CHAPTER ONE

Propositional logic

1.1 SYNTAX OF PROPOSITIONAL LOGIC

A formal system consists of a set of symbols called an alphabet and a set of rules describing the way in which these symbols may be joined together to form strings of symbols. Our initial formal system is built from the following alphabet:

$p, q, r, s, \ldots,$
$\bot,$
$\neg,$
$\wedge, \vee, \rightarrow, \leftrightarrow,$
$(,),$, ,

Symbols $p, q, r, s, \ldots$ represent an infinite number of atomic statements in the alphabet and may be seen as the building blocks of propositions. Three rows of symbols then show logical connectives that bind these atomic elements together according to the following rules:

a. Symbols $\bot, p, q, r, s, \ldots$ are themselves propositions.

b. If A is a proposition then $\neg A$ is also a proposition.

c. If A and B are both propositions then $A \wedge B$, $A \vee B$, $A \rightarrow B$, and $A \leftrightarrow B$ are also propositions.

d. Strings of symbols not built up according to these rules are not propositions.

Unlike the atomic statement symbols $p, q, r, \ldots,$ symbols A and B represent any proposition and are not themselves part of the formal system being described. Symbols of this kind are often called metasymbols. The rules above define the number of arguments that each logical connective requires to produce a correctly formed proposition, producing a form of valency called the *arity*. Statement symbols

$p, q, r, \ldots$ and the connective $\perp$ are themselves propositions and thus have an arity of zero. A zero-arity connective might at this point seem a little fraudulent, but later we shall see that it does have properties in common with the other connectives. Only one connective symbol is defined with an arity of one, limiting the form of propositions that might be constructed from this symbol and atomic statement symbols. Increasingly large propositions may be constructed by the repeated attachment of this connective to a simple atomic statement as follows:

$$\neg(\neg p) \qquad \neg(\neg(\neg p)) \qquad \neg(\neg(\neg(\neg p)))$$

or to the special connective $\perp$, e.g. $\neg\perp$, $\neg(\neg(\perp))$. All remaining connectives are defined to be of arity two, giving propositions of the form

$$p \wedge q \qquad r \vee (\neg p) \qquad \neg(\neg(\neg q)) \rightarrow \neg q \qquad \perp \leftrightarrow (\neg p)$$

with simple or negated atomic statements. Such connectives may take arguments that are themselves formed from arity-two connectives, creating propositions such as

$$(\neg(\neg q)) \rightarrow (r \vee (\neg p)) \qquad (\neg q) \wedge (r \vee (\neg p))$$

Parentheses are defined as part of the formal system to record the order in which a proposition is constructed from its atomic symbols. A proposition might be built from atomic statements p and q as follows:

$$p \qquad \neg q \qquad \neg p \qquad q$$
$$(p \wedge (\neg q)) \qquad ((\neg p) \wedge q)$$
$$(p \wedge (\neg q)) \vee ((\neg p) \wedge q)$$

but if the same starting propositions are combined in a different order, a different proposition is obtained:

$$p \qquad \neg q \qquad \neg p \qquad q$$
$$p \qquad ((\neg q) \vee (\neg p)) \qquad q$$
$$p \wedge ((\neg q) \vee (\neg p)) \wedge q$$

In order to reduce the number of brackets used in formulas a precedence order for arity one and two connectives is defined as follows:

low precedence $\leftrightarrow \rightarrow \vee \wedge \neg$ high precedence

Connectives with the highest precedence bind most tightly to the objects that they connect. Thus $\neg p \wedge q$ is understood to mean $(\neg p) \wedge q$ because a $\neg$ symbol has greater precedence than a $\wedge$ symbol. Explicit bracketing would have to be included if the alternative proposition, $\neg(p \wedge q)$, were intended. Similarly, proposition $p \wedge \neg q \vee \neg p \wedge q$ represents the first of the two examples constructed above and brackets would have to be retained to represent the second possibility. Symbol $\wedge$ has a higher precedence than $\rightarrow$, so the formula $p \wedge q \rightarrow r$ is assumed to represent $(p \wedge q) \rightarrow r$, a formula in which connective $\wedge$ is first applied to statements p and q then this proposition itself becomes an argument. The alternative proposition, $p \wedge (q \rightarrow r)$ requires brackets to overide the precedence rule.

Sometimes the legal or allowed propositions are called well-formed propositions or well-formed formulas, but we shall simply call them propositions or formulas because we have no interest in constructions that are not well formed. An ill-formed formula such as $p \wedge \vee q$ looks odd, even to the inexperienced eye, so no great analysis is required to remove such problems. In fact, an algorithm that decides whether or not a given proposition is well formed may be written, and the proposition property is said to be decidable. All it requires is a procedure that breaks propositions into symbols and arguments according to the formation rules until only statement symbols or the constant $\perp$ remain.

At this point we should be careful not to attribute a meaning to any of the symbols or propositions: all that is defined is an alphabet of symbols and some rules that specify the ways in which these symbols can be grouped together. In addition to the rules of construction, we might also have rules of deduction that allow further propositions to be derived from an existing set. The simplest and best-known rule may be written as

$$A, A \rightarrow B \vdash B$$

and is known as *modus ponens*. Here a syntactic turnstile symbol ($\vdash$) shows that proposition B follows from propositions with the forms A and $A \rightarrow B$ in which A and B are metasymbols representing any proposition. Thus, from p and $p \rightarrow q$ we may deduce q and from $(r \wedge s)$ and $(r \wedge s) \rightarrow (p \wedge q)$ we may deduce $p \wedge q$ by making appropriate substitutions in *modus ponens*. Multiple applications of *modus ponens* have a chaining effect as in the following derivation:

$$p, p \rightarrow q, q \rightarrow r \vdash r$$

in which q is deduced from the first two propositions then used with the third to produce the final proof. Derivations of this kind may be shown as follows:

1.	p	assumption
2.	$p \rightarrow q$	assumption
3.	q	1, 2 *modus ponens*
4.	$q \rightarrow r$	assumption
5.	r	3, 4 *modus ponens*

Propositions on the left of the syntactic turnstile are assumed, then the proposition on the right is proven from these assumptions. Each line of the derivation follows from earlier proven or assumed propositions in the proof and is in itself a proof. Reasoning of this kind is described as proof theoretic because it depends only on the application of a rule, making no appeal to any meaning that might be given to the symbols. Later in this chapter the Hilbert proof system using *modus ponens* and three axioms is described. In contrast a Gentzen proof system that has eight rules of deduction but only one axiom schema is also described. Chapter 3 includes a proof system called resolution that also has just one deduction rule, but this might be considered a special case of the Gentzen system.

1.2 SEMANTICS OF PROPOSITIONAL LOGIC

A formal system with the alphabet and proposition building rules described in the previous section may be used to construct propositions or to decide if a given string of symbols is a proposition. However, even when correctly formed, a proposition is no more than a string of symbols because it is defined by its syntactic form, the arrangement of its symbols. None of the symbols represents anything more than itself and we should avoid reading any meaning into the symbols themselves at this stage. One possible meaning for the symbols is provided by the semantic functions below, and this particular interpretation of the symbols has such widespread use that the symbols and this particular interpretation are easily confused. A meaning (a semantics) is provided for each of the symbols by defining semantic functions with arities corresponding to the syntactic forms as follows:

Syntactic form	Semantic function	Name
$\perp$	*false*	false
$\neg$	*not*	negation
$\wedge$	*and*	conjunction
$\vee$	*or*	disjunction
$\rightarrow$	*implies*	implication
$\leftrightarrow$	*iff*	mutual implication

First of all, a constant interpretation *false* is provided for the arity-zero symbol $\perp$ then an interpretation *not* is provided for the arity-one symbol $\neg$ in the form of a table as follows:

A	*not(A)*
false	*not(false)*
not(false)	*false*

Thus proposition $\neg\perp$ has the interpretation *not(false)* whereas proposition $\neg(\neg\perp)$ has the interpretation *not(not(false))* defined in the table as equivalent to the constant *false*. This interpretation represents the principle of the excluded middle because it forces a proposition to be either *false* or *not(false)*, excluding any other possibility. Later we outline an alternative semantics that is not so restrictive and proves to have useful properties, but for the moment we remain with two-valued logic. Our interpretation of *false* is the familiar one: something is *false* if it does not accord with our reasoning, if we would consider it wrong. In two-valued logic a statement is true if it is not *false*, so a new symbol *true* may be introduced into the semantic domain as an abbreviation for *not(false)*. This allows a more compact semantic function definition:

A	*not(A)*
false	*true*
true	*false*

It would have been possible to include a symbol in the alphabet of the formal system which would then have been interpreted as the constant *true*. An interpretation of the $\neg$ symbol could then operate directly on the interpretations *true* and *false*. Instead we use the word *true* simply as an abbreviation for *not(false)*. This might seem an unecessary distinction, but later we shall see that there are sometimes advantages in using a minimum number of symbols in an alphabet. More important, it will become clear that the alphabet described above already contains many more connective symbols than are strictly necessary.

Each of the statement symbols p, q, r, s, ... is mapped to a truth value in a valuation which may be shown as a number of valuation functions such as $val(p) = true$. Statements like "grass is red" or "the earth is spherical" are mapped to truth values in valuations, but the choice of truth value involves extralogical considerations connected to colour perception and physics. In electronics these statement symbols might simply represent transistor switches that may be either on or off and an allocation of truth is straightforward. At this point we are not concerned with the philosophical problems of assigning truth values to statements. Instead we just describe the consequences of different assignments to statement symbols p, q, r, ... A set of n statement symbols permits 2^n combinations of possible truth values that are conveniently displayed in the form of a truth table.

