INTRODUCTION TO
LOGIC PROGRAMMING

Jacques MENU
CERN Academic Training 1985-86

Warning:

These course notes are the first version of a text this author is currently writing on symbolic logic and Prolog. Though the final text aims at being sound and complete, this one is certainly not. In particular, chapter 3 is only a first draft. Please report for any error or possible enhancement you may find.
BIBLIOGRAPHY

[Bundy 83] A. BUNDY
"The computer modelling of mathematical reasoning"
Academic Press, 1983

[Campbell 84] J.A. CAMPBELL, editor
"Implementations of Prolog"
Ellis Horwood Ltd, 1985

[Carbonell 78] J. CARBONELL
"Politics: automated ideological reasoning"
Cognitive Science 2, 1978

[Chang 73] C. CHANG & R. LEE
"Symbolic logic and mechanical theorem proving"
Academic Press, 1973

[Chenique 74] F. CHENIQUE
"Comprendre la logique moderne"
(4 volumes, french text)
Dunod, 1974

[Clocksin 81] W.F. CLOCKSIN & C.S. MELLISH
"Programming in Prolog"
Springer Verlag, 1981, 1984

[Duda 80] R. DUDA
"Model design in the Prospector consultant system for mineral exploration"
in "Expert Systems in the micro electronic age"
by D. MICHE, p. 153-167,
Edinburgh University Press, 1980

"Prolog"
InterEditions, Paris, 1985

[Hamilton 78] A.G. HAMILTON
"Logic for mathematicians"
Cambridge University Press, 1978
[Henderson 80] P. HENDERSON
"Functional programming"
Prentice-Hall, 1980

[Kleene 67] S. KLEENE
"Mathematical logic"
John Wiley and Sons, 1967

[Kowalski 79] R. KOWALSKI
"Logic for problem solving"
North Holland, 1979

[Manna 85] Z. MANNA & R. WALDINGER
"The logical basis for computer programming" (2 volumes)
Addison-Wesley, 1985

[Nilson 82] Nils J. NILSSON
"Principles of Artificial Intelligence"
Springer Verlag, 1982

[Robinson 79] J.A. ROBINSON
"Logic: form and function"
Edinburgh University Press, 1979

[Shoenfield 67] J. SHOENFIELD
"Mathematical logic"
Addison-Wesley, 1967

[Shortliffe 76] E.H. SHORTLIFFE
"Computer-based medical consultations: Mycin"
Elsevier Computer Science Library, 1976

[Winston 81, 84] P.H. WINSTON & B. HORN
"Lisp"
Addison-Wesley, 1981 (first edition)
Addison-Wesley, 1984 (second edition)

[Fargues 84] Jean FARGUES
"Moteurs d'inférences"
Séminaire sur les systèmes experts
Ecole des HEC, Université de Lausanne, Avril 1984

[Gasser 85] Christian GASSER
"Réalisation d'un compilateur Prolog"
Travail pratique de diplôme EPFL, 1985
[Gondran 84] Michel GONDRAIN
"Introduction aux systèmes experts"
(french text)
Eyrolles, 1984

[Iselin 81] François ISELIN
"Technique du bâtiment"
Publication du Département d'architecture EPFL, 1981

[Laurière 82] Jean-Louis LAURIERE
"Les systèmes experts"
Technique et Science Informatique (TSI) no 1 et 2, 1982

[Pabion 76] J.F. PABION
"Logique mathématique" (french text)
Hermann, Paris, 1967

[Trentinian 85] Laure de TRENTINIAN
"ACE: un système expert pour l'aide à la conception d'enveloppes"
Travail pratique de diplôme EPFL, 1985
1. SYMBOLIC LOGICS

Symbolic logic is the study of correct reasoning. It is concerned with the reasoning process as such, and ignores whether the particular hypotheses and conclusions are meaningful in actual life. This chapter is an overview of the most well-known classical and non-classical logics. Logics are useful because they provide a well-defined, non-ambiguous language to describe the world for the purpose of problem solving. Logic Programming is based on the fact that symbolic logic is an adequate language to represent knowledge and manipulate it.

We shall give an intuitive feeling of what logics are, without debating technical details. Since the notations and terminology vary greatly from one author to another, we shall avoid introducing too many technical terms. The important concepts will be presented in the context of the propositional logic for simplicity. The so-called first order predicate logic, that we shall write 'the FOPL' for short, will be treated in chapter 3.

1.1 Propositions and operators

The propositional logic studies the truth values of propositions (sentences) as a whole, independently of their contents. The only truth values are true and false. This very simple logic is the one devised by Aristotle.

As examples of propositions, one can consider:
- 'it is hot today';
- 'all red cats like beer'.

Note that whether or not a proposition makes sense in real life is irrelevant to the propositional logic. A proposition is just an indivisible entity whose value can either be true or false. Since the contents of propositions
is irrelevant — they are just any sentence — we may well explain the propositional logic with symbolic names as P or Q instead of particular sentences.

Propositions alone would be of little interest. In practice, one handles so-called well-formed formulae, formulae for short, that are compound propositions in which so-called operators may appear. The term 'well-formed' is related to the syntax of formulae. A logical formula in the propositional logic is either:

- a proposition;
- an operator followed by a formula. Such an operator is said to be monadic;
- an operator surrounded by two formulae. Such an operator is said to be dyadic;
- a formula surrounded by a pair of matching parentheses.

This can be illustrated by the syntax diagrams in figure 1.1. Syntax diagrams are now common practise in information science as a way to represent the structure of programs as well as data. In such diagrams, a rectangle indicates a concept defined by the same or another such diagram, such as 'operator', while non-rectangle boxes contain actual characters such as '('). The arrows show what paths are allowed from the entry gate to the exit gate of the diagram.

There are three patterns in a syntax diagram:

- sequence, in which one sub-pattern should be followed by another. This is the case of 'formula', 'operator' and 'formula' in the branch marked with '1' in figure 1.1;
- choice, in which parallel branches contain alternative sub-patterns. This is the case of the whole diagram in figure 1.1;
- loops, that will be met only in chapter 2.
The truth tables of the four monadic operators are the second to fifth columns in the table shown in figure 1.2:

We shall call the propositions occurring in a formula its constituents. Parentheses can be added freely to improve readability. This definition of formulae makes use of itself, and is therefore said to be recursive; the reader can refer to chapter 7 for a discussion of recursion. In the following F, G and H will represent formulae, while P and Q will stand for constituent propositions.

At the syntactical level, a logical operator combines one or two formulae, called the operand(s), to form a new formula. At the semantical level, operators return a value computed (evaluated) from the values of their operand(s). For example, given:

'it is hot today': true

'all red cats like beer': true

the formula:

'it is hot today' and 'all red cats like beer'

has the truth value 'true'. An operator can be defined by its truth table, showing the truth value returned for all the possible values of the operand(s).

The truth tables of the four monadic operators are the second to fifth columns in the table shown in figure 1.2:
or (disjunction): returns true if both operands are true, false otherwise.

otherwise:

returns true if both operands have the same truth value, false otherwise.

and (conjunction):

returns true if both operands are true, false otherwise;

or (disjunction):

Among them, only 'not' is actually useful, as will shall see in the remainder of this book. In its definition, 'false otherwise' can be understood as 'false if its operand is true', since the only two possible values for the operand are the truth values true and false.

For the reason discussed above, there are 16 dyadic operators. The most common of them are:

= (equality):

returns true if both operands have the same truth value, false otherwise;

and (conjunction):

returns true if both operands are true, false otherwise;

or (disjunction):


Figure 1.2: the monadic operators

To each one of the two possible values true and false for F may correspond one of the two values true and false in the definition of a monadic operator. There are thus 4 distinct ways to fill a vertical column in the table above, hence there are 4 monadic operators. These operators are:

id (identity):
returns the value of its operand, whatever it is:

not (negation):
returns true if its operand is false, false otherwise:

cf (constant false):
returns false, whatever the value of its operand.

cf (constant true):
returns true, whatever the value of its operand;
returns false if both operands are false, true otherwise;

\[ \text{xor (exclusive disjunction):} \]

returns true if exactly one operand is true, false otherwise. The term 'exclusive' stems from the fact that 'xor' is rather like 'or', except that the operands cannot both be true: they exclude each other for truth;

\[ \Rightarrow (\text{implication}): \]

to be detailed in paragraph 1.7.

Their truth tables are shown in figure 1.3.

\[
\begin{array}{cccccccc}
\hline
F & G & F = G & F \lor G & F \land G & F \Rightarrow G \\
\hline
true & true & true & true & true & true \\
true & false & false & true & true & false \\
false & true & false & true & true & false \\
false & false & true & true & true & true \\
\hline
\end{array}
\]

*Figure 1.3: Truth tables of some dyadic operators*

We can represent operators as arrows showing the possible values, also called the type, of the operands and result:

\[
\begin{align*}
\text{monadic operators:} \\
\langle \text{true}, \text{false} \rangle & \rightarrow \langle \text{true}, \text{false} \rangle \\
\text{dyadic operators:} \\
\langle \text{true}, \text{false} \rangle \times \langle \text{true}, \text{false} \rangle & \rightarrow \langle \text{true}, \text{false} \rangle.
\end{align*}
\]

The 'x' in the case of dyadic operators indicates that there are two operands, each one belonging to \( \langle \text{true}, \text{false} \rangle \). Note that this '\( \rightarrow \)' arrow has nothing to do with the '\( \Rightarrow \)' implication dyadic operator. It is simply a convenient way to represent the type of the operands and the type of the result. This will prove useful in paragraph 1.10.

1.2 Interpreting formulae
Given a formula $F$, in which some propositions $P$, $Q$, ... occur, interpreting $F$ is the fact of giving a truth value to each one of its constituent propositions $P$, $Q$, ... . Such an interpretation of $F$, also called a model of $F$, will thus yield a resulting value for the whole formula $F$, computed according to the operators that link these constituent propositions to make $F$. In other words, an interpretation of $F$ is simply a horizontal row in its truth table, the one corresponding to the values given to the constituent propositions.

A tautology is a formula that always evaluates to true, whatever the values of its constituents. Its truth table contains only true's, that is, it is a vertical column in which false does not appear. Symmetrically, a contradiction is a formula that always evaluates to false whatever the values of its constituents, thus its truth table does not contain true.

Tautologies and contradictions play a central role in logic, as will be seen later in this chapter and in chapter 3. We shall write:

\[ \vdash F \]

\[ \vdash \neg(G) \]

to indicate that formula $F$ is a tautology. The $\vdash$ sign is not a logical operator that might occur inside formulae. Rather, it is a symbol expressing something about a formula as a whole, namely that it is true under all interpretations. Thus:

\[ \vdash \neg(G) \]

means that $G$ is a contradiction: since $\neg(G)$ is true under all interpretations, $G$ is necessarily false under all interpretations, hence $G$ is a contradiction.

The most simple examples of a tautology is 'true', and the most simple example of a contradiction is 'false':

\[ \vdash \text{true} \]

\[ \vdash \neg(\text{false}). \]

Very simple examples can be built with the 'ct' and 'cf' monadic operators met above: 'ct $F$' returns true whatever $F$,
and \( \text{ct} F \) returns false whatever \( F \). Thus \( \text{ct} F \) is a tautology, and \( \text{cf} F \) is a contradiction:

\[
\begin{align*}
\text{ct}(F) & = False \\
\text{not(\text{ct}(F))} & = True
\end{align*}
\]

Other examples are:

\[
\begin{align*}
\text{not}(F \text{ and not}(F)) & = True \\
\text{F or not}(F) & = False
\end{align*}
\]

whatever the truth value of \( F \), \( \text{F and not}(F) \) is false;

whatever the truth value of \( F \), \( \text{F or not}(F) \) is true.

This can be seen in the truth tables shown in figure 1.4.

\[
\begin{array}{cccc}
F & \text{not} F & F \text{ and not}(F) & F \text{ or not}(F) \\
\hline
true & false & false & true \\
false & true & false & true \\
\end{array}
\]

Figure 1.4: a contradiction and a tautology

A very common, everyday life tautology is the one called modus ponens. It simply states that, whatever \( F \) and \( G \), it is true that:

\[
( F \text{ and } (F \Rightarrow G)) \Rightarrow G,
\]

that is:

\[
\begin{align*}
\text{not(} (F \text{ and } (F \Rightarrow G) \text{)} & = G. \\
\text{not(} (F \text{ and } (F \Rightarrow G) \text{)} & = G.
\end{align*}
\]

The reader is encouraged to build the truth table for the formula \( (F \text{ and } (F \Rightarrow G)) \Rightarrow G \), to get convinced that false does not occur in it. Modus ponens is one way to do inference, as will be shown in paragraph 1.7.

It is worth noting that tautologies are contentless, since they can be used in any interpretation. In particular, the names of the constituents propositions of a tautology are irrelevant, as illustrated in the paragraph 1.4. The concept of a tautologies is a fundamental one in logic, and we shall use it in the forthcoming paragraphs to rewrite logical formulae and to do logical inference.
Another important concept related to interpretations is that of consistency. A set of formulae is consistent if there exists an interpretation that makes true each formula in the set. Otherwise the set is said to be inconsistent. The key idea is that the conjunction of the formulae that compose an inconsistent set is necessarily false. This fact is used in some proof techniques such as reductio ad absurdum and Robinson's resolution principle, as we shall see in chapter 3.

1.3 Formulae equivalence

Two logical formulae $F$ and $G$ are said to be logically equivalent, equivalent for short, if they have the same truth table, that is:

- they share the same constituents;
- in any interpretation the values of $F$ and $G$ are the same.

The term 'equivalent' simply means 'having the same value'. This is a generalization to formulae of the '=' dyadic operator we met above, in order to take interpretations into account.

For example, the formulae '$F \Rightarrow G$' and 'not($F$) or $G$' have the same truth table, as can be seen in figure 1.5, and they are thus equivalent.

```
<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>not F</th>
<th>not(F) or G</th>
<th>F \Rightarrow G</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
```

*Figure 1.5: two equivalent formulae*
In everyday life, this equivalence can be illustrated with the following example. We can state that every cat has claws, namely:

'the animal is a cat' \( \Rightarrow \) 'the animal has claws'.

The equivalent form for '\( \Rightarrow \)' leads to the following:

\[ \text{not('the animal is a cat')} \text{ or 'the animal has claws'}. \]

In other words, if the animal is a cat the first operand of 'or' is false, thus the animal should have claws. If the animal is not a cat the first operand of 'or' above is true, hence the whole formula is true whether the animal has claws or not. The semantics of the '\( \Rightarrow \)' operator will be further discussed in paragraph 1.7.

As another example, the formulae '\( (F \Rightarrow G) \) and \( (G \Rightarrow F) \)' and '\( F = G \)' are equivalent. The reader is encouraged to build the corresponding truth tables as an exercise. Note that there is no problem with the '=' operator, although it is used to define logical equivalence: the formula '\( F = G \)' returns true or false according to the truth values of \( F \) and \( G \), and it happens that this returned value is always equal to the one returned by '\( (F \Rightarrow G) \) and \( (G \Rightarrow F) \)', independently of the interpretation.

