a cut between the procedure calls `q` and `r`, the interpreter will not look for alternative matches for the procedure calls preceding `r` in the specific clause. In addition, the interpreter will not try any alternatives for the procedure call `c`; so, clause 3 is ignored. Its first action after encountering the cut during backtracking is to look for alternative matches for the condition preceding the call `c`, that is, for `b`. ◇

There are several circumstances in which specification of the cut is useful for efficiency or even necessary for correctness. In the first place, the cut may be used to indicate that the selected clause is the only one that can be applied to solve the (sub)problem at hand, that is, it may be used to indicate 'mutually exclusive' clauses.

Example. Suppose that the condition `b` in the following clause has been confirmed:

```
a :- b,c.
```

and that we know that this clause is the only one in the collection of clauses having `a` as a conclusion, which is applicable in the situation in which `b` has been confirmed. When the condition `c` cannot be confirmed, there is no reason to try any other clause concerning `a`: we already know that `a` will never succeed. This unnecessary searching can be prevented by specifying the cut following the critical condition:

```
a :- b,!,c.
```

$\diamond$

Furthermore, the cut is used to indicate that a particular procedure call may never lead to success if some condition has been fulfilled, that is, it is used to identify exceptional cases to a general rule. In this case, the cut is used in combination with the earlier mentioned predicate `fail`.

Example. Suppose that the conclusion `a` definitely may not be drawn if the condition `b` succeeds. In the clause

```
a :- b,!,fail.
```

we have used the cut in conjunction with `fail` to prevent the interpreter to look for alternative matches for `b`, or to try any other clause concerning `a`. $\diamond$

We have already remarked that the PROLOG programmer has to be familiar with the working of the PROLOG interpreter. Since the cut has a strong influence on the backtracking process, it should be applied with great care. The following example illustrates to what errors a careless use of the cut may lead.

Example. Consider the following three clauses, specifying the number of parents of a person; everybody has two of them, except Adam and Eve, who have none:

```
/* 1 */     number_of_parents(adam,0) :- !.
/* 2 */     number_of_parents(eve,0) :-  !.
/* 3 */     number_of_parents(X,2).
```

Now, the query

```
?- number_of_parents(eve,2).
```

is answered by the interpreter in the affirmative. Although this is somewhat unexpected, after due consideration the reader will be able to figure out why `yes` instead of `no` has been derived. $\diamond$

For convenience, we summarize the side-effects of the cut:

- If in a clause a cut has been specified, then we have normal backtracking over the conditions preceding the cut.
- As soon as the cut has been 'used', the interpreter has committed itself to the choice for that particular clause, and for everything done after calling that clause; the interpreter will not reconsider these choices.
- We have normal backtracking over the conditions following the cut.
- When on backtracking a cut is met, the interpreter 'remembers' its commitments, and traces back to the originating query containing the call which led to a match with the clause concerned.

We have seen that all procedure calls in a PROLOG clause will be executed successively, until backtracking emerges. The procedure calls, that is, the conditions are connected by commas, which have the declarative semantics of the logical $\wedge$. However, it is also allowed to specify a logical $\vee$ in a clause. This is done by a semicolon, `;`, indicating a choice between conditions. All conditions connected by `;` are evaluated from left to right until one is found that succeeds. The remaining conditions will then be ignored. The semicolin has higher precedence than the comma.

4.6 Manipulation of the database

Any PROLOG system offers the programmer means for modifying the content of the database during run-time. It is possible to add clauses to the database by means of the predicates `asserta` and `assertz`. Both predicates take one argument. If this argument has been instantiated to a term before the procedure call is executed, `asserta` adds its argument as a clause to the database before all (possibly) present clauses that specify the same functor in their conclusions. On the other hand, `assertz` adds its argument as a clause to the database just after all other clauses concerning the functor.

Example. Consider the PROLOG database containing the following clauses:

```
fact(a).
fact(b).
yet_another_fact(c).
and_another_fact(d).
```

We enter the following query to the system:

```
?- asserta(yet_another_fact(e)).
```

After execution of the query the database will have been modified as follows:

```
fact(a).
fact(b).
yet_another_fact(e).
yet_another_fact(c).
and_another_fact(d).
```

Execution of the procedure call

```
?- assertz(fact(f)).
```

modifies the contents of the database as follows:

```
fact(a).
fact(b).
fact(f).
yet_another_fact(e).
yet_another_fact(c).
and_another_fact(d).
```

◇

By means of the one-placed predicate `retract`, the first clause having both conclusion and conditions matching with the argument, is removed from the database.