Interpretations for the symbols $\wedge$ and $\vee$ are provided by the conjunction and disjunction semantic functions defined as follows:

A	*B*	*A and B*	*A or B*
true	*true*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*

A conjunction of two argument propositions is *true* only when both arguments evaluate to *true*, whereas a disjunction is *false* only when both arguments are *false*.

A valuation provided for each individual atomic statement in a proposition decides the value of the proposition itself because the interpretation of the connectives is fixed by the semantic function definitions. For example, valuations $val(p) = true$ and $val(q) = false$ decide the value of proposition $\neg(\neg p \vee \neg q)$ as follows:

1. $val(\neg(\neg p \vee \neg q))$
2. $not(val(\neg p \vee \neg q))$
3. $not(val(\neg p)\ or\ val(\neg q))$

4. *not(not val(p) or not val(q))*
5. *not(not true or not false)*
6. *not(false or true)*
7. *not(true)*
8. *false*

Valuation *val(¬proposition)* is replaced by the equivalent expression *not(val proposition)* when the outer ¬ symbol is replaced by its meaning. Gradually the valuation moves inwards until all the connective symbols are replaced by their meanings. In practice this is simply a matter of replacing syntactic symbols with their interpretations to give a result like that in line 4. Once this result is obtained, the statement valuations are inserted and the expression evaluated according to the truth table definitions. All of this work produces a result for just one valuation, the valuation for which *val(p)* is *true* and *val(q)* is *false*. In order to economise on effort, all four possible valuations could be deduced in a single truth table with an intermediate valuation:

p	q	$\neg p \vee \neg q$	$\neg(\neg p \vee \neg q)$
true	*true*	*false*	*true*
true	*false*	*true*	*false*
false	*true*	*true*	*false*
false	*false*	*true*	*false*

When a particular set of atomic valuations makes a proposition *true* that valuation is said to "satisfy" the formula or the valuation is said to be a "model" for the formula. But if a valuation makes the proposition *false*, it is said to falsify the formula and is not a model.

The final column in the truth table for proposition $\neg(\neg p \vee \neg q)$ is exactly the same as that shown in the table defining the interpretation of the $\wedge$ symbol, i.e. corresponding atomic valuations produce the same truth values. These two propositions are said to be logically equivalent and this property is shown by the logical equivalence symbol ($\equiv$)

$$A \wedge B \equiv \neg(\neg A \vee \neg B)$$

Logical equivalences allow one proposition to be substituted for another without changing the meaning of an overall expression. This allows propositions to be simplified while retaining their meaning, or perhaps manipulated into special forms that have useful properties. The equivalence noted above is one form of De Morgan's relation, the other form being

$$A \vee B \equiv \neg(\neg A \wedge \neg B)$$

Logical equivalence means that two syntactically different formulas evaluate to the same truth value for all valuations. The equivalence symbol used is another metasymbol because it is not part of the formal system under discussion. Like the natural language used for writing this text, it provides a method of describing the formal system without being part of it.

1.2.1 Tautology and contradiction

Propositions that evaluate to *true* in all valuations are called tautologies whereas those that evaluate to *false* in every valuation are said to be contradictory or unsatisfiable. Propositions that evaluate to *true* in some valuations and *false* in others are said to be satisfiable or contingent. A simple example of a tautology is provided by the disjunction of a proposition and its negation in an expression $A \vee \neg A$ that evaluates as follows:

val($A \vee \neg A$)
val(A) *or val*($\neg A$)
val(A) *or val*(*not* A)

When proposition A evaluates to *true* the first part of this disjunct is *true* and when A evaluates to *false* the second part evaluates to *true*. Since A has to evaluate to either *true* or *false*, the expression above always evaluates to *true*, regardless of the nature of proposition A. In a similar way, the conjunction of a proposition and its negation, $A \wedge \neg A$, always leads to a contradiction because it is unsatisfiable in this interpretation:

val($A \wedge \neg A$)
val(A) *and val*($\neg A$)
val(A) *and not val*(A)

Clearly this valuation must always produce a false result. Both *val*(A) and *not val*(A) must be *true* for the conjunction to be *true*, but this can never occur. Tautology and contradiction statements of this kind may be placed in an equivalence relation with the constants *true* and *false* as follows:

$A \vee \neg A \equiv$ *true*
$A \wedge \neg A \equiv$ *false*

Since a propositional tautology is equivalent to the constant *true*, a negated tautology such as $\neg(A \vee \neg A)$ is equivalent to the constant *false* and vice versa. This is in fact a very important relationship because we shall later see that a standard approach to proving tautology is to prove the negated formula to be a contradiction.

Equivalence relations might permit a reduction in the number of brackets required for a proposition. For example, the propositions

$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
$(A \vee B) \vee C \equiv A \vee (B \vee C)$

make it clear that the order of evaluation is unimportant when two conjunctions or two disjunctions are applied within a single proposition. The semantic functions

defining these connectives are said to be associative, allowing these expressions to be written without brackets as $A \wedge B \wedge C$ and $A \vee B \vee C$.

Two logical equivalences called the distribution rules have the following form:

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

The first case distributes a disjunction over a conjunction; the second case distributes a conjunction over a disjunction. These rules are used to manipulate propositions into the normal forms described later and to simplify propositions by extracting common propositional fragments. For example, proposition p is common to both parts of the following disjunction and may be withdrawn to simplify the formula as follows:

$$\begin{aligned}(p \wedge r) \vee (p \wedge \neg r) &\equiv p \wedge (r \vee \neg r)\\ &\equiv p \wedge \mathit{true}\\ &\equiv p\end{aligned}$$

One part of the resulting proposition is a tautology and may be replaced by the constant *true*, but in conjunction with this constant, the proposition simply reproduces itself.

A generalised form of De Morgan's relation proves to be very useful in later sections and is justified as an extension of the above equivalence definitions. Conjunctions of three propositions may be substituted by equivalences as follows:

$$\begin{aligned}(A \wedge B) \wedge C &\equiv \neg(\neg(A \wedge B) \vee \neg C) && \text{De Morgan}\\ &\equiv \neg((\neg A \vee \neg B) \vee \neg C) && \text{De Morgan}\\ &\equiv \neg(\neg A \vee \neg B \vee \neg C) && \text{distribution}\end{aligned}$$

and it is clear that this equivalence holds for repeated conjunctions with any number of arguments. Thus

$$A \wedge B \wedge C \wedge D \wedge \ldots \equiv \neg(\neg A \vee \neg B \vee \neg C \vee \neg D \vee \ldots)$$

and a similar equivalence holds for repeated disjunctions:

$$A \vee B \vee C \vee D \vee \ldots \equiv \neg(\neg A \wedge \neg B \wedge \neg C \wedge \neg D \wedge \ldots)$$

Interpretations for the symbols $\rightarrow$ and $\leftrightarrow$ are provided by the arity-two semantic functions *imp* and *iff*, standing for implies and "if and only if":

A	*B*	*A imp B*	*A iff B*
true	*true*	*true*	*true*
true	*false*	*false*	*false*
false	*true*	*true*	*false*
false	*false*	*true*	*true*

An implication $A \rightarrow B$ has left- and right-hand subformulas, A and B, called the antecedent and the consequent. A *false* antecedent makes the implication *true*

because anything can be implied from a *false* premiss. On the other hand, a *true* antecedent must lead to a *true* consequent: it would not be correct to deduce a wrong conclusion from the *true* facts. As a result, an implication is only *false* when it is claimed that a *true* antecedent implies a *false* consequent. Truth tables may be used to justify the following equivalences involving implication:

$$\begin{aligned}A \rightarrow B &\equiv \neg A \vee B\\ &\equiv \neg(A \wedge \neg B) \quad \text{De Morgan}\end{aligned}$$

Mutual implication is logically equivalent to the conjunction of two implications:

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

and, since the implications have to be true in both directions, this only occurs when both arguments are *true* or both are *false*. The following alternative logical equivalence follows from this simple observation:

$$A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$$

A mutual implication between two logically equivalent propositions always produces a tautology. For example, a tautology based on De Morgan's relation simply joins two equivalent formulas by a mutual implication:

$$\neg(\neg A \vee \neg B) \leftrightarrow A \wedge B$$

In noting this relationship, we should also note that symbol $\leftrightarrow$ is part of the formal system being described whereas $\equiv$ is a metasymbol used to describe the formal system.

An arity-two function requires a truth table of four rows to describe its semantics. Since the outcome of each row may be either *true* or *false*, there are two possible outcomes for each row and a total of $2 \times 2 \times 2 \times 2 = 16$ possible functions that might be defined. These functions may be divided into two groups of eight, the first of which includes the four connectives already described (*and*,*or*,*imp*,*iff*) together with a further group of four defined as follows:

A	*B*	*A rimp B*	*fst(A,B)*	*snd(A,B)*	*tconst(A,B)*
true	*true*	*true*	*true*	*true*	*true*
true	*false*	*true*	*true*	*false*	*true*
false	*true*	*false*	*false*	*true*	*true*
false	*false*	*true*	*false*	*false*	*true*

The first of these functions is a reverse implication that might have been given the symbol $\leftarrow$ in the formal system. But since this is just an ordinary implication written in reverse, it is not usually included. Nevertheless, this form of implication is particularly useful in logic programming because program statements are more naturally written in the reversed form. Functions *fst* and *snd* are projection operators that project their first or second arguments out as the value of the function.