Since formulae equivalence means equality under all interpretations, we can write:

'\( F \) is equivalent to \( G \)' as the tautology:

\[ \equiv \: F = G. \]

We shall call such tautologies 'equivalence tautologies'. The two examples of equivalent formulae above can be written:

\[ \equiv \: (F \Rightarrow G) = (\text{not}(F) \text{ or } G). \]

\[ \equiv \: (F = G) = ((F \Rightarrow G) \text{ and } (G \Rightarrow F)). \]

It is to be noted that in some interpretations, the formulae '\( F \)' and '\( G \)' and '\( F \) or \( G \)' are equal, such as the interpretation:

\( F = \text{false}, \: G = \text{false}, \).
in which both formulae are equal to false. But since there are other interpretations, such as:

\[ F = \text{false}, \ G = \text{true}, \]

in which these two formulae are not equal, they are not equivalent. Equality compares individual truth values, while equivalence compares whole formulae under all possible interpretations.

1.4 Formulae rewriting

The concept of formulae equivalence is very important due to the substitution theorem, that states that any formula can always be replaced by an equivalent formula in any context. This is obviously due to the fact that the resulting value will be the same in any interpretation. Based on this, Boolean algebra (after the logician Boole) is a calculus on logical formulae: it allows the mechanical transformation of a formula into an equivalent one, independently of the particular values of the formula constituent propositions.

Boolean algebra contains the concept of a variable, that can stand for any formula. Aristotle already allowed a single variable in his work on the propositional logic, and the reader may have noticed that we used such variable as \( F, G, P \) and \( Q \) in the paragraphs above. Boolean algebra also contains a set of rewrite rules that can be applied to any formula independently of any particular interpretation. These rules are simply equivalence tautologies as defined above.

Rewriting a formula using an equivalence tautology is done by instantiation and substitution. Instantiating a tautology means replacing each constituent proposition of the tautology by some other formula. Each occurrence of a given constituent in the formula should be replaced by the same 'thing'. Then an operand of the '=' operator in the instantiated tautology can be substituted for the other such operand in any context.

For example the formula '\( P \text{ nand} P \)' can be rewritten as '\( \text{not}(P) \)' using the following rewriting steps. First we instantiate the equivalence tautology:

\[ T1: \ \vdash \ (F \text{ nand} G) = \text{not}(F \text{ and} G). \]
with:
\[ F = P, \quad G = P, \]
leading to the instantiated tautology:
\[ T2: \quad ! = (P \text{ nand } F) = \neg(P \text{ and } F). \]
Then we instantiate the equivalence tautology:
\[ T3: \quad ! = F = (F \text{ and } F) \]
with the instantiation:
\[ F = P \]
giving:
\[ T4: \quad ! = P = (P \text{ and } F). \]
This allows us to substitute 'P' for 'F and P' in T2, leading to:
\[ T5: \quad ! = (P \text{ nand } P) = \neg(P). \]

Chaining such replacements obviously produces a formula equivalent to the one at the start of the chain, since the property of having the same truth table has been maintained along the chain. Hence we have derived the new equivalence tautology T5, showing how 'not' can be defined using the operator 'nand'. This result may then be used in further work at will.

Note that the step consisting of substituting 'P' for 'F' in T3 above is merely a renaming of the variables occurring in the tautology, and it can be short-cut. We use different names only to help the reader following the chain.

A Boolean algebra can be defined using only 'not', 'and' and 'or' by the 12 equivalence tautologies shown in figure 1.6.

Simplification of 'not':
\[ ! = F \text{ and } \neg(F) = \text{ false} \]
\[ ! = F \text{ or } \neg(F) = \text{ true} \]
Simplification of 'and' and 'or':
\[ ! = (F \text{ and } \text{true}) = F \]
\[ ! = (F \text{ and } \text{false}) = \text{false} \]
\[ ! = (F \text{ or } \text{true}) = \text{true} \]
\[ ! = (F \text{ or } \text{false}) = F \]
Commutativity of 'and' and 'or':
...
\[ \begin{align*}
&\begin{array}{c}
\text{(F and G)} = (G \text{ and } F) \\
\text{(F or G)} = (G \text{ or } F)
\end{array} \\
&\text{Associativity of 'and' and 'or':} \\
&\begin{array}{c}
\text{(F and (G and H))} = ((F \text{ and } G) \text{ and } H) \\
\text{(F or (G or H))} = ((F \text{ or } G) \text{ or } H)
\end{array} \\
&\text{Distributivity of 'and' and 'or':} \\
&\begin{array}{c}
\text{(F and (G or H))} = ((F \text{ and } G) \text{ or } (F \text{ and } H)) \\
\text{(F or (G and H))} = ((F \text{ or } G) \text{ and } (F \text{ or } H))
\end{array}
\]

Figure 1.6: defining a boolean algebra

These can be used to build new equivalence tautologies such as:

- De Morgan’s laws:
  \[ \begin{align*}
  &\begin{array}{c}
  \text{not}(\text{F and G}) = \text{not}(F) \text{ or } \text{not}(G) \\
  \text{not}(\text{F or G}) = \text{not}(F) \text{ and } \text{not}(G)
  \end{array}
  \]

- double negation:
  \[ \text{not}(\text{not}(F)) = F \]

- reversing implication:
  \[ \begin{array}{c}
  (F \Rightarrow G) = (\text{not}(G) \Rightarrow \text{not}(F))
  \end{array} \]

The instantiation chains for these is left as an exercise for the reader. Note that we may well substitute different things for various occurrences of a given formula; this will be considered as several successive rewritings.

1.5 Normal forms

A nice feature of the propositional logic is that any formula can be rewritten as a logically equivalent formula containing only a small number of basic operators. In fact some dyadic operators, such as 'nand' and 'nor', are powerful enough as to allow the rewriting of any formula into one containing only them. Equivalence tautologies for 'nand' and 'nor' are:

\[ \begin{align*}
&\begin{array}{c}
\text{(F nand G)} = \text{not}(F \text{ and } G) \\
\text{(F nor G)} = \text{not}(F \text{ or } G)
\end{array}
\]

Since only a few number of basic operators are sufficient to represent any logical formula, logicians have devised some rather simple forms, called normal forms, in which only these basic operators are used to build formulae.
Conversion from the original form into a normal form is done simply by successive rewritings.

One could use 'nand' or 'nor' to write these normal forms, but this would lead to formulae a human can hardly read. In practice, we shall use 'not', 'and' and 'or', that are closer to our perception of logic and are a good basis for automatic formula rewriting, as shown in chapter 3.

1.6 Logical consequence

The ultimate goal of logic is to study what Aristotle called 'correct arguments'. An argument is correct if it can be deduced from things supposed to be true by methods known to be correct. The concepts of formula equivalence and the substitution theorem we met above only allow us to manipulate formulae. They do not perform any reasoning, that is, they cannot help us derive new true facts from other facts.

The first approach to reasoning we shall consider is based on truth tables. A formula \( G \) is said to be the logical consequence — consequence for short — of another formula \( F \) if every interpretation that makes \( F \) true also makes \( G \) true. This will be noted:

\[
F \models G.
\]

More generally, a formula \( G \) is the logical consequence of a set of formulae \( F_1, \ldots, F_n \) if every interpretation that makes the \( F_i \) all true also make \( G \) true, leading to:

\[
(F_1 \text{ and } \ldots \text{ and } F_n \text{ and } G) = \text{true}
\]
in these particular interpretations. We shall write this:

\[
F_1, \ldots, F_n \models G.
\]

For example, consider the truth tables shown in figure 1.7.
In every interpretation such that: 'F and (F $\Rightarrow$ G)' is true, G is also true. In fact, the only such interpretation is the one in which both F and G are true. Hence 'G' is a consequence of 'F and (F $\Rightarrow$ G)', that is:

$$(F \land (F \Rightarrow G)) \models G.$$

This can also be considered as two formulae, 'F' and 'F $\Rightarrow$ G', whose consequence is 'G', namely:

$$F, (F \Rightarrow G) \models G.$$

In other words, if both 'F' and 'F $\Rightarrow$ G' are true, then 'G' is also true. This is where we reach the problem of reasoning: given a set of formulae, if they happen to be true in a particular interpretation, then any formula which is their consequence is also true in this interpretation.

Note that 'F $\models$ G' will not help us at all in interpretations in which 'F' has the value false. However this is no problem, since valuable reasoning can only be done on strong bases, that is, true formulae.

A tautology such as '$\models$ F' can be seen as a particular case of a consequence, in which no formula occurs at the left of the '!' sign. F is true in any interpretation in which 'nothing particular' is true, thus simply in any interpretation.

1.7 Logical inference

The second way to do reasoning we shall consider in this chapter is based on the use of modus ponens, that relies on
the '=>' operator. Remember that we have defined this tautology in paragraph 1.2 as:

\[ \vdash (F \text{ and } (F \Rightarrow G)) \Rightarrow G. \]

Modus ponens allows us to infer G from F and 'F => G'. 'To infer' is merely a synonym for 'to deduce'. We shall say that 'F' and 'F => G' are the the conditions of this inference, while 'G' is its conclusion.

The fact that:

from 'F' and 'F => G', we can infer 'G'

will be written:

\[ F, F \Rightarrow G \vdash G. \]

The "\(-\)" sign is comparable to "\(-\)": it is no logical operator, it expresses something about formulae. In everyday language, this is to be understood as:

if 'F' and 'F => G', then G.

More generally, a formula G can be inferred from a set of formulae F1, ..., Fn, noted:

\[ F1, ..., Fn \vdash G \]

if it is the last element of a sequence of formulae in which each formula can be:

- one of F1, ..., Fn;
- a tautology instance;
- a formula inferred, using modus ponens, from two other formulae occurring prior to it in the sequence.

A formulae sequence built this way is called an inference chain for G from F1, ..., Fn. A tautology instance occurring in an inference chain is usually the rewriting of a formula occurring prior to it in the chain. Infering a formula from two others using modus ponens is the basic inference step. The case in which there are no Fi's in the chain, that is, only tautologies instances, will be discussed in the next paragraph.

Examples of building an inference chain for a given formula will be shown in the next paragraphs. Let only mention here that the classical use of modus ponens illustrated at the beginning of this paragraph is clearly an
inference chain as we have just defined it. The inference chain for $G$ is:

- $F$: a formula;
- $F \Rightarrow G$: another formula;
- $G$: modus ponens applied on '$F$' and '$F \Rightarrow G$'.

Hence:

$$F, F \Rightarrow G \vdash G.$$  

Since our definition of inference relies heavily on the '$\Rightarrow$' operator, it is worth discussing its properties. Remember that we have defined it in paragraph 1.1 by the truth table shown in figure 1.8.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$P \Rightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true (1)</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false (2)</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true (3)</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true (4)</td>
</tr>
</tbody>
</table>

Figure 1.8: truth table of '$\Rightarrow$'

In everyday life, we understand 'it is true that $P \Rightarrow Q$' as meaning that if $P$ is true, then $Q$ is also true. This can also be expressed as:

$$\text{if } P \text{ then } Q.$$  

The first two interpretations of '$P \Rightarrow Q$' — the lines numbered 1 and 2 in the above truth table — correspond to this intuitive meaning. The only thing that might be discussed is the occurrence of 'true' in interpretations 3 and 4, indicated by an asterisk in the table. Refering to paragraph 1.1, this particular choice can be justified by the following considerations:

- if we had put 'false' twice instead, the truth table of '$\Rightarrow$' would have been the same as that of 'and';
- if we had put 'false' and 'true' in interpretations 3 and 4 respectively, the truth table of '$\Rightarrow$' would have the same as that of '=$$';
1.4

- if we had put 'true' and 'false' in interpretations 3 and 4 respectively, we would have lost the 'reversing implication' tautology we met in paragraph 1.4:

\[ !(F \Rightarrow Q) = (\neg Q \Rightarrow \neg F). \]

By analogy to equivalence tautologies, we shall say that modus ponens:

\[ !(F \text{ and } (F \Rightarrow G)) \Rightarrow G \]

is an inference tautology. Such a tautology is composed of an '=' operator, the first operand of which is a conjunction of formulae in general. In order to perform inference, we can instantiate modus ponens, with instantiation being used in the sense defined above, and state that from the first operand of '=' in the instantiated tautology we can infer the second such operand.

It is fundamental to note that inference is useful for reasoning only if the conditions of the inference are true formulae. In the case above, if \( F \) is false, \( !(F \text{ and } (F \Rightarrow G)) \Rightarrow G \) is true as always, but \( G \) may be either true or false, that is, we can conclude nothing about \( G \)'s truth value. Logic is only concerned with the form of the reasoning process; we may well infer false conclusions using correct inferences but false conditions.

1.8 Tautologies, theorems and proofs

We have defined logical inference in the previous paragraph using a sequence of formulae \( F_1, \ldots, F_n \). If no particular \( F_i \)'s occur in it, that is:

\[ !- G, \]

\( G \) can be inferred only using tautologies instances and modus ponens, and it is called a theorem. An inference chain for a theorem is called a proof of that theorem. Note that such a proof is not necessarily unique.

As an example, we can build an inference chain for the formula:

\[ (F \Rightarrow G) = (\neg G \Rightarrow \neg F) \]
by rewriting the following tautology we met in paragraph 1.6:

\[ T_1: \Rightarrow (P \Rightarrow Q) = (\neg P \lor Q). \]

First, the double negation tautology instance:

\[ T_2: \Rightarrow \neg \neg Q = Q. \]

allows us to substitute \(\neg \neg Q\) for \(Q\) in \(T_1\), leading to:

\[ T_3: \Rightarrow (P \Rightarrow Q) = (\neg P \lor \neg \neg Q). \]

Instantiating the symmetricity of 'or':

\[ T_4: \Rightarrow (X \lor Y) = (Y \lor X). \]

we can rewrite \(T_3\) as:

\[ T_6: \Rightarrow (P \Rightarrow Q) = (\neg P \lor \neg \neg Q). \]

We can now instantiate \(T_1\) again, this time with \(P = \neg \neg Q\) and \(Q = \neg \neg P\), giving:

\[ T_7: \Rightarrow (\neg \neg Q \Rightarrow \neg \neg P) = (\neg \neg P \lor \neg \neg Q). \]

This instantiation may seem intricate, but remember that \(T_1\) can first be rewritten using the renaming instantiation:

\[ P = A, Q = B \]

as:

\[ T_1': \Rightarrow (A \Rightarrow B) = (\neg A \lor B), \]

so that the previous instantiation now looks like \(A = \neg \neg Q\) and \(B = P\). Of course when replacing \(Q\) by \(\neg \neg (P)\), \(P\) should not be further replaced by \(\neg \neg (Q)\), and so on! In other words, substitution is only performed on one level.

The substitution of \(\neg \neg (Q) \Rightarrow \neg (P)\) for \(\neg \neg (Q)\) or \(\neg (P)\) in \(T_6\) leads to:

\[ T_8: \Rightarrow (P \Rightarrow Q) = (\neg Q \Rightarrow \neg P), \]

which is the tautology we wanted to establish.