4.7 Manipulation of terms

Terms are used in PROLOG much in the same way as records are in Pascal, and structures in C. In these languages, various operations are available to a programmer for the selection and modification of parts of these data structures. PROLOG provides similar facilities for manipulating terms. The predicates `arg`, `functor` and `=..` (pronounced as 'univ') define such operations.

The predicate `arg` can be applied for selecting a specific argument of a functor. It takes three arguments:

```
arg(I,T,A).
```

Before execution, the variable `I` has to be instantiated to an integer, and the variable `T` must be instantiated to a term. The interpreter will instantiate the variable `A` to the value of the `I`-th argument of the term `T`.

Example. The procedure call:

```
arg(2,employee(john,mccarthy),A)
```

leads to instantiation of the variable `A` to the value `mccarthy`. ◇

The predicate `functor` can be used for selecting the left-most functor in a given term. The predicate `functor` takes three arguments:

```
functor(T,F,N).
```

If the variable `T` is instantiated to a term, then the variable `F` will be instantiated to the functor of the term, and the variable `N` to the number of arguments of the functor.

Example. The procedure call

```
functor(employee(john,mccarthy),F,N).
```

leads to instantiation of the variable `F` to the constant `employee`. The variable `N` will be instantiated to the integer `2`. ◇

The predicate `functor` may also be applied in a 'reverse mode': it can be employed for constructing a term with a given functor `F` and a prespecified number of arguments `N`. All arguments of the constructed term will be variables.

The predicate `=..` also has a dual function. It may be applied for selecting information from a term, or for constructing a new term. If in the procedure call

```
X =.. L.
```

`X` has been instantiated to a term, then after execution the variable `L` will be instantiated to a list, the first element of which is the functor of `X`; the remaining elements are the successive arguments of the functor.

Example. Consider the following procedure call:

```
employee(john,mccarthy,[salary=10000]) =.. L.
```

This call leads to instantiation of the variable `L` to the list

```
[employee,john,mccarthy,[salary=10000]]
```

◇

The predicate `=..` may also be used to organize information into a term. This is achieved by instantiating the variable `L` to a list. Upon execution of the call `X =.. L`, the variable `X` will be instantiated to a term having a functor which is the first element from the list; the remaining elements of the list will be taken as the arguments of the functor.

Example. The procedure call

```
X =.. [employee,john,mccarthy,[salary=10000]].
```

leads to instantiation of the variable `X` to the term `employee(john,mccarthy,[salary=10000])`. ◇

Note that, contrary to the case of the predicate `functor`, in case of the predicate `=..` prespecified arguments may be inserted into the new term.

To conclude this section, we consider the predicate `clause`, which can be used for inspecting the contents of the database. The predicate `clause` takes two arguments:

```
clause(Head,Body).
```

The first argument, `Head`, must be sufficiently instantiated for the interpreter to be able to find a match with the conclusion of a clause; the second argument, `Body`, will then be instantiated to the conditions of the selected clause. If the selected clause is a fact, `Body` will be instantiated to `true`.

5 Suggested reading and available resources

Readers interested in the theoretical foundation of PROLOG and logic programming should consult Lloyd's *Foundations of Logic Programming* [2]. PROLOG is one of the few programming language with a simple formal semantics. This is mainly due to the declarative nature of the language. Students of computing *science* should know at least something of this semantics. A good starting point for the study of this semantics is knowledge of logical deduction in predicate logic [3].

An excellent introductory book to programming in PROLOG, with an emphasis on Artificial Intelligence applications, is [1].

The PROLOG community has its own Usenet newsgroup: `comp.lang.prolog`. There are quite a number of PROLOG programs in the public domain which researchers can use in their own research. SWI-PROLOG is a good complete PROLOG interpreter and compiler, which is freely available for Linux, Solaris and Windows at:

`http://www.swi.psy.uva.nl/projects/SWI-Prolog`

References

[1] I. Bratko. *PROLOG Programming for Artificial Intelligence*, 3rd ed. Addison-Wesley, Harlow, 2001.

[2] J. Lloyd. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, Berlin, 1987.

[3] P.J.F. Lucas and L.C. van der Gaag. *Principles of Expert Systems*, Addison-Wesley, Wokingham, 1991.