Notice that these functions are written in prefix notation as opposed to the infix notation used for all other expressions. This notation is standard in functional programming, an area in which the two functions are of fundamental importance. The remaining function maps every argument combination onto the constant *true* and is of little interest in practice.

A second set of eight semantic functions is obtained by negating the results of the first eight described above. Negations of the original four (*and*,*or*,*imp*, *iff*) produce the following functions:

A	*B*	*A nand B*	*A nor B*	*A nimp B*	*A niff B*
true	*true*	*false*	*false*	*false*	*false*
true	*false*	*true*	*false*	*true*	*true*
false	*true*	*true*	*false*	*false*	*true*
false	*false*	*true*	*true*	*false*	*false*

This table may be seen as an interpretation of the propositions $\neg(A \wedge B)$, $\neg(A \vee B)$, $\neg(A \rightarrow B)$ and $\neg(A \leftrightarrow B)$ using the semantic functions already provided for the connectives within these expressions. In fact, it has to be seen in these terms because the formal system defined above contains no symbols that these semantic functions might interpret. It would be possible to add more symbols to the original alphabet and then interpret these symbols with the semantic functions. Three of the functions have accepted syntactic forms: the semantic functions *nand*, *nor* and *niff* are often given syntactic forms $|$, $\downarrow$ and $\oplus$ and these symbols could be included in a formal system. The vertical bar of the *nand* syntax is called a Sheffer stroke and the *niff* function symbol is more often described as the exclusive-or (exor) symbol.

We have shown that interpretations for the symbols $\{\rightarrow, \leftarrow, \leftrightarrow, \wedge\}$ may all be expressed in terms of negation and disjunction interpretations. Semantic functions for the remaining three cases of the first group of eight may be written without any other connectives as

$$fst(A,B) = A$$
$$snd(A,B) = B$$
$$tconst(A,B) = true$$

Since the second group of eight semantic functions is formed by prefixing negations to the earlier eight, we conclude that all sixteen may be expressed in terms of negation and disjunction alone. Thus the set of symbols $\{\neg, \vee\}$ interpreted as above is adequate to generate all sixteen possible semantic functions. There are other adequate sets of symbols containing the connective $\neg$ with an arity-two connective, notably the sets $\{\neg, \wedge\}$ and $\{\neg, \rightarrow\}$. Perhaps more surprising, the *nand* and *nor* semantic functions interpreting symbols $|$ and $\downarrow$ are individually adequate sets, so that either one alone could represent all sixteen semantic functions. Integrated circuit devices that implement either *nand* or *nor* logic are relatively easy to produce because these functions reflect the physical behaviour of transistors. Since any

circuit can be implemented from collections of these elements, they are widely used as building blocks in electronics.

EXERCISES 1.2

1. Produce truth tables representing each of the following propositions and state whether each proposition is a tautology, a contradiction or a contingent proposition.
   a. $(p \wedge \neg q) \vee (\neg p \wedge q)$
   b. $(p \vee \neg q) \wedge (\neg p \vee q)$
   c. $(p \wedge q) \rightarrow (p \vee q)$
   d. $(p \vee q) \rightarrow (p \wedge q)$
   e. $(p \wedge q) \rightarrow r$
   f. $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$
   g. $((p \rightarrow q) \wedge (q \rightarrow r)) \wedge \neg(p \rightarrow r))$
2. Consider the following propositions:
   $(p \rightarrow q) \rightarrow ((q \rightarrow r) \wedge (p \rightarrow r))$
   $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$
   a. Express them in terms of negations and disjunctions alone.
   b. Express them in terms of negations and conjunctions alone.
3. Produce truth tables to evaluate the following propositions for all possible interpretations of their atomic symbols:
   $p \oplus q \rightarrow p \vee q$
   $p \mid q \downarrow p \mid q$

1.3 SEMANTIC TABLEAUX

A semantic tableau is a graphical method of showing the conditions under which a proposition evaluates to *true*. For example, according to the interpretations given earlier, the formula

$$p \wedge \neg q \vee \neg p \wedge q$$

evaluates to *true* when either subformula $p \wedge \neg q$ or subformula $\neg p \wedge q$ evaluates to *true*. The fact that there are two ways of making the formula *true* is shown in the semantic tableau of Figure 1.1 as a splitting between lines 1 and 2. Next we need to know when subformula $p \wedge \neg q$ evaluates to *true* and from the interpretation given earlier it is clear that this occurs only when both p and $\neg q$ both evaluate to *true*. Thus p and $\neg q$ appear along a single path below the subformula in the tableau. A

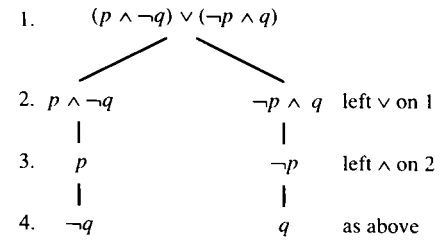

Figure 1.1 Tableau for $(p \wedge \neg q) \vee (\neg p \wedge q)$

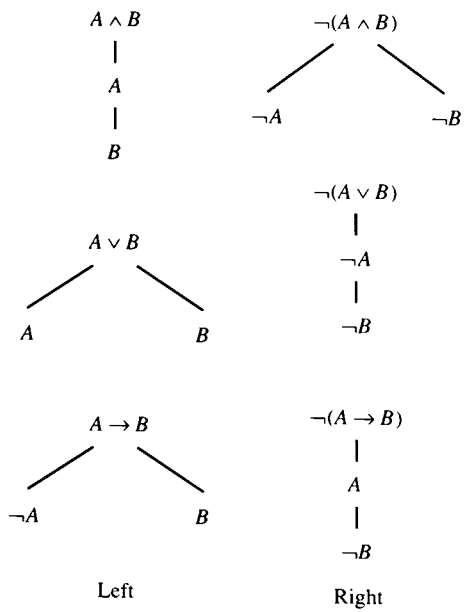

Figure 1.2 Semantic tableau rules

similar argument applies to the subformula on the right-hand side of the tableau. Once a rule has been applied, the formula to which it is applied is of no further interest and might be marked with a tick as having been discharged. Only the subformulas it produces are of further interest. Eventually a situation is reached where no further rule can be applied, and at this point the tableau is complete.

The arguments used here can be generalised into two of the rules shown in Figure 1.2. Formulas with $\vee$ symbols as a principal connective are decomposed by the "left $\vee$" rule whereas an $\wedge$ symbol is decomposed by the "left $\wedge$" rule. At this point, left and right rules are simply those on the left and right of Figure 1.2, but later the left and right tags acquire greater significance. Earlier it was shown that

formula $A \rightarrow B$ is logically equivalent to formula $\neg A \vee B$ and is *true* when either $\neg A$ *or* B is *true*. As a result, the "left $\rightarrow$" rule looks like the "left $\vee$" rule except that one of its subformulas is negated.

Each rule on the right-hand side of Figure 1.2 relates to the same connective as the left, but the whole proposition lies within the scope of a $\neg$ symbol. A justification for the "right $\wedge$" rule is provided by one of the De Morgan equivalences given earlier:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

A proposition of form $\neg(A \wedge B)$ is *true* when either $\neg A$ *or* $\neg B$ is *true* (or both are *true*) and therefore causes branching in the tableau. Justification for the "right $\vee$" rule is also provided by a De Morgan rule, this time in the form

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

A proposition subject to a "right $\vee$" rule is *true* only if both $\neg A$ and $\neg B$ are *true*, so its structure resembles the "left $\wedge$" rule but with negated subformulas. Finally the truth of a negated implication proposition follows from the "right $\rightarrow$" rule because of the following equivalences:

$$\neg(A \rightarrow B)$$
$$\neg(\neg A \vee B)$$
$$A \wedge \neg B$$

A negated formula with an implication principal connective is only *true* when both A and $\neg B$ are *true*, so its structure is that of a "left $\wedge$" with one negated subformula. These inference rules are sometimes divided into two classes: a class of non-branching rules called the alpha or conjunctive set and a class of branching rules called the beta or disjunctive set.

A Hintikka set S is a set of propositions with the following properties:

a. If P is a conjunctive (alpha type) formula in S then both of its subformulas are also in set S.

b. If P is a disjunctive (beta type) formula then one of its subformulas is in set S.

c. An atom and its negation must not both occur in set S.

Looking back at the tableau in Figure 1.1, we see that a Hintikka set may be obtained by collecting propositions along a path from the root to a leaf of a semantic tableau. Since the tree in this figure has two branches, there are two Hintikka sets:

$$S1 = \{p \wedge \neg q \vee \neg p \wedge q, p \wedge \neg q, p, \neg q\}$$
$$S2 = \{p \wedge \neg q \vee \neg p \wedge q, \neg p \wedge q, \neg p, q\}$$

Every formula in a Hintikka set must evaluate to *true* in order to make the root proposition *true*. Consequently, the subset of atoms and negated atoms in a Hintikka set provides a valuation that is a model for the proposition. Every proposition in Hintikka set S1 evaluates to *true* for valuations $val(p) = true$ and $val(q) = false$, and

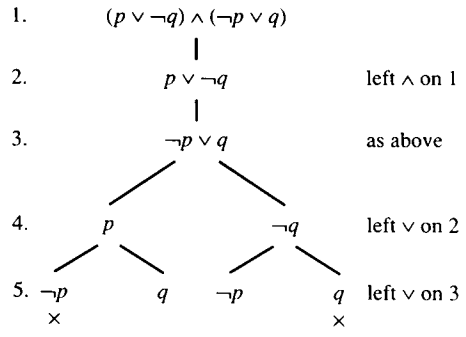

Figure 1.3 Tableau for $(p \vee \neg q) \wedge (\neg p \vee q)$

this set of valuations is a model for the proposition. Similarly, every proposition in set S2 is *true* for the pair of valuations $val(p) = false$ and $val(q) = true$, providing a further model for the proposition. In this simple example the tableau did not tell us anything that could not have been seen in the original formula, but as formulas become larger and more complex the tableau becomes more useful.