The formula sequence:

\[ 'T_1, T_3, T_6, T_8' \]
contains only tautologies instances, not even a use of modus
ponens. According to our definition of proof above, this
sequence is thus a proof of theorem TB, and we can write:

\[ \vdash (F \Rightarrow G) = (\neg G \Rightarrow \neg F). \]

As the reader may remember, TB has been shown to be a
tautology in paragraph 1.4, where we called it 'reversing
implication':

\[ \models (F \Rightarrow G) = (\neg G \Rightarrow \neg F). \]

This leads us to an important question: is there some
relation between the concept of a tautology and that of a
theorem? The answer is 'yes': 'tautology' and 'theorem' are
synonyms, and the following statements about a formula F
express exactly the same thing in the propositional logic as
well as in the FOPL:

\[ \models F : F \text{ is a tautology}; \]
\[ \vdash F : F \text{ is a theorem}. \]

In fact, it can even be shown that the concepts of logical
consequence and logical inference are one and the same thing.
The synonymy of tautology and theorem above is merely the
case in which there are no formulae at the left of the '\(\models\)'
or '\(\vdash\)' signs. The two following statements express the same
thing:

\[ \text{F1, ..., Fn } \models G \]
\[ \text{G is a consequence of F1, ..., Fn}; \]
\[ \text{F1, ..., Fn } \vdash G \]
\[ \text{G can be inferred from F1, ..., Fn}. \]

We have established the 'removing implication' theorem
using an inference chain in this paragraph, and it could be
established using truth tables and logical consequence, as
left to the reader in paragraph 1.2. The approach using truth
tables and the one using modus ponens lead to the same
results in the propositional logic, and we may use whichever
we prefer.

There are cases however in which one approach is clearly
more practical than the other. Inference is preferable in the
propositional logic when the truth table of a formula is very
large, or when we want to perform reasoning automatically. It
will be seen in chapter 3 that in the FOPL truth tables are
not usable in general, so that inference will be the truly basic way to do reasoning in this context.

Truth tables are easier to understand and to manipulate for beginners, and are usually used to establish the basic bricks of logic such as double negation and modus ponens. The more complex forms of reasonings may then be performed using inference. Note that we have proceeded exactly this way to introduce reasoning in this chapter.

Since logical consequence and logical inference are one and the same concept, and due to the above remarks, we shall concentrate in the remainder of this book on inference. Remember anyway that '=!=' and '=!:' can be freely exchanged in the propositional logic and in the FOFL. As stated by a very important result in logic, showing that G can be inferred from the Fi's, that is:

\[ F_1, \ldots, F_n \vdash G, \]

is exactly the same thing as showing one of the following:

\[ F_1, \ldots, F_{n-1} \vdash F_n \Rightarrow G; \]
\[ F_1 \vdash (F_2 \text{ and } \ldots \text{ and } F_n) \Rightarrow G; \]
\[ != \text{ not } (F_1 \text{ and } \ldots \text{ and } F_n \text{ and not}(G)); \]
\[ !\vdash (F_1 \text{ and } \ldots \text{ and } F_n \text{ and not}(G)) \Rightarrow \text{false}. \]

The last two forms express what is known as 'reductio ad absurdum': to prove that G can be inferred from the Fi's, show that the conjunction of the Fi's and not(G) is a contradiction or, equivalently, that a contradiction can be inferred from this conjunction. These results can be shown rather easily and are left to the reader as an exercise.

1.9 Inference chaining

We have seen in paragraph 1.7 that inference is done by building a chain of formulae using modus ponens as the elementary inference step. Let us consider for example reasoning with the following facts, that are well-known in zoology:

- if the animal has feathers, then the animal is a bird;
- if the animal lays eggs, then the animal is a bird;
- if the animal is a bird and the animal flies well, then the animal is an albatross.

These facts can be written in the propositional logic as the formulae:

A1: 'the animal has feathers' => 'the animal is a bird';
A2: 'the animal lays eggs' => 'the animal is a bird';
A3: 'the animal is a bird' and 'the animal flies well' => 'the animal is an albatross'.

Using inference, we can easily show that:

'the animal has feathers', 'the animal flies well'

|- 'the animal is an albatross',

that can be understood as:

if the animal has feathers and the animal flies well then the animal is an albatross.

The corresponding inference chain can be built as follows. At the beginning of the chain, we place the two conditions of the inference we want to establish:

F1: 'the animal has feathers'
F2: 'the animal flies well'

The fact A1 states that 'F1 => F3': we can thus place F3 next in the chain:

F3: 'the animal is a bird'.

The fact A3 states that 'F3 and F2 => F4'. Thus we place F4 in the chain:

F4: 'the animal is an albatross'.

This last formula is precisely the conclusion of the inference we wanted to establish, thus the formulae sequence
"F1, F2, F3, F4" establishes that F4 can be inferred from F1 and F2.

The way we have built the inference chain for F4 is natural and called forward chaining. Going forward means that given some formulae, one repeatedly uses modus ponens to infer new formulae that may be used to infer other formulae, and so on. This is the 'natural' way to combine intermediate results in order to produce new ones. Chaining simply means that the conclusion of some inference step may be a condition of a later step.

Inference can also be done in another way called backward chaining: given a formula G we would like to infer from a set of other formulae F1, ..., Fn, one wonders whether it may be the conclusion of some inference step: if so, the problem is to see whether the conditions of that step can be inferred from the Fi's, thus leading to a new problem to be solved using backward chaining.

Both chainings can be illustrated in the example of zoology by the diagram shown in figure 1.9, in which 'the animal' has been omitted in the formulae F1 to F4. The node marked with 'and' corresponds to:

A3: F3 and F2 => F4,

the node marked with 'or' corresponds to the fact that either A1 or A2 may be used to prove F3:

A1: F1 => F3
A2: F2 => F3,

and the nodes marked with '*' are the ones that are not the conclusion of any of the implications A1 to A3:
Reasoning can be viewed as following paths in this diagram, also called an and/or tree. Forward chaining walks from the 'is an albatross' node at the bottom of the tree towards the root of the tree. Backward chaining walks from the root of the tree to its leaves.

There is thus a strong analogy between inference chaining and the well-known top-down or bottom-up policies in computer science. When working bottom-up, one assembles new results from results already there, and so on: this corresponds to forward chaining. When working top-down, one relies on the ability to find partial results and then combine them: this corresponds to backward chaining. Examples of bottom-up and top-down behaviours will be detailed in chapter 5.

Of course, any kind of forward and backward mixture is conceivable. Forward chaining leads to an explosion of inferred formulae, that are produced in a rather un-ordered way: each newly derived formula may lead to new inference steps. Backward chaining is ordered in the sense that the successive formulae one tries to infer are produced in a directed way, from conclusion to premisses. One may also say that forward chaining is data-driven, since the production of inferred formulae is directed by the initial set of formulae,
while backward chaining is goal oriented, that is, every inference goal may lead to other inferences as intermediate goals.

1.10 Predicates

Our discussion of logic up to now has been restricted to the case of the propositional logic. This is a strong limitation because we cannot manipulate the 'inside' of propositions. For example, if we know that:

'the age of john is 19'

is true, we cannot extract from this the fact that there is a relation between 'john' and '19', and that the relation in question is the fact that 19 is the age of 'john'. Remember that a proposition is an indivisible sentence, that might as well be written as 'foo'.

The way to avoid this limitation is the use of predicates. A predicate establishes a relation, that is, a correspondance, between members of a set of individuals S and the truth values true and false.

For example, the set of individuals S may contain 'john', 'paul' and '19', that is:

\[ S = \{john, paul, 19\}. \]

We can define a predicate 'age' as establishing a relation between a pair of individuals in S and an element in \{true,false\}. According to the notation we introduced in paragraph 1.1, 'age' can be represented by the diagram shown in figure 1.10.

\[
\begin{array}{c}
\text{age} \\
(S \times S) \rightarrow \{\text{true,false}\} \\
\text{II}, \text{I2} \\
\text{age(II,I2)}
\end{array}
\]

Figure 1.10: describing the 'age' predicate

In this diagram, what occurs at the left of the '--->' arrow is put in relation with what occurs at the right of the arrow. This 'age' relation might be defined by the truth table shown in figure 1.11.

\[ \text{II : I2 : age(II,I2)} \]
<table>
<thead>
<tr>
<th>john : john : false</th>
</tr>
</thead>
<tbody>
<tr>
<td>john : paul : false</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>john : 19 : true</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>paul : john : false</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>paul : paul : false</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>paul : 19 : true</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>19 : john : false</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>19 : paul : false</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>19 : 19 : false</td>
</tr>
</tbody>
</table>

Figure 1.11: a truth table for 'age'

What has been stated in this table is simply that the age of both 'john' and 'paul' is '19'.

The 'human' predicate described in figure 1.12 can be understood as: the individual X in S is a human if human(X)=true, he is no human otherwise.

\[
\text{human} \\
\text{S} \rightarrow \{\text{true}, \text{false}\} \\
\text{X} \rightarrow \text{human}(X)
\]

Figure 1.12: the 'human' predicate

We shall say that an 'n'-place predicate is a predicate with 'n' arguments. In everyday language, we can consider a one-place predicate such as 'human' as expressing a property of individuals, and a two-place predicate such as 'age' as a relation between two individuals in the set S. In general we can always consider a predicate as a relation. For example, the four-place predicate 'person' allowing:

\[
\text{person}(\text{john}, 1900, \text{male}, 180)
\]

actually establishes a relation between 'john', the name of a person, '1900', the year of his birth, 'male', his sex, and '180', his height in centimeters.
An important thing is that such a 'n'-place predicate, in which 'n' is greater than two, can always be rewritten using 'n-1' two-place predicates. In the case of 'person' above, the same information can be described in the following way:

\[
\begin{align*}
\text{person\_birth\_year} & \ (\text{john,1900}) \\
\text{person\_sex} & \ (\text{john,male}) \\
\text{person\_height} & \ (\text{john,180}) \\
\end{align*}
\]

### 1.11 Predicates Logics

The predicates logics are based on the predicate (or relation) concept we have met above. Their great expressive power stems from the fact that they also feature the so-called variables, that stand for something unspecified. How is this 'something' to be understood depends on how the variable is quantified.

A universally quantified variable can be replaced by any individual of S without changing the truth value of the formula in which it appears, provided every occurrence of the variable is replaced by the same element of S. A universally quantified variable is introduced by the 'whatever' quantifier, as is the case for 'X' in the following example:

\[
(\text{whatever } X, \ \text{human}(X) \Rightarrow \text{living\_being}(X)).
\]

An existentially quantified variable stands for a particular individual of S, uniquely identified by this means. It is introduced by the 'there exists' quantifier. For example, 'Y' is universally quantified in the sub-formula following the '⇒' operator in the following:

\[
(\text{whatever } X, \\
\text{human}(X) \Rightarrow \\
(\text{there exists } Y, \ \text{mother}(X,Y))
\]

If the arguments of a predicate can only be individuals of the set S, thus excluding other predicates as arguments, that predicate is said to be first order and belongs to the First Order Predicate Logic, FOPL for short. It is worth noting that the propositional logic and Boolean algebra are
included in the first order predicate logic: they correspond to the particular case in which there are no variables nor predicates arguments in formulae.

 Universally and existentially quantified variables are part of the classical, everyday logic. Replacing a variable by an individual of $S$ is called binding the variable to that individual. Due to the presence of variables and quantifiers, the concepts of logical consequence and logical inference are rather complex in the context of the FOL, and they will be studied in chapter 3.

 Now, consider the following statement:

>'red is a color'.

Here we have two 'levels' of properties, that is, of predicates: 'red' is a property of individuals, while 'color' is a property of other properties such as 'red'. Suppose that $S$ contains the individual 'table' that is both red and long, another property of individuals. If we write $B$ the set $\{\text{true, false}\}$ we are led to the corresponding diagrams:

- $S \rightarrow \{\text{true, false}\}$
  - $\text{red}$(table) = true
  
- $S \rightarrow \{\text{true, false}\}$
  - $\text{long}$(table) = true
- $(S \rightarrow B) \rightarrow \{\text{true, false}\}$
  - $\text{color}$(red) = true
  - $\text{color}$(long) = false

The unrestricted use of predicates may lead to the well-known antinomy first discovered by Peano and Russel. For example, given the 'ill' first order predicate:

- $S \rightarrow \{\text{true, false}\}$
  
- $\text{mary}$
  - $\text{ill}$(mary)

is it meaningful to write a formula like:

$\text{ill}$(ill) ?
Such a formula is intended to mean that the fact of being ill is itself ill. In fact, it does not make sense, because the argument of 'ill' in 'ill(ill)' is of type 'S -> {true, false}', not of type 'S'. In other words, 'ill' is a property of individuals, not of other properties! Problems such as the one above are well known as paradoxes, and they have long been a trouble for logicians. The way to avoid them is to stratify the predicate logics in orders, that is, in types.

A predicate without arguments belongs to the zeroth order predicate logic. It is merely a proposition and the zeroth order predicate logic is just the propositional logic. A proposition will be said to be of type:

\[ S \rightarrow B, \]

where \( B \) stands for the set of truth values \{true, false\}.

A predicate whose arguments are individuals is a first order predicate, and its type will be noted:

\[ S \rightarrow B. \]

A predicate whose arguments are first order predicates is a second order predicate, noted:

\[ (S \rightarrow B) \rightarrow B, \]

and so on. In this context, the antinomies mentioned above are easily detected as syntax errors, that is, as the fact that these tentative formulae are not well-formed. A simple way to recognize an \( n \)-th order predicate is that \( 'n' \) is the number of parentheses that can surround an individual of \( S \) in a formula involving this predicate.

As an example, given \( S \) as the set of human beings, we may define the predicates 'true', 'nice', 'friend' and 'transitive' as stated in figure 1.13.
declare a variable of type 'procedure', and assign it various OCR Output

It is worth noting that some programming languages contain a concept that is the counterpart of second order predicate 'transitive'. The arguments of P can in turn be variables — X, Y and Z in our example — standing for individuals in S. In spoken language, the fact that 'friend' relation is transitive might be defined as:

\[
\neg \text{transitive}(P) = \\
(\text{whatever X,} \\
(\text{whatever Y,} \\
(\text{whatever Z,} \\
(P(X,Y) \text{ and } P(Y,Z)) \Rightarrow P(X,Z)) \\
)
)
\]

Note that the unknown first order predicate P is nothing more than a variable that is used as argument of the second order predicate 'transitive'. The arguments of P can in turn be variables — X, Y and Z in our example — standing for individuals in S. In spoken language, the fact that 'friend' is transitive can be stated as:

the friends of my friends are my friends.

It is worth noting that some programming languages contain a concept that is the counterpart of second order predicates. In Algol-68 and Modula2, for example, one can declare a variable of type 'procedure', and assign it various

<table>
<thead>
<tr>
<th>predicate</th>
<th>type</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>nice</td>
<td>S \rightarrow B</td>
<td>1</td>
</tr>
<tr>
<td>friend</td>
<td>(S \times S) \rightarrow B</td>
<td>1</td>
</tr>
<tr>
<td>transitive</td>
<td>((S \times S) \rightarrow B) \rightarrow B</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 1.13: some various orders predicates
'procedure constants' during its life-time. Such a constant is simply the header and body of a procedure.

It is also possible for a predicate to have as arguments other predicates of various orders. For example, the predicate 'arity' may be used to specify a relation between a first order predicate and an integer:

\[
\text{arity}((S \times S) \rightarrow B) \times S \rightarrow B
\]

The following formulae are well-formed:

\[
\text{arity} (\text{true}, 0) \\
\text{arity} (\text{nice}, 1) \\
\text{arity} (\text{friend}, 2) \\
\text{arity} (\text{transitive}, 1).
\]

In fact, 'arity' is an omega order predicate. The omega order predicate logic is the fusion of all the predicate orders. Any predicate can be the first argument of 'arity', including 'arity' itself, and:

\[
\text{arity} (\text{arity}, 2)
\]

is a well-formed formula that can be naturally understood has having the value true.