The same two rules can be applied to formula $(p \vee \neg q) \wedge (\neg p \vee q)$, giving the tableau of Figure 1.3, but here the result is less obvious than before. A single application of the "left $\wedge$" rule is followed by two applications of the "left $\vee$" rule, producing four paths down through the resulting tableau. Two of these paths are incapable of producing a Hintikka set because they contain both an atom and its negation: one contains both p and $\neg p$, the other q and $\neg q$. Every formula in a Hintikka set has to evaluate to *true* for a valuation indicated by its atomic components, but there can be no valuation of (say) p that makes both p and $\neg p$ *true*. A branch may be closed and marked with a cross as soon as a clashing pair of atoms appears because it could not lead to a satisfying valuation. Hintikka sets may be read from the remaining two open paths of the tableau as follows:

$S1 = \{(p \vee \neg q) \wedge (\neg p \vee q), p \vee \neg q, \neg p \vee q, p, q\}$
$S2 = \{(p \vee \neg q) \wedge (\neg p \vee q), p \vee \neg q, \neg p \vee q, \neg q, \neg p\}$

Set S1 indicates that valuations $val(p) = true$ and $val(q) = true$ provide a model for the proposition and S2 indicates a further model $val(p) = false$ and $val(q) = false$. Thus, the proposition is satisfied in valuations where both p and q are interpreted as *true* or when both p and q are interpreted as *false*, suggesting an alternative equivalent proposition:

$(p \wedge q) \vee (\neg p \wedge \neg q)$

Semantic tableaux are very useful for producing certain equivalent forms of propositions called normal forms; this feature is explored in more detail in Section 1.6.

Inference rules are applied to propositions containing implication symbols in much the same way as in the examples above. A tableau constructed by applying rules to the proposition $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$ is shown in Figure 1.4. Three open

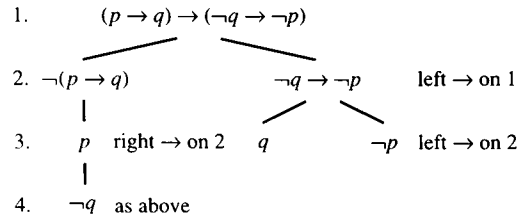

Figure 1.4 Implications in a semantic tableau

paths are visible and three corresponding Hintikka sets suggest that this formula is satisfied by the following three sets of valuations:

	val(p)	*val(q)*
model1	*true*	*false*
model2	*	*true*
model3	*false*	*

Asterisks in the table represent "don't care" situations in which atomic valuations of *true* or *false* do not affect the truth of the formula. This table clearly shows that the formula is *true* when $val(p) = true$ and $val(q) = false$ OR when $val(q) = true$ OR when $val(p) = false$, yielding an equivalent formula $(p \wedge \neg q) \vee q \vee \neg p$. Again a special form of the proposition has been obtained from the tableau, but this is not our immediate concern. If the table above is expanded by including explicit *true* and *false* values for each "don't care" value, it produces a larger table of five lines, but one of these lines is repeated. There are only four possible pairs of valuations for a two-symbol proposition, and from the table we deduce that all four are models for the formula. In other words, the formula is *true* in any valuation and is therefore a tautology, but this is not immediately obvious from the tableau of the proposition itself.

Tautologies are important in the applications of logic to computer science, and methods of deciding if a proposition is a tautology are of great interest. The example above suggests that the direct use of semantic tableaux does not provide an easy procedure for deciding if a given proposition is a tautology, but an alternative approach is possible. If a proposition is a tautology, it is satisfied by every set of valuations and its negation is a contradiction that cannot be satisfied by any valuation. As a result, the semantic tableau produced from a negated tautology has only closed branches containing clashing pairs. This property provides a convenient decision procedure for deciding tautologies. A tableau constructed from the negated formula is shown in Figure 1.5 and confirms the original (unnegated) proposition to be a tautology. Notice that the proposition in line 3 is first broken down by a "right $\rightarrow$" inference rule to give lines 4 and 5, then the proposition in line 2 is decomposed by a "left $\rightarrow$" rule to give line 7. It was not essential to apply the rules in this

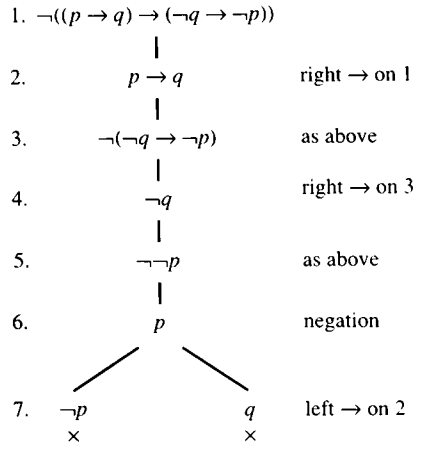

Figure 1.5 A semantic tableau from a negated tautology

order, so the proposition in line 2 could have been discharged before the proposition in line 3. The order in which inferences are applied does not affect the outcome of a deduction, but it does affect the shape of the tableau produced and the efficiency with which a result is obtained. Whenever a choice between a branching and non-branching step is possible, the non-branching inference should be applied first because it delays tableau spanning until it is unavoidable. Directives of this sort are often called *heuristics*.

1.3.1 Extended tableau rules

Disjunction and conjunction are associative operations and the equivalent representations

$$(A \vee B) \vee C \equiv A \vee (B \vee C)$$
$$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$$

may be represented by the unbracketed propositions $A \vee B \vee C$ and $A \wedge B \wedge C$. Since it is unimportant which pair of arguments is evaluated first, it is equally unimportant which connective is first removed in the semantic tableau. As a result, it is possible to define rules that remove two connectives at a time for these operations, leading to the extended tableau rules in Figure 1.6. These new rules are equivalent to two applications of the previous two-symbol rules and are only possible because of the associative nature of the connectives. Extensions of the "right" rules follow from the generalised form of De Morgan's relation:

$$\neg(A \wedge B \wedge C) \equiv \neg A \vee \neg B \vee \neg C$$
$$\neg(A \vee B \vee C) \equiv \neg A \wedge \neg B \wedge \neg C$$

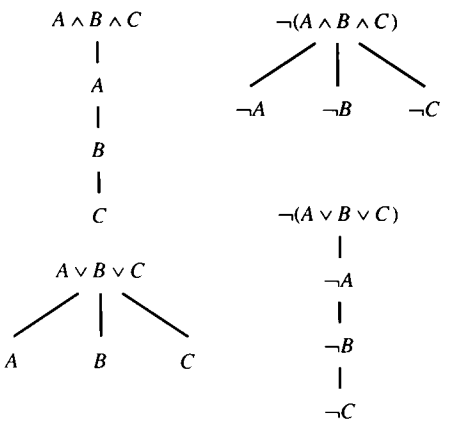

Figure 1.6 Extended tableau rules

A "right $\wedge$" is *true* when any one of its subformulas is *false* whereas a "right $\vee$" is *true* only when all of its subformulas are *false*. In effect, the negated formulas are also associative.

Extended tableau rules are not permitted for the $\rightarrow$ symbol because the interpretation provided for this connective is not associative. As a result, the two formulas $A \rightarrow (B \rightarrow C)$ and $(A \rightarrow B) \rightarrow C$ have different meanings and cannot be reduced with a single extended "left $\rightarrow$" rule. It would of course be possible to assume left association or right association and build extended tableaux on this basis, but this would not be very helpful. In practice the extended tableaux are required for propositions containing only $\wedge$ and $\vee$ symbols, so a generalised implication is not required.

1.3.2 Soundness and completeness

Proof systems are said to be sound if any theorem proven by the system is indeed valid, i.e. in the case of propositions every theorem is a tautology. A propositional theorem is proven by a semantic tableau when it is shown that the negated theorem produces a closed semantic tableau. A negated tautology is certainly a contradiction, so it is sufficient to be sure that a contradiction always produces a closed tableau. This is easily done because the semantic tableau rules implement a systematic search for a satisfying valuation. If one existed it would be found as an open path with a corresponding Hintikka set; the absence of such a path must indicate a contradiction.

Conversely, a proof system is said to be complete if every valid proposition may be proven within the system. In this case we have to show that a closed semantic tableau may be constructed for every contradictory proposition. First of all, we note that a finite tableau may be constructed for any finite proposition. The argument

here is simple: each time a rule is applied, the number of connectives is reduced and eventually there are no more connectives to which rules may be applied. Thus any finite root proposition yields a tableau of some sort. A contradictory root proposition is unsatisfiable and is connected through the rules to closed paths. This may be proven by showing that a satisfiable proposition has a tableau with at least one open path that defines a satisfying valuation. In fact, we have already shown that the Hintikka set defines a model for a satisfiable root proposition and the absence of a Hintikka set implies an unsatisfiable root proposition.