We shall see in the remainder of this book that although Prolog is based on the FDPL, it is rather an implementation of the omega order predicate logic, augmented with completely extra-logical things such as input-output.

1.12 Consistency and decidability

Two important concepts in logic are those of consistency and decidability. Both of them are related to theorem proving, that is, to the concept of tautology.

A set formulae \( F_1, \ldots, F_n \) is said to be inconsistent if there exists a formula \( G \) such that we can infer both \( 'G' \) and \( 'not(G)' \) from the \( F_i \)'s, that is:

\[
F_1, \ldots, F_n \models G; \\
F_1, \ldots, F_n \models \text{not}(G).
\]

If the \( F_i \)'s are not inconsistent, they are said to be consistent. The name 'inconsistent' stems from the fact that
if we can infer a contradiction such as the one above from the Fi's, then any formula H can be inferred from them. This can be shown in a straightforward way using the inference tautology:

\[ \text{H} \equiv \text{G} \Rightarrow (\neg \text{G} \Rightarrow \text{H}), \]

that can easily be established using truth tables. We can build an inference chain for G in the following way. The first formulae in the chain are:

(1) \( F_1 \)
\[ \ldots \]
(\( n \)) \( F_n \)
(\( n+1 \)) \( G \)
(\( n+2 \)) \( \neg \text{G} \).

Now we instantiate the inference tautology mentioned above with \( G = G \) and \( H = H \) and place the resulting instantiated tautology at the end of the chain, leading to:

(\( n+3 \)) \( \equiv \text{G} \Rightarrow (\neg \text{G} \Rightarrow \text{H}) \).

Modus ponens can now be used to infer the formula \( \neg \text{G} \Rightarrow \text{H} \), which we place at the end of the chain:

(\( n+4 \)) \( \neg \text{G} \Rightarrow \text{H} \).

Finally, using modus ponens with formulae (\( n+2 \)) and (\( n+4 \)) we can infer H:

(\( n+5 \)) \( H \).

The inference chain for H is now complete, thus H can be inferred from the Fi's, whatever it may be.

We have seen that tautologies — that is, theorems — play a central role in logic: they are the basis of mechanical formulae rewriting as well as the basis of inference, together with modus ponens. Since a contradiction is merely the negation of a tautology, contradictions have the same importance as tautologies. A question has long interested logicians: given a formula F, is it possible, in a finite amount of time, to determine whether one of 'F' and 'not(F)' is a tautology? Note that a finite amount of time, in this context, may well be millions of centuries. According to the fact that tautology and theorem are one and the same concept, an equivalent formulation is:
is it possible, in a finite amount of time, to prove that F is a theorem?

The answer in the case of the propositional logic is clearly yes: using the truth table of the formula, we can determine whether it is true in all interpretations, that is, whether the formula is a tautology. A logic such as the propositional logic, in which we can decide (determine) whether any formula is a tautology or not, is said to be decidable.

It has been shown independently by Church and Turing that FOPL is only semi-decidable: if a formula is a tautology, this can be proved in a finite amount of time, but if it is no tautology, we cannot guarantee to establish this fact in a finite amount of time. An automatic proof method such as those presented in chapter 3 may well run forever without telling us whether the given formula is a tautology or not. This partially negative result should not discourage us in our search for automatic proofs. After all, if everything were so simple, human beings would not need sophisticated reasoning capabilities.

The equivalent decision problem in computer science is: given a program and some input data, can we determine, using another program, whether or not the execution of the former program with this input data will terminate in a finite amount of time? The answer is no. We cannot write a program that would answer the previous question in a finite amount of time.

1.13 Non-classical logics

The logics we have discussed so far, namely the propositional logic and the predicates logics, have been studied for years, and even centuries in the case of the propositional logic. Other logics have been developed more recently as attempts to bypass some of their limitations.

We have seen that in the classical predicate logics a formula may be always true (a tautology), such as 'F or not(F)', always false (a contradiction), such as 'F and not(F)', or possibly true depending on the values of the
components of the formula, such as 'F and G'. One might then think of a characterization of a formula with one of the three so-called modes 'always true', 'always false' and 'possibly true'. Modal logics allow one to handle modal operators as parts of the logic itself. The usual universal and existential quantifiers give an idea of what modes are, but they are no operators in the predicates logics: they just 'quantify' variables.

The idea in multi-valued logics is to handle not only true and false as truth values, but a range of intermediate values. For example, Lukasiewicz used: 0, 1/2 and 1 with meanings 'false', 'possible' and 'true', respectively. Another approach is to use the integers from -100 to 100, say, as truth values, with -100 being false and 100 being true. Such so-called 'certainty factors' are a truth degree. The logical operators are defined in this context according to Bayes theory. This has been used by Shortliffe to implement Mycin, an automatic theorem proving system in the context of such a logic.

The intuitionist logic is based on the assumption that the excluded middle (two truth values only) principle and the double negation rule do not hold any longer. If the set of individuals $S$ is not finite, such a the set of natural numbers, this logic says that no human can guarantee that the formula $F$:

$$(\text{there exists } X, \text{predicate}(X))$$

is true. This is because trying every $X$ in $S$ in turn to find one, call it $X_0$, such that predicate($X_0$) is true may require an infinite amount of time. In the intuitionist logic, using the supposition that there is no such $X_0$ to derive a contradiction does not prove that $F$ is true: this merely proves that 'not (not $F$)' is true.

An even more ambitious approach is the fuzzy logic, as defined by Zadeh, that allows one to perform approximate reasoning with things like:

- john is very tall;
- john is rather tall;
- john is much taller than peter;
- the temperature is normal for the season.
Of course, reasonings such as the ones allowed by the modal or fuzzy logics seem closer to our own reasoning. Some problems may arise however:

- in the case of an 100—valued logic, the size of the truth tables of common formulae is enormous;

- certain tautologies one finds in the first order predicate logic are no longer tautologies in the multi—valued or fuzzy logics;

- some formulae one gets in the modal logics are somewhat artificial or can hardly be understood. For example, what is to be thought of the formula:

  \[ F \text{ is necessary and } G \text{ is probable} \Rightarrow H \text{ is not impossible?} \]

- the truth table of certain modal operators are incomplete: hence, they are not actually truth operators.

An important thing to know is that all the logics we have mentioned so far — excepted the propositional logic — are strictly equivalent: no one has more expressive power than the others. Thus the debate of which logic to choose may seem of secondary importance.

Another non—classical logic is the temporal logic, in which time plays an important role. For example, one can express that some property will not hold after a given time, or that it holds only before a given time. This kind of logic can be useful in the context of robotics.

Let us also mention the knowledge logic, that contains operators such as 'k', meaning 'we know'. In such a logic,

\[ F \Rightarrow k(F) \]

can be understood as: 'if F is true, then we know F', or conversely: 'if we do not know F, then F cannot be true, hence it is false'. This is a way to formalize the so—called closed world assumption that we shall meet in the presentation of Prolog in the next chapter.

1.14 Logic Programming
The key idea behind Logic Programming is the use of logic to express knowledge about a given domain of human activity and to perform reasoning in this domain. The execution of a logic program can be viewed as showing that a relation exists between the input data and the results. Another point of view is to say that solving a problem is merely proving that a formula is a theorem: the answer to the problem will be a by-product of the proof, as shown in chapter 3.

Due to the underlying logic, Logic Programming emphasises the declarative approach to problem solving. A logic program describes what we know about the problem, either as facts or reasoning rules. The implementation of the corresponding logic language is then in charge to solve the problem. This is in contrast with the usual approach with imperative — algorithmic — languages, in which we have to give the details of how the desired solution should be determined. Thus Logic Programming is more concerned with what we are looking for than with how it should be looked for.

Of course a logic programming language implementation will have to manage the way the problems are solved, but ideally this is the implementation's job, and the programmer need not be aware of it. This has the nice property of removing many machine dependencies from programming languages. Furthermore, the 'programmer' using a logic language will naturally concentrate on what the solution is, that is, on the analysis phase of the problem solving activity. As a result the specification of the problem will be, to a great extent, the program to solve the problem.

The situation described above is what Logic Programming should be. In fact, the logic languages currently available fail to meet these requirements, essentially due to pragmatic reasons. The most widely used such language is Prolog (PRogramming in LOGic), that is now coming to maturity. The remainder of this book will be devoted to:

- an introduction to Prolog;
- the illustration of some logic programming techniques using it;
- the implementation in logic of various interesting functionalities;
- the implementation of Prolog itself.

1.14 Notes and bibliography

When we are talking about logic, there are always two languages in use: one is logic itself, the object we are studying, and the other is the language we use to talk about logic. The latter is sometimes 'meta-language', that means 'the language about the language'. This may lead to some confusion when it is not clear at which level, language or meta-language, we are. We have tried to avoid such problems by using a completely informal meta-language.

In particular we have used sentences like 'a formula is a tautology if ...' to define concepts, while mathematicians would use 'if and only if'. We have omitted on purpose to say that 'F is equivalent to G' is often stated as 'F if and only if G' by mathematicians for the same reason.

The general approach we have used to present logic is inspired from that of [Kleene, 1967], that is also a good of the intermixion of language and meta-language. This author has also benefitted from [Chenique, 1974]. Another good reference is [Manna, 1985], with the original particularity that is introduces an 'if then else' triadic operator.

It was decided to use a Prolog-like syntax to attain as much coherency as possible throughout the book. In particular, the Edinburgh syntax allows the declaration of new operators by the user, so that a formula like:

( whatever X, human(X) => likes(X,logic) )

can be written exactly this way in applications.

We have used the equality operator '=' to compare the equality of two truth values, though a so-called equivalence operator, noted '=<>', is usually used instead. This is not to confuse the reader when we later present the concept of logical equivalence of formulae, and because this notion of
equality is actually the same as the one for comparing non-truth values.

By the way, terminology is by far the most complex problem when one studies symbolic logic. Tautologies are often called valid formulae, logical consequence is sometimes called tautological consequence, inference is often called deduction, and so on. Our choices were guided by simplicity and clarity, but we are conscious that they may be criticized.

1.15 Exercises

1) What is the contrary (the negation) of:
   a/ there is a winner in every race;
   b/ nobody is sick;
   c/ this table is not both read and long;
   d/ in every logic, all the contradictions can be shown to be equivalent to any tautology.

2) Is it true that the contrary of the sentence:
   'when I drink coffee, I cannot sleep'

   is the sentence:
   'when I sleep, I cannot drink coffee'?

3) Where is the mistake in the following reasoning:

   Socrates said that all the Greeks are liers, but he was a Greek himself, so that he lied when he said that all Greeks are liers, so it is false that all Greeks are liers, thus Socrates himself was not a lier since he was Greek, so it is true that all Greeks are liers because Socrates said it, so ...

4) Find out the error in the following story:

   In a restaurant, the menu is charged 30 francs. After eating there, a usual client gives exactly that amount of money to the waiter. Since the boss had a good day, he asks the waiter to give back 5 francs to the client.

   The latter takes this opportunity to keep 3 francs as a tip for him, and gives only 2 francs back to the
the client, who thus paid only 28 francs. But if we take into account the 3 francs that the waiter kept, we arrive to a total of 31 francs ...

5) Let us state that an adjective is "blurk" if it cannot be applied to itself. For example "long" is blurk but "short" is not, and "french" is blurk but "english" is not. The question is: is "blurk" blurk itself?

6) In a street there are 3 houses, each of which has a distinct color. Furthermore the men that live in these houses have 3 distinct professional activities. It is known that:

- the frenchman lives in the red house;
- the german plays music;
- the englishman lives in the house that lies between the two others;
- the red house neighbours the green one;
- the writer lives in the first house in the street.

What nationality has the writer? Who lives in the yellow house?
2. PROLOG: EXAMPLES AND TERMINOLOGY

The Prolog language allows the user to reason about a certain world. The individuals that this world contains are purely formal - they have no relationship with anything in the real world. They just exist in the mind of the programmer. The power of the language is due the fact that one can manipulate relations among the individuals in the world.

A Prolog program is composed of facts that are known to be true in this world, as well as rules that can be used to infer new facts from other ones already known. Remember that 'to infer' is a synonym of 'to deduce', as stated in chapter 1. A logical program is thus a description of the world we are interested in, and it can respond to queries about the world. A query succeeds if the information it asks for is a known fact or can be inferred from known facts, otherwise it is said to fail.

This chapter presents the declarative semantics of Prolog, that is, how Prolog specifications can be understood from a logical point of view. The examples for this introduction have been selected in the context of family relations for simplicity. The so-called procedural semantics of the language, that is, how things work, will be covered in chapter 4.

2.1 A symbolic language

Prolog is a symbolic language, meaning that it allows the user to manipulate symbols. Such symbols can represent individuals in some world as well as relations between such individuals. A symbol is defined by its occurrence in the program text. It can be an atom, a number or variable.
An atom is a sequence of contiguous characters composed of either:

- a lower-case letter, followed by any number of letters, digits or underscore (_) characters. Any other character ends the atom name;
- a quote (') character, followed by any printable characters and then another quote. If we need quotes inside the atom, they should be doubled.

This can be viewed on the syntax diagram in figure 2.1. Examples of atoms are:

father
day
'how are you'
'john''s car'.

```
normal_atom:
+----------+
>--------> normal_atom +--------->
|   +----------+                  |
|   v                     |
|   +----------+                  |
>+-------> quoted_atom +------->
+----------+

lower-case:
+----------+
>--------> lower-case +-------->
|   +----------+                  |
|   v                     |
|   +----------+                  |
>+------> letter +------>
|   +----------+                  |
|   v                     |
|   +----------+                  |
>+------> digit +------>
<p>|   +----------+                  |</p>
<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
</table>
>+------>( _) +------>
```
A variable is a sequence of characters starting with a capital letter or an underscore character (_), followed by letters, digits or underscore characters. Figure 2.3 contains the corresponding syntax diagram. The concept of a variable in Prolog is sometimes referred to as the logical variable. A particular variable is the anonymous variable, denoted by a
single underscore character. Its meaning will be seen below. Examples of variables are:

X
F2
-
My_age
z1
17m.

variable:

----------

Figure 2.3: variables syntax

Apart from symbols, Prolog programs contain special symbols such as:

// */ ( ) . :- , ; [ ] !

The atoms and variables in a program can be separated by blank spaces, end of lines and comments. A comment is a sequence of characters starting with /* and terminated by */. Comments cannot be nested, and the first */' after the initial '/' of a comment terminates that comment. Note that a comment may spread over several lines, and that '/****/' is a legal comment.

The symbols are merely names that the programmer gives to the individuals and relations he needs to use. Note that some symbols are pre-defined by the language: among them are
symbols to handle input-output, clause management, arithmetic and so on.

2.2 Expressing facts

Suppose the world we are to reason about is a family. The facts about this particular family can be specified in Prolog in the following way:

```prolog
/* a particular fact */
'john loves mary'.

/* facts for the 'mother' relation */
mother(john,mary).

/* facts for the 'father' relation */
father(mary,mark).
father(john,paul).
father(paul,peter).

/* facts for the 'brother' relation */
brother(paul,george).
brother(john,henry).
```

In this world, the individuals are john, mary, mark, paul, peter, george and henry, and the relations are mother, father and brother. The case of 'john loves mary' may seem particular: in fact, it is simply a relation without arguments, what we called a proposition in chapter 1. Note that we cannot use upper-case letters as the first character in the names of individuals or relations. The comments, that is, the characters enclosed between '/*' and '*/', are destined to the human reader, and Prolog simply ignores them.

The above facts will be understood by Prolog as meaning:
- it is true that john loves mary;
- it is true that the mother of john is mary,
and so on. This is typical of the so-called declarative approach to problem solving: we just specify what we know about the world, that is, facts known to be true. The whole information contained in this world can be pictured as in figure 2.4, showing the individuals in the word and, enclosed in parentheses, the relations between them. Since 'john loves
mary' has no arguments, it is connected to nothing in this
diagram.

```prolog
mark
| |
| |
(father) | (father) |
| |
| |
| (brother)

mary       paul  --------------  george
\ /      \ /          \ /  \
(mother) \ / (father) \ /  \
\ /      \ /          \
\ /      \ /          
john  --------------  henry
(brother)
```

Figure 2.4: the relations in our sample family

In practice, we first have to enter the Prolog system
available on the machine we are going to use. After this is
done by a suitable command, we get the prompt:

?-

indicating that Prolog is ready to accept commands from the
user. Every command, also called query, is issued by typing
it in response to the '?' prompt and should be followed by a
terminating dot ('.') and a carriage return (CR). If we forget
the dot or the carriage return, Prolog will kindly wait for
us to type them in. In this book we shall always show queries
together with the prompt and the dot for clarity, though the
prompt should not actually be typed by the user.

In order to introduce the facts about the family
relations above in the description of the world, we can
proceed in one of two ways:

- create a file with some text editor, call it
  'family.pro', containing them in text form and load
  the contents of this file into the Prolog system
  using the particular query:

  `- consult('family.pro').

- issue the query:
?- consult(user).

and then type the facts directly on the keyboard, followed by an end_of_file marker, usually obtained by 'CTRL z', that is, depressing the 'CTRL' and 'z' keys simultaneously.

After all these facts have been loaded, Prolog will issue the '?' prompt, indicating that other queries can be issued by the user.

It is to be noted that the meaning (also called the semantics) of the facts we state about the world is purely conventional. Moreover, Prolog will not perform any coherence check on them. For example, one might specify both:

\[
\text{father(john,paul).} \\
\text{father(paul,john).}
\]

meaning that John and Paul are the father of each other. Of course Prolog cannot know that this is biologically nonsense. By the way it does make sense to specify simultaneously that the brother of Paul is George and that the brother of George is Paul:

\[
\text{brother(paul,george).} \\
\text{brother(george,paul).}
\]

It is worth noting that the order of the individuals occurring in facts and separated by commas is left to the programmer: our convention is that in a 'father' specification, the son occurs first and the dad next. Simply, we will have to be consistent with this convention in the usage we make of these facts.

2.3 Asking simple questions

Assuming the world is described by the facts we met in paragraph 2.2, we may wonder whether Mary is the mother of John. The query:

?- mother(john,mary).

will cause the answer:

yes
to be printed on the terminal. This is simply because the fact that Mary is the mother of John, namely:

mother(john,mary).

is known to Prolog. This affirmative answer can be understood as:

according to what I know of the world, the answer to this query is 'yes',

and we shall say that this query has succeeded.

On the other hand, the query:

?- mother(mary,john).

will produce the answer:

no

because it has not been specified that John is the mother of Mary, namely, no 'mother(mary,john)' fact has been specified. This negative answer can be understood as:

according to what I know of the world, the answer to this query is 'no',

and we shall say that this query has failed.

We may also issue to Prolog the query:

?- 'john loves mary'.

in which case the answer will be:

yes.

This query has succeeded because we have specified that it is true that 'john loves mary' in the world we are reasoning about.

Now, what about the following queries?

?- mother(mary,ann).
?- uncle(john,mary).

Both will fail due to the 'negation as failure' policy of Prolog, that states that anything not known to be true is false. In particular:

- an individual (here Ann) that does not appear in any fact is not part of the world we are reasoning about, and no fact can be true about it;
- a query concerned with a relation (here uncle) about which no fact has been specified cannot produce an affirmative answer.

Thus the two previous queries will fail and the answer of the Prolog system will be:

no.

A query to the world description is said to be a goal that Prolog has to satisfy. If an answer is found, the goal (and thus the query) succeeds, otherwise it fails. The notions of success and failure are fundamental in Prolog: they come from the fact that Prolog is a theorem prover, and that an attempt to prove a theorem can succeed or fail. The query:

?- mother(john,mary).

can be understood as:

can it be proved that
mary is the mother of john?

and Prolog is asked to try to prove that fact. If it succeeds we will get the answer 'yes', otherwise the answer 'no'.

2.4 Using variables

If our world description could only be used to retrieve the facts we have put in it, Prolog would be of little interest. By the way, the same effect can be obtained by the query:

?- listing(brother).

that will produce the following output on the terminal:

brother(paul,george).
brother(john,henry).

yes.

This query has asked to Prolog to write every fact it knows about the 'brother' relation. This is often useful in practise to obtain what our description of the world contains at a given moment.

Now consider the problem of asking who the mother of john is. The way to do that is just to use a variable in the
query, instead of only atoms. A variable simply stands for something unknown. Thus, the query:

?- mother(john,Mummy).

succeeds, producing the answer:

Mummy = mary,

and then a message is issued, in the form:

--> Next solution (y/n)?

The answer 'Mummy = mary' means that if 'Mummy' stands for mary, then it is true that the mother of john is 'Mummy'. This illustrates the use of the logical variable: it is a symbolic name for something not yet known. If the query succeeds, the variable will stand for some individual accordingly. The fact of replacing a variable by a particular individual is called binding the variable.

We shall say that the success or the query, as a side product, has bound the variable 'Mummy' to 'mary'. In the case of a successful query containing variables, Prolog issues the corresponding bindings if any, instead of simply 'yes'. The term 'if any' is needed in the previous sentence because the success of a query will not necessarily bind all the variables in the query, as we shall see. Variable binding in Prolog is to some extent analogous to assignment in classical programming languages. The properties of the logical variable will be studied in detail in a later chapter.

The message from Prolog produced after the binding of the variable 'Mummy' means:

would you like me to try to find out another solution to this query?

The particular message may vary from one implementation of Prolog to another. This behaviour of Prolog illustrates what is called the non-deterministic philosophy of the language: if a query may give multiple answers, Prolog can try to find out all of them in turn on request from the user. For each such alternative answer the corresponding variable bindings, if any, will be output. Note that though non-determinism sounds somewhat negative, it is indeed part of the power of the language.
In the particular case of the query for the mother of John, there is clearly only one solution. If we type 'y' in response to the message proposing another solution, Prolog will answer 'no', meaning that the search for an alternative solution has failed. If we type 'n' instead in response to this message, we just get the '?' prompt again, and we can proceed with other queries.

Of course, if the query has no solution at all, as in:

?- father(suzan,Dad).
we simply get the answer:

no.
In this case, we asked Prolog who the father of Suzan is: since nobody in our world is known to be her father, no binding of the variable 'Dad' can be done, and we just get a failure for this query.

The case of multiple alternative solutions can be illustrated by the following query:

?- father(Son,Dad).

It asks to Prolog what all the respective fathers and sons are, according to the description of the world. We just use two variables in the query, one for the son and one for the father. If we always type 'y' to get alternative solutions, the output for this query will be:

Son = mary, Dad = mark
--- Next solution (y/n) y
Son = john, Dad = paul
--- Next solution (y/n) y
Son = paul, Dad = peter
--- Next solution (y/n) y
no.

The query above is to be understood as: 'does there exist some Son and Dad related by the fact that Dad is the father of Son?'. The first answer means:

yes, a particular solution is:
Son = mary and Dad = mark.
We shall from now on omit the '---> Next solution (y/n)' message that queries may produce and give only the successive alternative answers. Don't forget however that, in case a query succeeds, this message will be issued to the user after the bindings of the variables occurring in the query have been output.

Suppose now that we only want to know who the fathers in our world description are, that is, we don't care of the identity of their sons. This can be obtained by the following query, containing an anonymous variable as the first argument of 'father':

?- father(_,Dad).

This query is to be understood as:

does there exist some Dad who is the father of someone, whoever this someone is?

and it succeeds, producing the successive answers:

Dad = mark
Dad = paul
Dad = peter.

The use of an anonymous variable as the first argument of 'father' prevents us from getting any information about the son. Whatever is bound to an anonymous variable by the success of a query is just forgotten at once by Prolog. In other words, the anonymous variable is there merely to fill a hole.

It is important to note that any two occurrences of the anonymous variable are distinct from each other, as our next example shows:

?- father(_,_).

In this query, the use of two anonymous variables as the arguments of 'father' prevents us from getting any particular bindings. Thus, we get no information at all about who is the father of who in our world description. Simply, this query will succeed because there is at least one son-dad pair in the 'father' relation. It can be understood as:

does there exist two individuals, one of which is the father of the other?
and the answer is simply:

yes.

The choice of the names for the non-anonymous variables in a query is irrelevant. The mother of john can be equally well obtained by one of the following queries:

?- mother(john,Mother_of_john).
?- mother(john,Mother).
?- mother(john,X).
?- mother(john,The_adress_of_suzan).

The only noticeable difference will be that, in the answer, the resulting binding will mention the actual name of the variable that occurs in the given query.

2.5 More complex queries

As the reader may have noticed, the facts held in the world description implicitly state that peter is the grand_father of john. This can be seen easily in the diagram in figure 2.4. In order to reason about the grand_father relation, Prolog gives us the possibility to query not for a single goal, but for a conjunction of goals that are to be satisfied simultaneously. For example the query:

?- father(john,Dad), father(Dad,Grand_pa).

can be understood as:

does there exist some Dad and Grand_pa such that
Dad is the father of john
and
Grand_pa is the father of Dad?

This query will succeed, giving the only answer:

Dad = paul, Grand_pa = peter.

The bindings for 'Dad' and 'Grand_pa' give us the answer that the father of the father of paul is peter. The comma (',') separating the two 'father' goals in the query is the logical 'and' operator we met in chapter 1.

If we are to find whose grand_father peter is, we can use the query:
?- father(Son of the son, Son), father(Son, peter).
that succeeds once with the bindings:
   Son of the son = john, Son = paul.

The fact that mark is the grand_father of john can be obtained with the query:
?- mother(john, Mother), father(Mother, Grand_pa).
that succeeds once with the answer:
   Mother = mary, Grand_pa = mark.

In all of the conjunctions of queries above, the variable that is common to the two sub-goals establishes a logical link between these goals. In the query for the father of the mother of john, the variable 'Mother' stands for the one individual who is the mother of john and whose father is mark.

As has been said in the previous paragraph, any names can be chosen for the variables in a query. Note however that since each occurrence of a given variable stands for one and the same individual, the query:

?- father(Someone, Someone).

fails, due to the fact that no individual in the world is known to be his own father.

2.6 Using inference rules

In order to simplify queries such as the one about the grand_father of john, we can specify who the grand_father of someone is in full generality. A possible definition of the 'grand_father' relation is:

   grand_father(Someone, Grand_pa) :-
       father(Someone, Dad), father(Dad, Grand_pa).

This is to be understood as:
A definition as the one for 'grand_father' above is called a rule. As the reader will have noted, the Prolog formulation is more concise than the one in natural language! Facts and rules are the components of Prolog programs: they are called clauses. The reason for this name is to be found in logic, as stated in paragraph 2.9.

Another possibility for Grand_pa to be the grand_father of Someone is that he is the father of the mother of Someone. This can be written straightforwardly:

\[
\text{grand_father}(\text{Someone}, \text{Grand_pa}) :- \\
\text{mother}(\text{Someone}, \text{Mother}), \text{father}(\text{Mother}, \text{Grand_pa}).
\]

Determining whose grand_father peter is can then be done with:

\[
?- \text{grand_father}(\text{Son of the son}, \text{peter}).
\]

that succeeds with the answer:

\[
\text{Son of the son} = \text{john}.
\]

In the same way, obtaining the grand_father of john can be written:

\[
?- \text{grand_father}(\text{john}, \text{Grand_pa}).
\]

that succeeds with the successive answers:

\[
\text{Grand_pa} = \text{peter} \\
\text{Grand_pa} = \text{mark}.
\]

There we meet non-determinism again: 'john' has actually two grand_fathers, the father of his father and the father of his mother. They are simply proposed by Prolog as successive solutions to the query.
Note that the intermediate links between 'john' and its grandfathers are not shown in the answers to the query above. These links, 'paul' and 'mary', are represented by the variables Dad and Mother in the rules that define the 'grandfather' relation. Only the possible bindings of variables occurring in the query will be output by Prolog.

The part of the rule on the left of the ':—' sign is called the head of the rule. The part on the right of ':—' is called the body of the rule. It is worth noting that a variable occurring only in the body of a rule, such as Dad above, is existentially quantified ('there exists'), whereas a variable occurring at least in the head of a rule is universally quantified ('whatever'). A fact in the form:

\[ \text{father}(\text{john}, \text{paul}) \]

is simply a particular case of a rule, whose body is empty: the ':—' sign can then be omitted.

As another example of a rule, let us consider the 'brother' specification. In the current state of the world description, Prolog cannot deduce the fact:

\[ \text{brother}(\text{george}, \text{paul}) \]

from the known fact:

\[ \text{brother}(\text{paul}, \text{george}) \]

In order to specify the well-known symmetry of brotherhood, one can replace 'brother' by 'brother_2' and define a new relation 'brother', leading to the following specification:

/* facts for 'brother_2' */
brother_2(paul,george).
brother_2(john,henry).

/* rules for symmetric 'brother' */
brother(Someone,Brother) :-
    brother_2(Someone,Brother).
brother(Someone,Brother) :-
    brother_2(Brother,Someone).

The last two rules mean that:
whatever Someone and Brother,
Brother is the brother of Someone if
Brother is the brother.2 of Someone
or
Someone is the brother.2 of Brother.

Then the query:
?- brother(Brother.1,Brother.2)
succeeds, producing the four successive answers:
Brother.1 = paul, Brother.2 = george
Brother.1 = john, Brother.2 = henry
Brother.1 = george, Brother.2 = paul
Brother.1 = henry, Brother.2 = john.

Note that the technique of using auxiliary relations such as 'brother.2' here is very frequent. This is typical of the top-down way to specify a problem. It is necessary in this particular case for a technical reason called left recursion, to be discussed in chapter 5.

2.7 Clauses selection

When we issue a query to Prolog, each goal in it has to be satisfied for the query to succeed. How does Prolog know which clauses are to be used to satisfy a particular query? This is done by selecting candidate clauses that might help solving the query. Since we have seen that a query can be a conjunction of elementary goals, the whole point for Prolog it to be able to satisfy an individual goal.