EXERCISES 1.3

1. Produce a semantic tableau for each of the following propositions. From each tableau produce Hintikka sets to show the conditions under which the proposition is true:
   a. $q \rightarrow (p \rightarrow q)$
   b. $(p \vee q) \rightarrow (p \wedge q)$
   c. $(p \wedge q) \rightarrow r$
2. Use semantic tableaux to show that each of the following propositions represents a tautology:
   a. $(p \wedge q) \rightarrow (q \wedge p)$
   b. $(p \wedge q) \rightarrow (p \vee q)$
   c. $(p \rightarrow q) \rightarrow ((q \rightarrow r) \wedge (p \rightarrow r))$
   d. $(p \vee q) \leftrightarrow (q \vee p)$
   e. $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \wedge q) \rightarrow r)$
   f. $(\neg p \vee \neg r) \leftrightarrow \neg(p \wedge r)$

1.4 SEMANTIC ENTAILMENT

Formula A is said to entail formula B if every valuation that makes A *true* also makes B *true*. In other words, any set of atomic valuations that is a model for A is also a model for B. A semantic entailment, sometimes called a logical consequence, is shown with the aid of the semantic turnstile symbol:

$$A \models B$$

Although formula B must evaluate to *true* in any valuation where A evaluates to *true*, it might also be *true* in valuations where A is *false*. Consider as an example the following entailment:

$$(p \wedge \neg q) \vee (\neg p \wedge q) \models p \vee q$$

Truth tables for the propositions on each side of the turnstile are easily constructed:

p	q	$(p \wedge \neg q) \vee (\neg p \wedge q)$	$p \vee q$
true	*true*	*false*	*true*
true	*false*	*true*	*true*
false	*true*	*true*	*true*
false	*false*	*false*	*false*

The formula on the left of the turnstile is in fact the exclusive-or function and evaluates to *true* when just one but not both of its arguments is *true*. An ordinary or function, on the other hand, is *true* when either or both of its arguments are *true*. Clearly the ordinary or function is *true* whenever exclusive-or is *true*, but it is also *true* in one valuation where the exclusive-or is *false*. An entailment such as the one above is equivalent to a valid implication, i.e. the proposition $(p \wedge \neg q) \vee (\neg p \wedge q) \rightarrow p \vee q$ is a tautology because its consequent must be *true* whenever its antecedent is *true*. A tautology can always be written from a semantic entailment in this way, and the semantic tableau of the negated proposition in Figure 1.7 confirms this particular example. Notice, however, that the semantic entailment symbol is a metasymbol and is outside propositional logic, whereas the implication symbol is part of the formal system being examined.

More generally, a proposition B is entailed by (or is a logical consequence of) a set of propositions M, represented as $M \models B$. A specific example is provided by the entailment

$$\{p \rightarrow q, q \rightarrow r\} \models p \rightarrow r$$

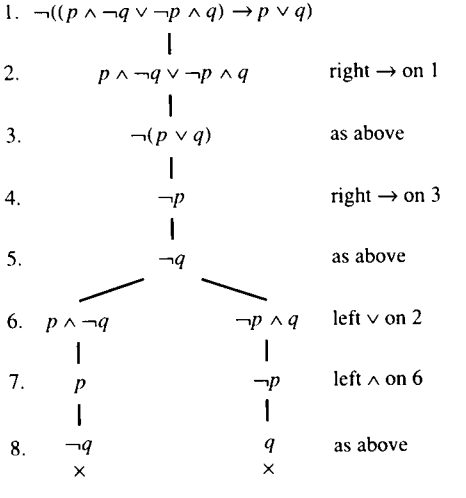

Figure 1.7 Tableau for an entailment

and a first attempt to check this entailment could involve writing a truth table for each of the formulas:

p	q	r	$p \rightarrow q$	$q \rightarrow r$	$p \rightarrow r$
true	*true*	*true*	*true*	*true*	*true*
true	*true*	*false*	*true*	*false*	*false*
true	*false*	*true*	*false*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*
false	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*true*	*false*	*true*
false	*false*	*true*	*true*	*true*	*true*
false	*false*	*false*	*true*	*true*	*true*

It is clear that no valuation simultaneously makes both $p \rightarrow q$ and $q \rightarrow r$ *true* but $p \rightarrow r$ *false*, so the entailment is proven. In fact, there was no need to write out the full truth table because it is only necessary to check that no valuation makes every formula on the left *true* while making the one on the right *false*. Only the rows in which the entailed formula $(p \rightarrow r)$ is *false* need to be checked and, since this only occurs when p is *true* and r is *false*, just rows 2 and 4 have to be checked. This semantic entailment confirms the tautology

$$(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$$

and this is further confirmed by the semantic tableau of Figure 1.8.

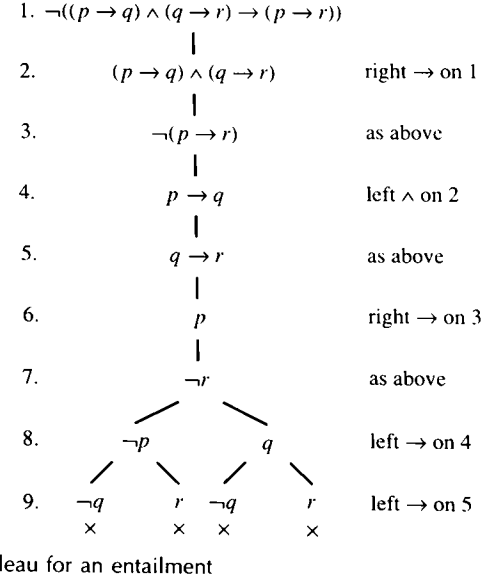

Figure 1.8 Tableau for an entailment

A larger example with more proposition symbols illustrates the advantage of not considering every possible valuation:

$$p, p \rightarrow (q \vee r), q \rightarrow s, r \rightarrow s \models s$$

We have to check there is no valuation that satisfies every proposition on the left while falsifying the proposition on the right of the turnstile. Since the formula on the right is a single symbol s, only a valuation of *false* for this symbol has to be considered. Equally, since all propositions on the left have to be *true*, only valuations that set p to *true* have to be considered and the truth table is reduced to four lines:

q	r	$p \rightarrow (q \vee r)$	$q \rightarrow s$	$r \rightarrow s$
true	*true*	*true*	*false*	*false*
true	*false*	*true*	*false*	*true*
false	*true*	*true*	*true*	*false*
false	*false*	*false*	*true*	*false*

No valuation makes all of these propositions *true*, so there is no valuation that makes every formula on the left *true* at the same time as making the entailed formula *false*. As a result, the entailment is proven and the following formula must be a tautology:

$$p \wedge (p \rightarrow (q \vee r)) \wedge (q \rightarrow s) \wedge (r \rightarrow s) \rightarrow s$$

In each of the examples above, an implication was derived from an entailment, and for the general case this may be shown as

$$A_1 \wedge A_2 \wedge A_3 \wedge \ldots \wedge A_n \rightarrow B$$

If any of the subformulas joined by conjunctions in the antecedent evaluates to *false*, the whole antecedent is *false* and formula B may be either *true* or *false*. If every subformula is *true* the antecedent as a whole is *true* and formula B has to be *true* in order to make the implication *true*. Thus an entailment has the properties indicated by an implication.

It is worth reviewing the procedure used to establish semantic entailment because it has much in common with the Gentzen G system described in the next section. This procedure takes a proposed entailment of the form

$$\{A_1, A_2, A_3, \ldots, A_n\} \models B$$

and systematically attempts to satisfy every formula on the left while falsifying formula B on the right. It is a systematic search for a counterexample that will disprove the entailment, i.e. an attempt to find one set of valuations where each formula on the left evaluates to true when B evaluates to *false*. Failure to find such a valuation set establishes the entailment.

A set of n formulas

$$M = \{A_1, A_2, A_3, \ldots, A_n\}$$

is said to be consistent if there is at least one valuation that makes every formula in the set *true*, i.e. there exists at least one model for the set of formulas. An inconsistent set of formulas has no model and therefore entails any other formula: if there is no valuation that makes every formula on the left *true*, the result of a valuation on the right-hand formula is irrelevant. But if the set M is consistent, an entailment imposes the requirements described above on formula B, then

$$M \models B$$

and B has to be *true* when all of M are *true*. It follows that, if B is entailed by set M, the set of formulas

$$\{A_1, A_2, A_3, \ldots, A_n, \neg B\}$$

is inconsistent because $\neg B$ evaluates to *false* whenever the other formulas in the set all evaluate to *true*. Consistency and inconsistency are the proof theoretic equivalents of satisfiability and contradiction in interpretations. Just as it is easier to prove tautology by showing that the negated formula is a contradiction, it is easier to prove entailment by showing the set $\{A_1, A_2, A_3, \ldots, A_n, \neg B\}$ to be inconsistent. Indirect proofs of this kind are usually described as refutation techniques because they work by refuting negated forms rather than demonstrating the feature directly.

Truth tables may be used to evaluate formulas and to establish entailments, tautologies and contradictions in the way shown above, but the procedure becomes more difficult as the number of statements to be interpreteted increases. A formula with n symbols has 2^n possible valuations, so the size of a truth table increases exponentially with the number of symbols involved. Relatively small examples with four or five symbols require truth tables of 16 or 32 lines, so the approach is already becoming impractical. Modern integrated circuits often have more than a million transistors that have to be represented by distinct symbols and a truth table of greater than $2^{1\,000\,000}$ lines. Clearly we have to develop methods of establishing the truth of a formula without the use of truth tables, and it is this purpose we now address.