When trying to select a clause to satisfy a goal, only facts and the head of clauses are relevant. In order to simplify the discussion below, it is convenient to state that a Prolog fact is a particular case of a rule, in which the body is absent. In practise, a fact such as:

\texttt{father(john,paul)}.

can simply be thought of as the rule: 

\texttt{father(john,paul) :- true}.

where 'true' is a predefined argumentless relation that always succeeds. This allows us to talk of the body of the rule even if we talk of a fact, and the head of such a rule
if simply the fact itself. As should be emphasized, the body of clauses is irrelevant in candidate clauses selection.

A clause is a candidate to satisfy a goal if its head can be 'unified' with the given goal. Trying to unify a goal and a clause head can be seen as answering the question:

is that clause applicable to solve the goal?

Note that unification itself may succeed or fail, depending on the goal and clause head to be unified.

In order for the unification of a goal and the head of a clause to succeed, they should involve the same relation, have the same number of arguments, and these arguments if any should be unifiable pair-wise. In the previous sentence, 'if any' is needed because argumentless facts may also be used to satisfy argumentless goals. The reader may have noticed that our specification of the unification algorithm is recursive.

For example, if we try to unify the goal 'it rains' with the fact 'it rains', noted:

? 'it rains' <-? 'it rains'

we get:

success

because both involve the same argumentless relation.

If we try to unify the goal 'father(john,paul)' with the fact 'father(paul,henry)'

? father(john,paul) <-? father(paul,henry)

we get:

failure

because though they involve the same relation and the same number of arguments, the latter cannot be unified pairwise.

More precisely, the success of the two unifications:

? john <-? paul

? paul <-? henry
is required in order for the unification of 'father(john,paul)' and 'father(paul,henry)' to succeed. Since they both fail, the former unification also fails.

Similarly, the unification:

\[ ? \text{brothers(john,paul)} \leftarrow \text{brothers(john,paul,henry)} \]

fails because the number of arguments is not the same in the two 'things' we try to unify.

The unification mechanism in Prolog is the combination of classical pattern matching and the logical variable, that may be bound in the process. The unification algorithm will be presented in full detail in chapter 3. As a last example, the unification of the goal 'father(Son,paul)' with the fact 'father(john,paul)'.

\[ ? \text{father(Son,paul)} \leftarrow \text{father(john,paul)} \]

succeeds with the binding:

Son = john.

It should be remembered that variables bindings, as a side effect of unification, is the way answers are given to a Prolog query. Furthermore, after a goal and a clause head have been successfully unified, they are just one and the same, that is, they can be exactly superposed. In the previous query, both are:

father(john,paul)

due to the binding of Son to paul.

2.8 Looking for solutions

If the unification of a goal succeeds with a fact the goal succeeds right away, since a fact expresses something known. If unification succeeds with the head of a rule, the initial goal is replaced by the body of the rule. In this context, one can see Prolog rules as rewriting rules: the goal at hand is rewritten into the body of a clause selected by unification between that goal and the head of the clause.
The initial goal succeeds if the body of the rule succeeds, otherwise it fails.

For example, given the rule:

\[
\text{grand_father}(\text{Someone}, \text{Grand_pa}) :\neg \\
\text{mother}(\text{Someone}, \text{Mother}), \text{father}(\text{Mother}, \text{Grand_pa}).
\]

the query:

\[
?- \text{grand_father}(\text{john}, \text{Unknown}).
\]

can be unified with the head of the above rule, with the bindings:

\[
\text{Someone} = \text{john}, \text{Unknown} = \text{Grand_pa}.
\]

That query is thus rewritten as the two goals:

\[
\text{mother}(\text{john}, \text{Mother}) \\
\text{father}(\text{Mother}, \text{Unknown})
\]

that have to be satisfied by Prolog in order for the initial query to succeed.

We have seen that Prolog attempts to find all the answers to a query upon request, a philosophy called non-determinism. This is achieved simply by trying in turn each clause whose head can be unified with the given goal. Later on, if the chosen clause leads to a failure or if alternative solutions are sought, the choice of that clause will be reconsidered, a process called backtracking. This implies undoing all variables bindings done when that clause was chosen by unification.

The reader may have noticed that we can always predict the order in which the successive answers will be produced, though the logical semantics of the clauses in the program does not prescribe any particular order in which they should be used. This is simply because Prolog uses a well-determined strategy when trying to prove a goal. This leads to the nice property that a Prolog program has a unique semantics, whatever the particular implementation we are working with. Of course arithmetic will have to be excluded from the above statement as long as the implementations of Prolog will rely on machine dependant numerical computations semantics.

The strategy of Prolog states that:
- when trying to solve a conjunction of goals, these are considered in their textual order in the clause body or in the query, from left to right;
- when trying to solve a goal, the clauses are considered in their textual order, from top to bottom;
- when a failure occurs, Prolog backtracks to the last clause choice for which alternative, not yet tried clauses remain.

It should be noted that backtracking can occur only when unification of a goal with a head clause fails. Indeed, failure in Prolog is just unification failure. Backtracking is the fact of reconsidering the most recent choice of a particular fact or rule, in order to look for other solutions. When the user asks for other answers, Prolog reacts as if a failure had occurred, thus backtracking to the most recent choice point.

Let us illustrate all these points by an example. Consider the facts:

\[
\begin{align*}
\text{father}(\text{john}, \text{paul}). \\
\text{father}(\text{paul}, \text{henry}). \\
\text{father}(\text{mark}, \text{peter}).
\end{align*}
\]

The query:

\[
?\leftarrow \text{father}(\text{Son}, \text{Dad}).
\]

leads to the following scenario:

The goals at hand are:

\[
1 \Rightarrow \text{father}(\text{Son}, \text{Dad}): \text{still to be satisfied}...
\]

Trying to unify goal (1) with the head of the rule:

\[
\text{father}(\text{john}, \text{paul}) :- \text{true}.
\]

This succeeds with the bindings:

Son = john, Dad = paul

The goals at hand are:

\[
1 \Rightarrow \text{father}(\text{john}, \text{paul}): \text{satisfied}
\]

A solution is:

Son = john, Dad = paul

*** Next solution (n/_)? y

Unbinding variables Son and Dad
The goals at hand are:
   \[ 1 \implies \text{father}(\text{Son}, \text{Dad}): \text{still to be satisfied} \ldots \]

Trying to unify goal (1) with the head of the rule:
   \[ \text{father}(\text{paul}, \text{henry}) :- \text{true}. \]
This succeeds, with the bindings:
   \[ \text{Son} = \text{john}, \text{Dad} = \text{paul} \]
The goals at hand are:
   \[ 1 \implies \text{father}(\text{paul}, \text{henry}) : \text{satisfied} \]

A solution is:
   \[ \text{Son} = \text{paul}, \text{Dad} = \text{henry} \]
\[ \text{Next solution } (n/.) ? y \]
Unbinding variables \text{Son} and \text{Dad}

The goals at hand are:
   \[ 1 \implies \text{father}(\text{Son}, \text{Dad}): \text{still to be satisfied} \ldots \]

Trying to unify goal (1) with the head of the rule:
   \[ \text{father}(\text{mark}, \text{peter}) :- \text{true}. \]
This succeeds, with the bindings:
   \[ \text{Son} = \text{john}, \text{Dad} = \text{paul} \]
The goals at hand are:
   \[ 1 \implies \text{father}(\text{mark}, \text{peter}) : \text{satisfied} \]

A solution is:
   \[ \text{Son} = \text{mark}, \text{Dad} = \text{peter} \]
\[ \text{Next solution } (n/.) ? y \]
Unbinding variables \text{Son} and \text{Dad}

All the candidate clauses for goal (1) have been tried

This example illustrates how clause selection by unification binds variables in order to construct a solution, and how backtracking operates.

It is important to note the following properties of the logical variable:

- it may be bound as a side product of unifying a goal with the head of a clause;
- this binding can only be undone when a failure causes backtracking to take place. This occurs when unifying a goal with a clause head fails, or when the user asks for alternative solutions.
In other words, the logical variable behaves much like a read-only variable in usual programming languages: once it has been bound in the course of selecting a particular clause to satisfy a given goal, there is no means to bind it to something else, that is, to change its value. This is will only be done in the case another clause is selected to try to satisfy the goal in question.

2.9 More examples of goals satisfaction

We have seen in the previous paragraph how a simple query can be satisfied by Prolog. Let us now consider as a more complex example the following program:

```
goal_1(X1,Y1,X2,Y2) :-
    brother(X1,Y1), brother(X2,Y2),
    goal_1(_,_,_,_).
```

The query:

```
?- goal_1(X,Y,Z,T).
```

will produce the five successive answers:

- \( X = \text{paul}, Y = \text{george}, Z = \text{paul}, T = \text{george} \)
- \( X = \text{paul}, Y = \text{george}, Z = \text{john}, T = \text{henry} \)
- \( X = \text{john}, Y = \text{henry}, Z = \text{paul}, T = \text{george} \)
- \( X = \text{john}, Y = \text{henry}, Z = \text{john}, T = \text{henry} \)

*** That's all ***

\( X = X, \quad Y = Y, \quad Z = Z, \quad T = T. \)

Note that 'print' and 'nl' (start new line) are extra-logical primitives that perform output from the program. In the last solution, no binding has occurred for either of \( X, Y, Z \) and \( T \), which is denoted that they are simply equal to themselves. First, all the possible combinations for these four variables under the constraint:

```
brother(X,Y) and brother(Z,T)
```

have been exhausted, producing the first four solutions. Finally the second clause for 'goal_1' has been tried, due to the choice point at that place, leading to the printed message and the fifth solution.
The detail of what happens when Prolog tries to satisfy the query:

?- prove goal_1(X,Y,Z,T).

is as follows:

The proof tree is:
1 ==> goal_1(-46,-47,-48,-49): still to be proved...

Trying to solve goal (1) using the rule:

goal_1(-185,-186,-187,-188) :-
brother(-185,-186), brother(-187,-188).

The proof tree is:
1 ==> goal_1(-46,-47,-48,-49)
2 ==> brother(-46,-47): still to be proved...
3 ==> brother(-48,-49): still to be proved...

Trying to solve goal (2) using the rule:
brother(paul,george) :- true.

The proof tree is:
1 ==> goal_1(paul,george,-48,-49)
2 ==> brother(paul,george): proved
3 ==> brother(-48,-49): still to be proved...

The proof tree is:
1 ==> goal_1(paul,george,-48,-49)
2 ==> brother(paul,george): proved
3 ==> brother(-48,-49): still to be proved...

Trying to solve goal (3) using the rule:
brother(paul,george) :- true.

The proof tree is:
1 ==> goal_1(paul,george,paul,george)
2 ==> brother(paul,george): proved
3 ==> brother(paul,george): proved

### Next solution (n/_) ? y

The proof tree is:
1 ==> goal_1(paul,george,-48,-49)
2 ==> brother(paul,george): proved
3 ==> brother(-48,-49): still to be proved...

Trying to solve goal (3) using the rule:
brother(john,henry) :- true.
The proof tree is:
1 =>> goal_1(paul, george, john, henry)
2 =>> brother(paul, george): proved
3 =>> brother(john, henry): proved

### Next solution (n/_) ? y

There are no more clauses to prove goal (3)

The proof tree is:
1 =>> goal_1(john, henry, _48, _49)
2 =>> brother(john, henry): proved
3 =>> brother(_48, _49): still to be proved...

Trying to solve goal (2) using the rule:
brother(john, henry) :- true.

The proof tree is:
1 =>> goal_1(john, henry, _48, _49)
2 =>> brother(john, henry): proved
3 =>> brother(_48, _49): still to be proved...

The proof tree is:
1 =>> goal_1(john, henry, _48, _49)
2 =>> brother(john, henry): proved
3 =>> brother(_48, _49): still to be proved...

Trying to solve goal (3) using the rule:
brother(paul, george) :- true.

The proof tree is:
1 =>> goal_1(john, henry, paul, george)
2 =>> brother(john, henry): proved
3 =>> brother(paul, george): proved

### Next solution (n/_) ? y

The proof tree is:
1 =>> goal_1(john, henry, _48, _49)
2 =>> brother(john, henry): proved
3 =>> brother(_48, _49): still to be proved...

Trying to solve goal (3) using the rule:
brother(john, henry) :- true.

The proof tree is:
1 =>> goal_1(john, henry, john, henry)
2 =>> brother(john, henry): proved
3 =>> brother(john, henry): proved
2.9 Controlling non-determinism

There are cases, however, in which all the possible solutions to a given query are not desired. For example, if we need only know whether there exists at least one son-dad pair, we can issue the following query:

?- father(_, _), !.

This will succeed exactly once. What has happened is that, after 'father(_, _)’ has succeeded a first time, the special goal ‘!', called 'cut', has removed the choice point for 'father(_, _)’, thus preventing the other, not yet tried alternative facts for 'father' to be tried.

In other words, 'father(mary, mark)' has been used to get a first success of 'father(_, _)’, and '!' has said to Prolog: 'do not try to find other solutions for that goal'. The last two alternative facts, namely 'father(john, paul)' and 'father(paul, peter)', cannot be used to re-satisfy the goal 'father(_, _)’.

The effect of attempting to prove a '!' goal is to succeed and to remove all the choice points from the choice of the particular rule down to the '!''. A query to Prolog has to be considered as a headless clause in that context. When backtracking encounters a '!', that is, from right to left, the goal which caused the selection of the corresponding rule fails.
As an example, let us consider the program for 'goal_2', a variant of the program for 'goal_1' above in which a '!' has been introduced between the two 'brother' goals in the body of the first clause:

\[
goal_2(X_1, Y_1, X_2, Y_2) :-
\]
\[
\quad \text{brother}(X_1, Y_1), !, \text{brother}(X_2, Y_2),
\]
\[
goal_2(_, _, _, _) :-
\]
\[
\quad \text{print}('*** That''s all ***'), \text{nl}.
\]

The query:

\[
?- \text{goal}_2(X, Y, Z, T).
\]

will produce the two successive answers:

\[
X = \text{paul}, \ Y = \text{george}, \ Z = \text{paul}, \ T = \text{george}
\]
\[
X = \text{paul}, \ Y = \text{george}, \ Z = \text{john}, \ T = \text{henry}
\]

First, the first possible solution for 'brother(X,Y)' has been found, then '!' has removed the choice points for that goal and the 'goal_2' clause, and the partial solution:

\[
X = \text{paul}, \ Y = \text{george}
\]

has been combined with the two possible solutions for 'brother(Z,T)', producing the two solutions to the initial query. Then there remains no choice point at all, and no other solutions are sought, thus preventing the two solutions corresponding to:

\[
X = \text{john}, \ Y = \text{henry}
\]

to be found and the final message '*** That''s all ***' to be printed.

The details of what happens on the previous query is as follows:

The proof tree is:

\[
1 \Rightarrow \text{goal}_2(\_46, \_47, \_48, \_49) : \text{still to be proved...}
\]

Trying to solve goal (1) using the rule:

\[
goal_2(\_189, \_190, \_191, \_192) :-
\]
\[
\quad \text{brother}(\_189, \_190), !, \text{brother}(\_191, \_192).
\]
The proof tree is:
1 => goal_2(_46, _47, _48, _49)
  2 => brother(_46, _47): still to be proved...
  3 => !: still to be proved...
  4 => brother(_48, _49): still to be proved...

Trying to solve goal (2) using the rule:
  brother(paul, george) :- true.

The proof tree is:
1 => goal_2(paul, george, _48, _49)
  2 => brother(paul, george): proved
  3 => !: still to be proved...
  4 => brother(_48, _49): still to be proved...