EXERCISES 1.4

1. Prove the following entailments by deriving truth tables for propositions on each side of the entailment symbol:
   a. $p \wedge q \models p \vee q$
   b. $\neg p \rightarrow p \models p$
   c. $q \models p \vee q$
   d. $(p \rightarrow r) \wedge (q \rightarrow r) \models (p \vee q) \rightarrow r$
   e. $(p \rightarrow (q \wedge r)) \models (p \rightarrow q) \wedge (p \rightarrow r)$

   Each entailment A $\models$ B indicates that the corresponding proposition A $\rightarrow$ B is a tautology. Draw semantic tableaux to confirm each entailment above.

2. Prove the following entailments from sets of propositions:
   a. $\{p \rightarrow q, r \rightarrow s\} \models (p \wedge r) \rightarrow (q \wedge s)$
   b. $\{p \rightarrow q, r \rightarrow s\} \models (p \vee r) \rightarrow (q \vee s)$
   c. $\{p \leftrightarrow \neg q, q \leftrightarrow \neg r\} \models p \leftrightarrow r$
3. Show that the following entailments do not hold:
   a. $p \vee q \models p \wedge q$
   b. $p \rightarrow q \models p \wedge q$

1.5 A GENTZEN PROOF SYSTEM FOR PROPOSITIONS

In 1935 Gerhard Gentzen laid down deduction rules for two formal proof systems which he called the LK and LJ calculi. The first of these is equivalent to a simpler approach called the G system of deduction later developed by Lyndon. Any LK proof may be translated into an equivalent G proof and, since this latter approach is easier, it makes a better starting-point than the LK form. We shall see that the inference rules of such a Gentzen-style proof system are compatible with the interpretation described in Section 1.2. This particular interpretation of propositional logic symbols has the advantage that it is very easily related to the rules of proof systems. Gentzen himself recognised a close relationship between what he called sequent systems and the concept of an entailment or logical consequence described in the preceding section. An LJ proof system may also be expressed in the style of Lyndon and rules for this variation are presented in Chapter 7. Although there are only small differences between the rules given here and those provided later, the effect of these changes is to define a completely different formal system called intuitionistic logic.

A G proof system may be seen as a proof theoretic form of the reasoning that establishes the extended semantic entailment

$$\{A_1, A_2, \ldots, A_m\} \models \{B_1, B_2, \ldots, B_n\}$$

This entailment is *true* if valuations that make every formula on the left *true* also make at least one of the formulas on the right *true*. In other words, there is no valuation that makes all of the formulas on the left *true* at the same time as making all formulas on the right *false*. As before, the entailment is proved by systematically searching for valuations that make formulas on the left *true* while making the right-hand side *false*. A proof of the entailment is provided by the failure of a systematic and exhaustive search for a counterexample. A Gentzen-style proof system applies rules to sequents of the form

$$[A_1, A_2, \ldots, A_m] \Rightarrow [B_1, B_2, \ldots, B_n]$$

in which the left- and right-hand lists, called respectively the antecedent and succedent, are separated by the sequent symbol ($\Rightarrow$). In the original (LK) formulation

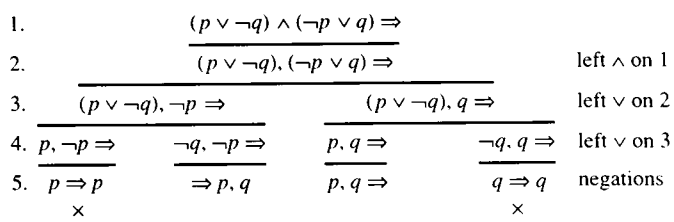

Figure 1.9 A deduction tree

the order of formulas in the lists is important and rules are provided to change the order in which objects appear. This order is unimportant in the G proof system described here, allowing the antecedent and succedent to be treated as sets rather than lists. Nevertheless, the square brackets are sometimes retained to keep the notation compatible with LK usage and in practice we show neither set nor list brackets. Just as in the entailment, a sequent is valid if there is no valuation that makes all of its antecedent formulas *true* without making at least one of the succedent formulas *true*.

As a first example, we demonstrate the conditions under which the proposition $(p \vee \neg q) \wedge (\neg p \vee q)$ is *true* by making it the antecedent of a sequent

$$(p \vee \neg q) \wedge (\neg p \vee q) \Rightarrow$$

A semantic tableau has already been provided for this formula in Figure 1.3 and the G system deduction tree is shown in Figure 1.9. The reasoning now is similar to that used in building the semantic tableau, but the notation in which it is expressed is rather different. In order to make the antecedent *true*, subformulas $(p \vee \neg q)$ and $(\neg p \vee q)$ have separately to be made *true*, so these subformulas are placed in the revised lower antecedent. If one of the subformulas had to be made *false* in order to make the initial antecedent *true*, it would then be moved to the new succedent. Subformula $(p \vee \neg q)$ is in turn *true* when either one of propositions p or $\neg q$ is *true*, generating two subsequents with p in one antecedent and $\neg q$ in the other. Finally in the leftmost branch we see that the sequent is *true* if p and $\neg p$ are both *true* or, equivalently, that p occurs in the antecedent where it has to be satisfied and in the succedent where it has to be falsified. A sequent containing the same atom in both its antecedent and its succedent is called an axiom. A branch may be terminated and marked with a cross whenever an axiom occurs. Similar reasoning is used to build the right-hand side of the deduction tree.

In addition to two axiom sequents labelled with crosses, the deduction tree in Figure 1.9 contains two non-axiom leaf sequents, indicating there are valuations capable of making the proposition true. From the non-axiom sequents $\Rightarrow p, q$ and $p, q \Rightarrow$ we deduce that the original formula is satisfied (is *true*) when both propositions are *false* or when both are *true*. The informal arguments used above may be formalised into a set of inference rules similar to those given earlier for semantic tableaux. Figure 1.10 shows eight G system inference rules as opposed to the six rules required for semantic tableau construction. Two extra rules arise because

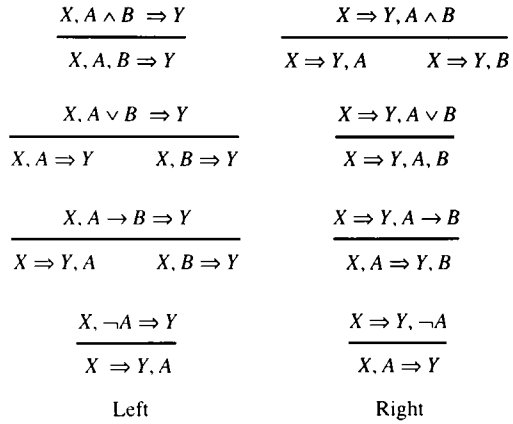

Figure 1.10 G system rules

formulas in the antecedent are distinguished from those in the succedent and "not" rules are required to transfer between the two. If a formula occurs on one side of a sequent symbol, it may be moved to the other side, provided an appropriate correcting negation is made. This seems quite reasonable: if we seek to satisfy antecedent formulas and falsify succedent formulas, a move from one to the other has to be accompanied by negation. Apart from the addition of a "left ¬" rule, the left G rules correspond almost exactly to the semantic tableau rules. At first, the obvious correspondence between right tableau and G system rules might seem surprising, but it has a simple explanation. The right tableau rules derive conditions under which a negated proposition is *true* whereas G rules derive conditions under which the formula itself is *false*. This amounts to the same thing, hence the similar rule structure. Like the semantic tableau in Figure 1.3, the deduction tree in Figure 1.9 required the application of "left ∧", "left ∨" and "left ¬" rules before it reached a point where no further rules could be applied. Unlike the semantic tableau, it required final negation rules to produce axioms.

The whole purpose of G system rules is to demonstrate the subformula conditions under which antecedent formulas are *true* and succedent formulas are *false*. These requirements are entered into subsequents that become the subjects of further applications of the rules. Eventually it becomes impossible to apply any more rules because all the connectives have been exhausted. If the initial antecedent proposition is a contradiction such as $p \wedge \neg p$, the following steps are observed:

$$p \wedge \neg p \Rightarrow$$
$$p, \neg p \Rightarrow$$
$$p \Rightarrow p$$

Antecedent $p \wedge \neg p$ is *true* if both p and $\neg p$ are separately *true*, so the single proposition is replaced by two separate propositions on the left. Proposition $\neg p$ is

in turn *true* only if p is *false*, so $\neg p$ on the left may be replaced by p on the right-hand side. In terms of the G system rules, this deduction consists of a "left $\wedge$" followed by a "left $\neg$" and results in a single-axiom sequent.

If the initial sequent contains an empy antecedent together with the tautology $p \vee \neg p$ in the succedent position, a similar sequence of steps is observed:

$$\Rightarrow p \vee \neg p$$
$$\Rightarrow p, \neg p$$
$$p \Rightarrow p$$

Proposition $p \vee \neg p$ is shown *false* if both p and $\neg p$ are separately shown *false*, and a single proposition on the right of the sequent symbol is replaced by two separate propositions. Proposition $\neg p$ in turn is *false* only if p is shown *true*, so $\neg p$ on the right, false side can be replaced by p on the left, true side. In terms of the rules, we see a "right $\wedge$" followed by a "right $\neg$", leading to an axiom.

Both of these small examples lead to an axiom that does not contain any other propositions, but in the more general case an axiom takes the form

$$X, P, Y \Rightarrow W, P, Z$$

in which symbol P represents the common proposition and X, Y, W, Z represent any other propositions. If P is *false* the sequent is *true* because an antecedent formula is *false*. If P is *true* the sequent is *true* because the succedent has at least one *true* element. In this case other elements of the antecedent may be *true* or *false*, but the sequent can never be falsified. An axiom in the G system proof derived from an antecedent contradiction has much in common with a clashing pair along a path in a semantic tableau.