Encountering '!!'

Removing choice point for goal (2)

Removing choice point for goal (1)

The proof tree is:
  * 1 => goal_2(paul, george, _48, _49)
  * 2 => brother(paul, george): proved
  * 3 => !: cut
  * 4 => brother(_48, _49): still to be proved...

Trying to solve goal (4) using the rule:
  brother(paul, george) :- true.

The proof tree is:
  * 1 => goal_2(paul, george, paul, george)
  * 2 => brother(paul, george): proved
  * 3 => !: cut
  * 4 => brother(paul, george): proved

### Next solution (n/_) ? y

The proof tree is:
  * 1 => goal_2(paul, george, _48, _49)
  * 2 => brother(paul, george): proved
  * 3 => !: cut
  * 4 => brother(_48, _49): still to be proved...

Trying to solve goal (4) using the rule:
  brother(john, henry) :- true.
The proof tree is:

1. \( \text{goal.
2. \( \text{brother(paul,george)}: \text{proved} 
3. \( \text{}!: \text{cut} 
4. \( \text{brother(john,henry)}: \text{proved} 

Next solution (n/.)? y

There are no more clauses to prove goal (4)

Backtracking on '!' (3)

There are no more clauses to prove goal (2)

There are no more clauses to prove goal (1)

2.10 Prolog and Logic

When introducing Prolog, we have met many concepts that are part of the first order predicate logic (FDPL). We shall only briefly show the correspondance between Prolog and the FOPL here. The full details are delayed until chapter 4.

Prolog clauses (facts and rules) are particular cases of clauses in the FOPL, namely Horn clauses, after the logician Horn. They are formulae known to be true in the world description. The relation in Prolog is the counterpart of the predicate in FOPL. By the way, we shall use from now on the word 'predicate' as a synonym for 'relation', and the word 'clause' as a synonym for 'fact or rule'.

The logical variable in Prolog covers the variable in the FOPL, either universally or existentially quantified. It stands for something not specified and can be bound to some value in order to satisfy a query.

A query has the form of a conjunction of goals, possibly a single goal. If the formula represented by the query cannot be proved to be true, the query fails. Otherwise it succeeds, and variables occurring in the query are output together with their binding, if any.

Some logical operators are also present in Prolog: the comma (',') in the body of a clause stands for 'and'. The negation as failure policy of Prolog implements a kind of negation, though this is not exactly the negation in the
FOFL. This is due to the fact that Horn clauses are only a subset of FOFL, as will be shown in chapter 4.

The ':-' sign is a right to left implication: the body of a rule, when true, implies the head of the rule. The 'grand_father' predicate above:

\[
\text{grand_father}(\text{Someone}, \text{Grand_pa}) :- \\
\text{father}(\text{Someone}, \text{Dad}), \text{father}(\text{Dad}, \text{Grand_pa}).
\]

can be written in the FOFL:

\[
(\text{whatever \text{Someone}}, \\
\text{whatever \text{Grand_pa}}, \\
(\text{there_exists \text{Dad}}, \\
(\text{father}(\text{Someone}, \text{Dad}) \\
\text{and} \\
\text{father}(\text{Dad}, \text{Grand_pa}) \\
) \\
) \\
\Rightarrow \\
\text{grand_father}(\text{Someone}, \text{Grand_pa})
).
\]

The case in which more than one clause defines a predicate, such as 'father' or 'brother' above, is interesting. When one specifies these facts, they are implicitly connected by 'and' operators. The clauses:

\[
\text{father}(\text{john}, \text{paul}). \\
\text{father}(\text{paul}, \text{peter}).
\]

state that:

the father of john is paul

and

the father of paul is peter.

When a query is issued to Prolog, these two clauses are connected by 'or': to prove that some formula involving the 'father' predicate, we can:

try the first clause

or

try the second clause.

This is typical of interrogation mode, that behaves rather like negating. This will be met again in chapter 4.
We can also use the 'or' operator in the body of a clause, written as semicolon (';'). For example, the two clauses for 'brother':

\[
\begin{align*}
\text{brother}(\text{Someone}, \text{Brother}) & : - \\
& \quad \text{brother}_2(\text{Someone}, \text{Brother}). \\
\text{brother}(\text{Someone}, \text{Brother}) & : - \\
& \quad \text{brother}_2(\text{Brother}, \text{Someone}).
\end{align*}
\]

can be rewritten as the single clause:

\[
\begin{align*}
\text{brother}(\text{Someone}, \text{Brother}) & : - \\
& \quad \text{brother}_2(\text{Someone}, \text{Brother}) ; \\
& \quad \text{brother}_2(\text{Brother}, \text{Someone}).
\end{align*}
\]

Though this ';' operator is useful, it should not be used too often since it tends to make programs obscure. There are cases however in which it cannot be replaced by a set of equivalent clauses, as will be seen in a later chapter.

Now, how is '!' related to logic? Removing choice points clearly has nothing to do with the declarative logical semantics of Prolog. '!' seems to be a pure control device used to guide the implementation, or we might say, to constrain it. In fact, '!' does have a logical effect: it transforms dynamically an inclusive 'or' into an exclusive 'or'. That means that if control reaches a '!' some possible solutions are excluded, and these will not even be looked for. The 'or' operator that implicitly holds between clauses thus becomes exclusive.

It is important to note two things about '!' :

- if control does not reach the '!' occurring in a clause body, either because unification could not succeed with the clause head or because a goal occurring at the left of the '!' in the clause body has failed, then no choice point will be removed at all by this '!'. This is why we said that the effect of '!' is dynamic;

- the dynamic effect of a '!' occurring in a clause body, if control ever reaches this '!', can be determined statically, that is, from the text of the program. This is a point in favour of a clean semantics of the language.
Though Prolog is clearly based on the FOPL, it is indeed much more than that. We shall see in the remainder of this book that Prolog can handle second order predicates as well as first order ones. In fact, any predicate order can be used in Prolog, which thus to some extent an omega order logical system, as defined in paragraph 1.12.

We shall present in detail in chapter 4 how things work, that is, how a Prolog implementation can try to satisfy queries. This is also known as the operational semantics of the language, and it will be placed in the more general framework of automated reasoning in the first order predicate logic.

2.11 Exercises

1) Give another specification of the 'grand_father' relation using a 'parent' relation, to be defined.

2) Define a predicate 'offspring' with two arguments. 'offspring(X,Y)' means that Y is an offspring of X, that is, X is an ancestor of Y.

3) Indicate which answers will be produced by the following queries:
   
   ?- brother(X,Y), brother(Z,T), !.
   ?- !, brother(X,Y), brother(Z,T).
   ?- goal_1(X,Y,Z,T), !.
   ?- goal_2(X,Y,Z,T), !.
3. AUTOMATED REASONING
IN THE FIRST ORDER PREDICATE LOGIC

The idea of automatic theorem proving seems to be as old as logic itself. In particular Leibnitz, in a letter dated 1679, stated what might be considered as the plan for research in Artificial Intelligence, including automatic theorem proving. We shall restrict ourselves to the context of the first order predicate logic, in which much effort has been done.

We have seen in chapter 1 that the propositional logic is decidable, that is, we can determine in a finite amount of time whether a given formula F is a theorem — a tautology — or not. This chapter covers the same subject in the context of the first order predicate logic.

3.1 Formulae in the FOPL

We have given in chapter 1 a flavour of what predicate logics are. Since we are now concerned only with first order predicate logic, we should first specify precisely what it is.

In the propositional logic, a formula is composed of indivisible propositions and logical operators. The FOPL is an extension of it in the sense that every formula in the propositional logic is also a formula in the FOPL. The features that are added as the extension are:

- the concept of a variable;
- quantifiers;
- predicates and functions.

Here are some sample formulae in the FOPL:
The precise definition of formulae in the FOPL first requires the definition of terms. Terms are the basic constituents of formulae, and are thus analogous to propositions in the propositional logic. A term in the FOPL can be defined as follows:

- an atom, such as 17 or 'john' and 'paul' above, is a term;
- a variable, such as 'Person' above, is a term;
- if 'function' is an atom and F1, ..., Fn are terms, then 'function(F1,...,Fn)' is a term. An example is 'collegues(john)' above;
- nothing else is a term.

It is worth noting that a function, in this context, is just a notation. 'collegues(john)' can be understood as 'the colleagues of john', but it does not indicate who these colleagues are. In other words, there is no function evaluation in a formula of the FOPL.

Terms are the building blocks for formulae. The way to build a formula out of terms is by using predicates, logical operators and quantified variables. The precise definition of formulae in the FOPL is:

- if 'predicate' is an atom and T1, ..., Tn are terms, then 'predicate(T1,...,Tn)' is a formula. The examples above contain the predicates 'beatle', 'musician' and 'perform_with'.
- if F is a formula, then 'not(F)' is a formula;
- if F and G are formulae, then 'F and G', 'F or G' and 'F => G' are formulae.
- if F is a formula and V is a variable, then
'(whatever V, F)' and '(there_exists V, F)' are
formulae;

- nothing can be a formula in the first order
predicate logic if it is not the result of a finite
number of applications of the four above criteria.

The reader may have noticed that there is no apparent
distinction between functions and predicates. Both are atoms
and can have terms as arguments. These are written between a
pair of parentheses and separated by commas. In fact
distinguishing between a function and a predicate can
actually be done easily, since predicates are the operands of
operators or are controlled by quantified variables.
Functions are simply embedded as predicates arguments.

We have seen that variables may be quantified. We shall
say that in:

(whatever V, Formula)

and

(there_exists V, Formula),

the 'Formula' is the scope of the variable 'V'. Each
occurrence of V in that formula is said to be quantified,
since it is in the scope introduced by a quantifier
controlling that variable. The use of parentheses around both
the quantified variable and the corresponding formula is a
help to visualize the scopes.

Note that a variable can occur in a formula without
being quantified. Such unquantified variables may pose some
problems, as we shall show in paragraph 3.4.

If the formula in the scope contains other scopes, the
latter are simply embedded in the former. When looking at a
formula in a scope, a variable is always controlled by the
innermost possible quantifier. For example, in:

(whatever X,
   (there_exists X, concert(X)
),

the X in 'concert(X)' is quantified existentially. The above
formula can be rewritten equivalently as:
This process is called renaming variable $X$ as $Y$. The two formulae are equivalent simply because a variable is merely a formal name. A given name is as good as any other, provided the scopes remain the same.

3.2 Formulae interpretation in the FOPL

In the propositional logic, an interpretation of a formula was the selection of a truth value for each basic proposition in the formula. In the FOPL, the case is more complex since there may be variables and functions involved.

The first concept to introduce is that of a domain. The domain $D$ of an interpretation is the set of individuals from which the variables take their values. The variables thus stand for individuals that are elements of the domain.

An $n$-place function '$f$' takes '$n$' individuals from the domain as arguments and returns another such individual as result. Such a function can be noted:

$$f: D \times D \times \ldots \times D \rightarrow D$$

$$I_1, \ldots, I_n \rightarrow f(I_1, \ldots, I_n)$$

An '$n$-place' predicate establishes a relation between '$n$' individuals from $D$ and a truth value, true or false. This can be noted:

$$p: D \times D \times \ldots \times D \rightarrow \{\text{true}, \text{false}\}$$

$$I_1, \ldots, I_n \rightarrow p(I_1, \ldots, I_n)$$

Since a functional term represents an individual from $D$, it can be used as a function argument as well as a predicate argument, as the definition of terms and formulae above allow.

This definition of the FOPL is in accordance with the concept of an order introduced in chapter 1. A variable
stands for an individual from $D$, and a predicate can have only such individuals as arguments.

An interpretation of the FOPL is composed of:
- a domain $D$ (the set of individuals);
- a collection of distinguished elements from $D$, that are the constants in formulae;
- a collection of functions to build terms;
- a collection of predicates to build formulae.

Such an interpretation is said to be over the domain $D$. Once this domain of interpretation is fixed, the truth value of a formula can be determined as follows:
- every constant is assigned an element of $D$;
- an individual from $D$ is assigned to unquantified variables;
- an individual of $D$ is chosen as the value of '$f(...)$' for each functional term;
- a truth value is chosen as the value of '$p(...)$' for each such predicative term;
- a formula in the form '(whatever $X$, Formula)' evaluates to true if 'Formula' is true for all possible individuals $X$ belonging to $D$;
- a formula in the form '(there_exists $X$, Formula)' evaluates to true if 'Formula' is true for at least one particular individual $X$ belonging to $D$;
- the truth value returned by the logical operators occurring in the formula are evaluated using the truth tables we met in chapter 1.

As can be seen from the above rules, unquantified variables are treated in interpretations exactly as constants. Let us see how things work on the following example. Let $F$ be the formula:

$$\text{musician}(X) \text{ or ( whatever } Y, \text{ musician}(Y)) \text{.}$$

In $F$, the variable $X$ is unquantified, while $Y$ is universally quantified, with scope '$\text{beatle}(Y)$'. Let the domain $D$ be the set containing 'john' and 'paul':
The unquantified variable X thus stands for either 'john' or 'paul'.

The case of predicates is treated as follows: 'musician' establishes a relation between an individual of D and a truth value. There are 4 ways to define such a relation, that we may call 'r1' to 'r4', as the following truth table shows:

\[
\begin{array}{c|c|c|c|c}
X & r1(X) & r2(X) & r3(X) & r4(X) \\
\hline
\text{john} & \text{true} & \text{true} & \text{false} & \text{false} \\
\text{paul} & \text{true} & \text{false} & \text{true} & \text{false} \\
\end{array}
\]

When interpreting '(whatever Y, \text{musician(Y)})', we must use the same 'musician' predicate as in 'musician(X)'. The truth table for F is thus as follows:

\[
\begin{array}{c|c|c|c|c}
\text{musician} & X & \text{musician}(X) & (\text{whatever Y}, & F \\
& & & \text{musician}(Y)) & \\
\hline
& \text{john} & \text{true} & \text{true} & \text{true} \\
& \text{paul} & \text{true} & \text{true} & \text{true} \\
\hline
\text{r1} & & & \text{true} & \text{true} \\
& & \text{false} & \text{false} & \text{false} \\
\hline
\text{r2} & & & \text{false} & \text{false} \\
& & \text{false} & \text{false} & \text{false} \\
\hline
\text{r3} & & & \text{false} & \text{false} \\
& & \text{true} & \text{true} & \text{true} \\
\hline
\text{r4} & & & \text{false} & \text{false} \\
& & \text{false} & \text{false} & \text{false} \\
\end{array}
\]

A formula such as '(whatever Y, \text{musician(Y)})' makes it necessary to build the truth tables of all the possible 'musician' predicates, here 'r1' to 'r4'. Building the truth table of a simple formula such as F above is thus more
Thus cannot be used to discuss the properties of formulae as in the propositional logic. In the case of \'(whatever Y, musician(Y))'\', one has to build the truth table of 'musician(Y)' to check whether it is true whatever Y.

Of course, if there are more than two individuals in D and we consider predicates with more than one place, the number of possibilities will increase drastically. The situation gets even worse if the domain D is not finite, such as the natural numbers.

There are infinitely many possible domains for interpretation. We might well have chosen:

\[ D = \{-3, \pi\} \]

instead of \[ D = \{\text{john, paul}\} \]. In fact, as the reader may have felt in the interpretation above, the actual domain is not relevant: only the number of individuals it contains is. By the way, the predicates names 'beatle' and 'musician' are not relevant either: we might have chosen 'foo' and 'blurk' as well. The actual choice is just a help to convey semantic information close to our world.

### 3.3 Tautologies, equivalence and consequence

The concepts of tautology, logical equivalence and logical consequence in the context of the FOPL are defined in the same way as in the propositional logic, that is:

- A formula F is a tautology if it is true under all interpretations. This is noted \( \models F \);

- Two formulae F and G are logically equivalent if they have the same value under all interpretations. This is noted \( \models F = G \);

- A formula G is the logical consequence of a set of formulae F₁, ..., Fn if G is true in any interpretation in which the F₁, ..., Fn are all true. This is noted \( \models F₁, ..., Fn \Rightarrow G \).

Since there are infinitely many domains over which to interpret formulae, there are infinitely many interpretations for any formula in the FOPL. The approach using truth tables thus cannot be used to discuss the properties of formulae as
we did in chapter 1 in the context of the propositional logic. This approach is important anyway, since it was used to build the first automatic theorem provers in the FOPL.

There are many important tautologies in the FOPL. The ones we shall use in this chapter to do inference and to rewrite formulae in normal form are:

\[ \neg (\forall x, P(x)) = (\exists x, \neg P(x)) \]
\[ \neg (\exists x, P(x)) = (\forall x, \neg P(x)) \]

The concept of tautology instantiation is more complex than in the propositional logic, due to the presence of variables. This is discussed in the next paragraph.

3.4 Renaming quantified variables

If variables and quantifiers are the reason why the FOPL is so powerful, they are also the reason why algebraic manipulations are more difficult than in the propositional logic.

Universally quantified variables are purely formal entities without any physical existence, and they do not even get values in interpretations. In the formula:

\[ (\forall x, \text{beetle}(x)) \]

the 'name' Person is a dummy name for every possible element of an interpretation domain D, but it does not represent any such element in particular. Any element of D can be put in the place of Person.

Existentially quantified variables denote one element of domain of interpretation, but nothing can help finding out which one in particular. In:

\[ (\exists x, \text{beetle}(x)) \]

the name Person is that of some element of D, about which we know nothing more than the fact that it is a 'beetle'.

Given a formula F, the particular name chosen for a quantified variable V occurring in F, possibly in a deeply embedded scope, can be replaced by any other provided the
semantics of the formula remains unchanged, that is, provided the formula obtained is equivalent to the original one. This requires that:

- the new name for \( V \) is not that of an unquantified variable occurring in the scope of \( V \);
- every occurrence of \( V \) in its scope should be replaced by the new name, but not those in embedded scopes controlled by a variable with the same name.

For example, the formula \( F \):

\[
(\text{whatever } Y, \text{performs_with}(Y,X))
\]

can be rewritten as the equivalent formula \( F_1 \):

\[
(\text{whatever } Z, \text{performs_with}(Z,X)),
\]

but not as the formula \( F_2 \):

\[
(\text{whatever } X, \text{performs_with}(X,X))
\]

that is not equivalent to \( F \) and \( F_1 \) above, since the replacement of \( Y \) by \( X \) has the undesirable effect of quantifying the second argument of 'performs_with' that was previously unquantified.

The reason why the latter replacement is not safe is that we want to be able to rewrite sub-formulae of a given formula as was the case in the propositional logic. The above original formula \( F \) may be embedded in another one, such as \( G \):

\[
(\text{there_exists } X, \text{performs_with}(Y,X))
\]

stating that there exists someone with whom everybody performs. The replacement of \( Y \) by \( X \) in \( F \) leads to \( G_1 \):

\[
(\text{there_exists } X, \text{performs_with}(X,X))
\]

that can be rewritten as the equivalent formula \( G_2 \):

\[
(\text{there_exists } X, \text{performs_with}(Z,Z))
\]

The latter \( G_2 \) means that there exists someone such that everyone performs performs with himself, which is quite
different from the original formula $G$ meant. It can be shown that this formula is not equivalent to the original one by exhibiting a particular interpretation over a particular domain in which the values of the two formulae are not the same.

In the second condition for the renaming of a quantified variable $V$ is necessary because, in an embedded scope controlled by the same variable name $V$, all occurrences of $V$ are quantified by the innermost quantifier. The only thing these two variable have in common is their name, hence renaming the outermost one should not affect any occurrence of the innermost one. For example, in the formula $H$:

```latex
(\text{there exists } X, \text{musician}(X) \text{ or} \\
(\text{whatever } Y, \\
(\text{whatever } X, \text{performs_with}(Y,X)) \\
) \\
),
```

renaming the outermost $X$ as $Z$ leads to $H1$:

```latex
(\text{there exists } Z, \text{musician}(Z) \text{ or} \\
(\text{whatever } Y, \\
(\text{whatever } X, \text{performs_with}(Y,X)) \\
) \\
).
```

In complex formulae, a good practise is to rename the innermost quantified variables first using brand new names, thus avoiding the possibility of a mistake. If in $H$ above we first rename the innermost $X$ as $T$, leading to $H2$:

```latex
(\text{there exists } X, \text{musician}(X) \text{ or} \\
(\text{whatever } Y, \\
(\text{whatever } T, \text{performs_with}(Y,T)) \\
) \\
).
```

it becomes obvious why the occurrence of $X$ in:

'\text{performs_with}(Y,X)'
should not be modified when we rename the outmost \( X \) in \( H \) above in order to obtain \( H_1 \). As a matter of fact, the best practise is simply to avoid synonyms in quantified variables names: we have usually enough possible names available.

The concept of variable renaming does not make sense for unquantified variables because they are actually constants in interpretations. Note however that the variable \( Y \) in:

\[
(\text{whatever } T, \text{ performs}_\text{with}(Y,T))
\]

is unquantified if we look at this formula alone, but if we embed the latter in some other formula, leading for example to \( H_2 \) above, then \( Y \) becomes quantified.

A formula in the FOPL is said to be closed if all the variables that occur in it are quantified. This is to avoid having to keep in mind that uncontrolled variables behave as constants.

3.5 Formulae rewriting

We have seen that the crucial point when rewriting formulae is to guarantee that the resulting formula is equivalent to the original one. Now that we know how to rename quantified variables, it is rather easy to define how a sub-formula \( G \) of a formula \( F \) can be replaced by a formula \( H \).

A formula \( F \) can be rewritten as another formula \( G \) if the following conditions hold:

- if \( F \) is in the scope of a quantified variable \( V \) that occurs unquantified in \( F \), then \( F \) cannot be rewritten as another formula;

- if \( F \) is in the scope of a quantified variable \( V \) that occurs unquantified in \( G \), then \( V \) should be renamed in the scope containing \( F \) before \( F \) is replaced by \( G \).

Note that if \( F \) is not a sub-formula of another formula, it cannot be in the scope of any quantified variable, thus rewriting can be done rather simply.
The first criterion above can be illustrated by the following example. Supposing that:

\[ \text{musician}(X) = \text{beatle}(X), \]

replacing the formula 'musician(X)' by 'beatle(X)' in:

\( (\text{whatever } X, \text{musician}(X)) \)

leads to:

\( (\text{whatever } X, \text{beatle}(X)) \)

which is not equivalent to the former formula. In fact, 'musician(X) = beatle(X)' simply means that for some individual X in the domain D, 'musician(X)' and 'beatle(X)' have the same truth value. But this is not necessarily the case for all the individuals in D. In this case, X is unquantified in 'musician(X)', but it is universally quantified in '(whatever X, musician(X))'.

The second criterion above for correct formula rewriting can be illustrated by the following example. Supposing that:

\[ \text{musician}(Y) = \text{beatle}(X) \]

holds, replacing 'musician(Y)' by 'beatle(X)' in the formula:

\( (\text{whatever } X, \text{musician}(Y)) \)

leads to:

\( (\text{whatever } X, \text{beatle}(X)) \)

which is clearly not equivalent to the original formula.

As a matter of facts, an unquantified variable such as X occurring in:

\[ \text{musician}(X) = \text{beatle}(X) \]

behaves much like an existentially quantified variable.

3.6 Logical inference in the FODL

The concept of logical inference was already met in chapter 1 for formulae containing only propositions. The major difficulty in introducing it for the FODL is the presence of variables and quantifiers.
The fact that, in a particular interpretation, we can show that \( F \Rightarrow G \) is not sufficient to state that \( G \) can be inferred from \( F \).

This can be illustrated by the example of the formula \( F \):

\[
(\text{whatever } X, \text{beatle}(X) \Rightarrow \text{musician}(X)).
\]

As we have seen above, in interpretations over the domain \( D = \{\text{john}, \text{paul}\} \) 'beatle' and 'musician' can be freely chosen as one of 'r1' to 'r4', and the particular choice of 'beatle' and 'musician' are unrelated. If the choice of 'beatle' is 'r4', then 'beatle(X)' is false whatever \( X \) in \( D \). Since 'false \Rightarrow G' is a tautology (it is true whatever the value of \( G \)), the formula:

\[
\text{beatle}(X) \Rightarrow \text{musician}(X)
\]

is true for all \( X \)'s in \( D \) whatever the choice for 'musician', hence \( F \) above is true. But there are other choices of 'beatle' and 'musician' for which it is not the case that the above implication is true whatever \( X \), thus \( F \) is not true in these interpretations, and \( F \) is no tautology.

The basic inference tautologies we shall use in the FOPL are the following, in which \( F \) and \( G \) are closed formulae:

- Modus Ponens:
  \[
  \vdash ( F \land (F \Rightarrow G) ) \Rightarrow G;
  \]

- whatever introduction:
  \[
  \vdash (F \Rightarrow G) \Rightarrow (F \Rightarrow (\text{whatever } X, G));
  \]

- there exists introduction:
  \[
  \vdash (F \Rightarrow G) \Rightarrow ((\text{there exists } X, F) \Rightarrow G).
  \]

We shall say that a closed formula \( G \) can be logically inferred from the set of closed formulae \( F_1, \ldots, F_n \), noted:

\[
F_1, \ldots, F_n \vdash G,
\]

if it is the last element of a sequence of closed formulae in which each formula can be:

- one of \( F_1, \ldots, F_n \);
- a tautology instance;
- a formula inferred, using Modus Ponens, the 'whatever introduction' rule or the 'there exists introduction' rule, from two other formulae occurring prior to it in the sequence.

As is the case in the propositional logic, the concepts of logical consequence and logical inference are one and the same in the FOPL restricted to closed formulae, thus:

\[
\begin{align*}
F & \vdash G \\
\vdash F \rightarrow G \\
F & \vdash G \\
\vdash F \rightarrow G
\end{align*}
\]

are all equivalent, and the words 'theorem' and 'tautology' are merely synonyms in this context.

### 3.6 Theorem proving

A fundamental theorem due to Goedel establishes the fact that 'tautology' and 'theorem' are two synonyms in the FOPL.

An important theoretical result, due to Curry and Church, is that FOPL is semi-decidable:

- if a formula is a tautology, then this can be proved in a finite amount of time;
- if a formula is not a tautology, then we may work forever without getting any answer from a theorem prover. From this can be shown in a finite amount of time.

In order to demonstrate that a particular formula is a tautology, we shall thus use refutation: the negation of this formula is added to the axioms (the known facts) and we demonstrate that they are inconsistent.

### 3.4 The resolution principle

Robinson discovered in 1965 the so-called resolution principle, that is a powerful inference tautology. The reader may be surprised, but it is merely a generalization of Modus Ponens. It states that from 'P or Q or R' and 'not(P) or S or T', we can infer 'Q or R or S or T'.
\((P \lor Q \lor R), (\neg(P) \lor S \lor T) \vdash Q \lor R \lor S \lor T\)

that can be equivalently rewritten:

\[
\begin{align*}
&\vdash \\
&(P \lor Q \lor R) \\
&\text{and} \\
&(\neg(P) \lor S \lor T) \\
&\Rightarrow \\
&Q \lor R \lor S \lor T.
\end{align*}
\]

This process is said to resolve the two clauses \(\neg(P)\) or \(Q\) or \(R\) and \(P\) or \(S\) or \(T\) upon \(P\), producing the resolvent \(Q\) or \(R\) or \(S\) or \(T\). Note that the clauses may be arbitrarily complex, and different in length. The essence of resolution is to take two clauses, one containing a given term \(P\), and the other one containing \(\neg(P)\), to eliminate these from both clauses respectively, and to combine the remainders of the two clauses with \(\lor\) to form a new clause called the resolvent.

The fact that the resolution principle is an inference tautology can be seen very easily. In two-valued logics, \(P\) is either true or false. If \(P\) is true, \(\neg(P)\) is false, hence \(\neg(P)\) or \(Q\) or \(R\) can be simplified as \(Q\) or \(R\). Similarly, if \(P\) is false, then \(P\) or \(S\) or \(T\) can be simplified as \(S\) or \(T\). Combining these with the \(\lor\) operator — remember that \(P\) is either true or false — we conclude that \((P \lor Q)\) or \((S \lor T)\) holds. The latter formula can be simply rewritten as \(P\) or \(Q\) or \(S\) or \(T\), and that is the conclusion of the resolution principle.

We have used clauses that are disjunctions of three literals each just for illustration. There can be any number of literals in the two clauses used for resolution, provided one contains a literal \(P\) and the other contains \(\neg(P)\). The particular case of Modus Ponens:

\[
\vdash P \text{ and } (\neg(P) \text{ or } Q) \Rightarrow Q
\]

is the one in which the first clause is \(P\) and the second one is \(\neg(P)\) or \(Q\), leading to the resolvent \(Q\).

The importance of the resolution principle stems from the fact that it is sufficient to prove any theorem in FOPL
and that it is easy to automitize. An inference tautology is sufficient for theorem proving if it possesses the two necessary properties called soundness and completeness. It is sound in the sense that if a refutation exists for a given formula, then this refutation will be found by resolution. It is complete in the sense that if resolution finds a refutation, then the formula is actually inconsistent with the axioms.

3.3 Herbrand interpretations

A way to avoid the great number of potential substitutions was found by Herbrand in 1930: he defined the so-called Herbrand universe of a formula, that has the nice property that we need only consider the replacement of variables by elements of this universe when looking for a refutation. In other words, a refutation for the formula exists if and only if it exists in the Herbrand universe.

The Herbrand universe of a formula is composed of all the variable free terms that can be built from the constants and functions in the formula. Note that we have to build such terms in all possible manners, including combining previously found ones with the functions that occur in the formula. The reader should notice that functions in the formula cause the Herbrand universe to be infinite: we shall apply them to constants, thus producing some terms, then apply these functions to these terms, producing other terms, and so on. If there are no functions at all in the formula, its Herbrand universe is the set of the constants occurring in it. We cannot afford the Herbrand universe of any formula to be empty, since looking for a refutation by replacing variables in the formula by elements of it would be non-sense. Thus in the case of a formula without any constant, an arbitrary constants, say 'foo', is put initially in the Herbrand universe prior to apply the functions to it.

The ideas of Herbrand have been implemented in 1960 by Davis and Putnam in the form of a computer program. Though it worked for small theorems, it could not demonstrate complicated one, due to the huge amount of work it had to do. Even faster technologies would not have helped it to meet its objectives.