In the more general case a deduction tree beginning with a sequent of the form

$$\text{contradiction} \Rightarrow$$

must lead to a tree in which every branch terminates in an axiom. A deduction tree with this property is called a proof tree. The object of the systematic search is to find valuations that make the antecedent true, but no valuation can make a contradiction true. In the same way, an initial sequent of the form

$$\Rightarrow \text{tautology}$$

leads to a proof tree because the search for a valuation that makes a tautology false inevitably ends in failure. This line of thought suggests a method of proving propositions to be tautologies: make the proposition a succedent in a sequent and apply the G system rules until the tree is complete. If every branch of the tree terminates with an axiom, the initial proposition is indeed a tautology because an exhaustive search has failed to find any valuation to falsify it. If every possible rule has been applied to a branch and no axiom has resulted, the branch remains open, producing a counterexample to the proof.

As an example of a proof tree produced from a tautology, we now consider a proposition called the contrapositive:

$$\begin{array}{lll}
1. & \Rightarrow (\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p) & \\
2. & \dfrac{(\neg p \rightarrow \neg q) \Rightarrow (q \rightarrow p)}{} & \text{right} \rightarrow \text{ on 1} \\
3. & \dfrac{(\neg p \rightarrow \neg q), q \Rightarrow p}{} & \text{right} \rightarrow \text{ on 2} \\
4. & \dfrac{q \Rightarrow p, \neg p}{} \qquad \dfrac{\neg q, q \Rightarrow p}{} & \text{left} \rightarrow \text{ on 3} \\
5. & p, q \Rightarrow p \quad \times \qquad q \Rightarrow p, q \quad \times & \text{negations}
\end{array}$$

Figure 1.11 A proof tree for a tautology

$$(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$$

In order to prove this, we could prefix the formula with a negation symbol and systematically attempt to satisfy the negated formula. This was the procedure adopted to demonstrate a tautology with a semantic tableau. A failure to find any valuation at all in which the negated formula is *true* then indicates that the original unnegated formula is a tautology. This approach is equivalent to proving the sequent

$$\neg((\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)) \Rightarrow$$

but the first inference rule to be applied to such a sequent would be the "left $\neg$" rule and would result in the sequent

$$\Rightarrow (\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$$

Now the objective is to falsify the succedent, to decompose the sequent into subformulas with the objective of showing this succedent to be *false*. Successive decompositions generate subsequents with antecedents to be shown *true* and succedents to be shown *false*. Figure 1.11 shows that two applications of the "right $\rightarrow$" rule followed by a single "left $\rightarrow$" rule result in subsequents that are easily converted to axioms by the "left $\neg$" and "right $\neg$" rules. The succedent proposition produces a proof tree containing only leaf axioms and is therefore a tautology. It is interesting to note that a semantic tableau proof beginning with the negated formula uses equivalent inference rules in the same order. A sequent is deemed valid if it produces a proof tree, so a valid sequent containing only a succedent formula implies the validity of that formula.

No rule has been offered for mutual equivalence, but this connective can be implemented by substituting one of the following equivalent propositions:

$$(A \leftrightarrow B) \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$
$$(A \leftrightarrow B) \equiv (A \wedge B) \vee (\neg B \wedge \neg A)$$

after which the existing rules may be applied. Consider as an example the following mutual implication:

$$((p \rightarrow q) \rightarrow r) \leftrightarrow (p \rightarrow (q \rightarrow r))$$

Using the first logical equivalence, this problem may be divided into two separate sequents with implications as principal connectives. The first of them is

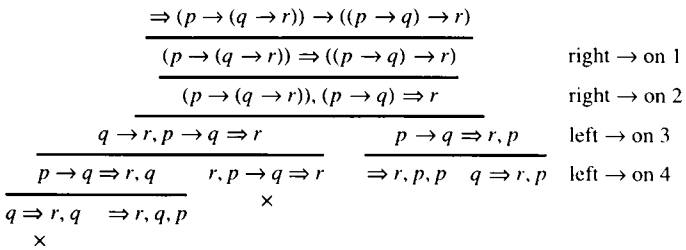

Figure 1.12 A counterexample tree

$$\Rightarrow ((p \rightarrow q) \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$$

It produces a proof tree and is therefore a tautology. Figure 1.12 shows a deduction tree for the reverse implication

$$\Rightarrow (p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow r)$$

It is clear this tree contains a number of leaf sequents that are not axioms. Consequently, the original mutual implication is not valid because counterexample valuations may be deduced from the tree.

A sequent is valid if at least one succedent formula is *true* in every valuation that makes all of its antecedent formulas *true*. A single *false* antecedent formula is sufficient to make the whole antecedent *false* and the sequent valid. Conversely, a single *true* succedent formula is sufficient to make the whole succedent *true* and the sequent valid. This suggests that the formulas might be written in the following form:

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_n$$

in which antecedent formulas are explicitly joined by conjunctions and succedent formulas are explicitly disjuncted. Explicit connectives may also be shown in the entailment

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \models B_1 \vee B_2 \vee \ldots \vee B_n$$

so the meaning of the sequent is expressed in the following entailment:

$$\models A_1 \wedge A_2 \wedge \ldots \wedge A_m \rightarrow B_1 \vee B_2 \vee \ldots \vee B_n$$

A sequent may be interpreted as a general implication from a conjunction of its antecedent formulas to a disjunction of its succedent formulas.

1.5.1 Proof systems, soundness and completeness

Although G rules have been explained in terms of making propositions *true* and *false*, the whole procedure should be seen purely as a proof system without regard to the interpretation. A sequent is proven if it is constructed in a proof tree, i.e. a deduction tree in which every leaf sequent is an axiom. Previously we thought of

decomposing sequents in a way that makes antecedents *true* and succedents *false*, but in proof theory we think of building large sequents from collections of axioms. Instead of seeing leaf sequents as the final step of a decomposition, a proof system sees them as a collection of premisses from which the sequent at the root of the tree is concluded. This means the reasoning steps are upwards in the deduction trees, as shown in this chapter, and the G system rules take the form

$$\frac{\text{premiss}}{\text{conclusion}} \quad \text{or} \quad \frac{\text{premiss1} \quad \text{premiss2}}{\text{conclusion}}$$

It would of course be possible to draw the diagrams the other way up so that initial premisses occur at the top and are joined together on moving down the page to reach a conclusion at the bottom. This approach might have some merit in a final presentation, but nobody actually attempts to prove a sequent by joining together a collection of axioms. It is far better to arrive at the necessary axioms by using G system rules to decompose sequents. One attractive feature of the G proof system is that its rules preserve validity in both directions, so there is no real distinction between premiss and conclusion. An LK system on the other hand only preserves validity in one direction, that of moving from axioms towards the final sequent. As a result, LK sequent deduction trees have traditionally been written with the root sequent at the bottom of the page.

A deduction that begins with the formula to be proven is sometimes said to be *goal oriented* and the reasoning procedure is said to be *top-down* or *backward chained*. A deduction that begins with a set of axioms that are then joined together through inference rules is described as a *bottom-up* or *forward-chained* procedure. Top-down and bottom-up may be interpreted literally in the G system deduction trees as they have been presented here.

A theorem is proven in the G proof system by showing that a sequent with the theorem as succedent can be constructed from axioms alone. However, we need to be sure that the G proof system is both sound and complete, meaning that every proven theorem is actually valid and that every valid theorem is provable. Soundness is easily demonstrated. Axioms are valid sequents and validity is preserved when sequent rules are applied, therefore a final concluding sequent must be valid.

To show completeness, we have to show that a proof tree may be constructed for any valid sequent, but first we need to be sure that a proof tree is always constructed with a finite number of inference rules. To do this, we note that the premiss or premisses of every rule contain fewer logical connectives than the conclusion. It follows that the total number of connectives in subsequents must reduce as we work backwards from the conclusion to more distant premisses. Since the sequent is of finite size, a point must be reached at which all the connectives are used and leaf sequents are visible. Thus the deduction tree obtained from a finite proposition is of finite size and must be either a proof tree or a counterexample tree. A root sequent is valid if and only if every leaf sequent in the tree is an axiom and falsifiable if and only if at least one leaf sequent is not an axiom. We can be sure that every falsifiable sequent yields a counterexample because this requirement motivated the

construction of the rules. Conversely, we are sure that every valid sequent is provable from some proof tree.

EXERCISES 1.5

1. Show that each of the following propositions is valid by adopting each proposition as a root succedent in a G system proof tree:
   a. $q \rightarrow (p \rightarrow q)$
   b. $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$
   c. $((p \rightarrow r) \wedge (q \rightarrow r) \wedge (p \vee q)) \rightarrow r$
   d. $(p \wedge q) \rightarrow (p \vee q)$
   e. $(\neg p \vee q) \leftrightarrow (p \rightarrow q)$
   f. $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$
   g. $\neg (((p \rightarrow q) \wedge (q \rightarrow r)) \wedge \neg(p \rightarrow r))$
   h. $((p \rightarrow q) \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$
   i. $((p \vee r) \wedge (q \vee \neg r)) \rightarrow (p \vee q)$
2. Show that the following propositions are not valid by producing counterexamples:
   a. $(p \vee q) \rightarrow (p \wedge q)$
   b. $(p \rightarrow q) \rightarrow \neg(q \rightarrow p)$
   c. $(p \rightarrow q) \wedge (\neg q \rightarrow \neg p)$

1.6 NORMAL FORMS

The dual proposition of an atomic statement p is the proposition $\neg p$ whereas the dual of $\neg p$ is proposition p, i.e. a statement is converted to its dual by adding or detaching a negation symbol. Atomic propositions and their negations are usually described as literals, and the dual proposition above is sometimes also called the complement of a literal. Propositions are called cubes if they contain just literals and conjunctions or clauses if they contain just literals and disjunctions. These two forms are illustrated by the following examples:

$p \wedge \neg q \wedge r \wedge \neg s$	cube
$p \vee \neg q \vee r \vee \neg s$	clause

Either a cube or a clause may be negated as a whole then simplified by the generalised De Morgan rule. Taking the clause above as an example, we obtain

$\neg(p \vee \neg q \vee r \vee \neg s)$	negated clause
$\neg p \wedge \neg\neg q \wedge \neg r \wedge \neg\neg s$	generalised De Morgan
$\neg p \wedge q \wedge \neg r \wedge s$	simplify

producing the complementary form as a result. A moment's reflection reveals that the dual of a clause is always a cube that may be written directly from an inspection of the clause. Disjunctions are changed to conjunctions and literals are replaced by their duals. Conversely, the dual of a cube always simplifies to a clause.

A proposition is said to be in negation normal form (NNF) if it contains only the connectives $\wedge$ and $\vee$ together with literals and any necessary bracketing. Any proposition not already in NNF may be converted to this form by the application of logical equivalences followed by the movement of negations into literals. Formulas containing implications are converted by the following procedure:

a. Eliminate all occurrences of implication and mutual implication using the following logical equivalences:

$$\begin{aligned} (A \leftrightarrow B) &\equiv (A \rightarrow B) \wedge (B \rightarrow A) \\ &\equiv (\neg A \vee B) \wedge (\neg B \vee A) \\ (A \leftrightarrow B) &\equiv (A \wedge B) \vee (\neg B \wedge \neg A) \\ (A \rightarrow B) &\equiv \neg A \vee B \end{aligned}$$

b. Move negation symbols inwards using De Morgan's rules until each one stands directly in front of an atomic statement, i.e. until it is contained in a literal.

Thus, formula $p \rightarrow q$ is not in negation normal form because it contains an implication, but it may be converted to the formula $\neg p \vee q$, which is in NNF. Equally, $\neg(p \wedge \neg q)$ fails to satisfy the requirement because a leading negation sign applies to the whole subformula inside the brackets, but it can be converted as follows:

$\neg(p \wedge \neg q)$	
$\neg p \vee \neg\neg q$	De Morgan
$\neg p \vee q$	double negation

Any proposition may be expressed in an equivalent negation normal form, but the form obtained is not unique: every proposition has many equivalent NNF propositions.

Two special cases of NNF are defined: disjunctive normal form (DNF) consists of cubes joined together by disjunctions; conjunctive normal form (CNF) consists of clauses joined together by conjunctions. These two forms have the general appearance

$(p \wedge \neg q \wedge r) \vee (s \wedge \neg t) \vee u$	DNF
$(p \vee \neg q \vee r) \wedge (s \vee \neg t) \wedge u$	CNF

CNF has a dual form that is easily found by negation and simplification as follows:

$\neg((p \vee \neg q \vee r) \wedge (s \vee \neg t) \wedge u)$	negated CNF
$\neg(p \vee \neg q \vee r) \vee \neg(s \vee \neg t) \vee \neg u$	De Morgan
$(\neg p \wedge q \wedge \neg r) \vee (\neg s \wedge t) \vee \neg u$	De Morgan

Clearly the dual of a CNF proposition simplifies easily to a DNF proposition and vice versa, but some caution needs to be exercised here. The dual of a given

formula is actually a different formula from the original, equivalent in fact to a negation of the original formula, and in later sections this conversion has the required properties. On other occasions it is necessary to find a CNF for the proposition itself, rather than its negation, and the techniques described below are then used. Any proposition can be expressed in either DNF or CNF, so we immediately have two possible equivalent NNFs. Later we shall see how a conversion to normal form can be a useful first step in deciding the truth of a proposition, so methods of converting formulas to normal forms are required.

1.6.1 Finding disjunctive normal forms

There exists a simple but tedious method of converting a proposition to its equivalent DNF: simply write out the truth table for the formula and read off the valuations that are a model of the formula. The proposition $(p \rightarrow q) \rightarrow r$ evaluates as follows:

p	q	r	$(p \rightarrow q) \rightarrow r$
true	*true*	*true*	*true*
true	*true*	*false*	*false*
true	*false*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*true*	*true*
false	*true*	*false*	*false*
false	*false*	*true*	*true*
false	*false*	*false*	*false*

A DNF of the proposition is obtained by extracting the five valuations for which the proposition is true:

$$(p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge \neg q \wedge r)$$

Although this approach provides an equivalent proposition in DNF, the resulting formula is not as simple as it might be. However, the size of this proposition may be reduced by pairing off cubes and joining such pairs through the distributive rule. For example, cubes 2 and 3 above are joined and simplified as follows:

$(p \wedge \neg q \wedge r) \vee (p \wedge \neg q \wedge \neg r)$	
$(p \wedge \neg q) \wedge (r \vee \neg r)$	distributive rule
$(p \wedge \neg q)$	tautology: $(r \vee \neg r) \equiv \mathit{true}$

because proposition $r \vee \neg r$ is equivalent to *true* and has no effect in conjunction with the other subproposition. If this procedure is repeated for other pairs of cubes, the large DNF is reduced to a much simpler form then to the minimal form

$(p \wedge r) \vee (p \wedge \neg q) \vee (\neg p \wedge r)$	
$(p \wedge \neg q) \vee (p \wedge r) \vee (\neg p \wedge r)$	rearrangement
$(p \wedge \neg q) \vee r$	distributive rule

1. $(p \rightarrow q) \rightarrow r \Rightarrow$
2. $\Rightarrow (p \rightarrow q) \quad r \Rightarrow$ left $\rightarrow$ on 1
3. $p \Rightarrow q$ right $\rightarrow$ on 2

Figure 1.13 Finding the DNF of $(p \rightarrow q) \rightarrow r$

In fact, this much simpler result could have been found directly by using formula equivalences:

$(p \rightarrow q) \rightarrow r$	
$\neg(\neg p \vee q) \vee r$	logical equivalence
$(p \wedge \neg q) \vee r$	De Morgan

Semantic tableaux and G system proofs incorporate the meaning of truth tables in a display and should be capable of producing normal forms for a formula. Figure 1.13 shows the deduction tree that arises when this proposition is made the antecedent of a sequent and G system rules are applied until termination. Leaf sequents $p \Rightarrow q$ and $r \Rightarrow$ arise, indicating that the formula is *true* if p is *true* when q is *false* or, reading from the second leaf sequent, if r is *true*. This requirement is expressed in DNF as $(p \wedge \neg q) \vee r$, the same result as above.

1.6.2 Finding conjunctive normal forms

Those valuations that falsify the proposition $(p \rightarrow q) \rightarrow r$ are read from the truth table above to produce a disjunctive normal form

$$(p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge \neg r)$$

and this proposition is therefore equivalent to the negated proposition $\neg((p \rightarrow q) \rightarrow r)$. It follows that the formula $(p \rightarrow q) \rightarrow r$ itself is equivalent to the proposition

$$\neg((p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge \neg r))$$

and this expression might be simplified to negation normal form by the application of De Morgan's rules. However, as explained earlier, the result of simplifying a negated DNF expression is the dual CNF

$$(\neg p \vee \neg q \vee r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee r)$$

This formula may be simplified with the distributive rules to give a simpler but equivalent CNF

$$(p \vee r) \wedge (\neg q \vee r)$$

In summary, a truth table approach to finding equivalent CNFs proceeds as follows:

a. Write a DNF proposition from those valuations that falsify the truth table proposition.

b. Write the dual CNF of this proposition (a step equivalent to negation and simplification).

c. Use the distributive rule to remove clashing literals from the result of step b.

Although the truth table approach provides considerable insight into the relationships between truth tables, valuations and normal forms, it is not a very practical approach. A truth table has to be constructed before the DNF can be extracted and, as the number of statement variables increases, this method becomes increasingly intractable. The procedure has an obvious exponential complexity because the number of truth table lines to be considered doubles with each extra variable.

A CNF proposition may be found by the direct application of a distributive rule to a DNF expression. Thus the DNF of proposition $(p \to q) \to r$ is converted to a CNF in one step:

$(p \wedge \neg q) \vee r$	DNF
$(p \vee r) \wedge (\neg q \vee r)$	distributive rule

Although this algebraic approach appears attractively simple, it too hides an underlying exponential complexity that creates problems when the examples become larger.

The above example showed that a truth table approach to finding CNFs proceeds by forming a DNF from those valuations that falsify a proposition then taking the dual of the resulting proposition. Apart from having to create the table, the process of simplifying the resulting proposition makes this method unattractive in practice. However, a simplified propositional DNF that falsifies a given proposition may be obtained by constructing a deduction tree with the proposition as its succedent. A CNF may then be written directly as the dual of this formula. Figure 1.14 shows that if the earlier example is made a succedent and inference rules are applied, it generates leaf sequents $\Rightarrow r, p$ and $q \Rightarrow r$, producing the DNF formula

$$(\neg p \wedge \neg r) \vee (q \wedge \neg r)$$

but the dual form of this proposition is easily written as

$$(p \vee r) \wedge (\neg q \vee r)$$

A little practice allows the CNF of a proposition to be read directly from the deduction tree by making the appropriate corrections for the dual form.

As a slightly larger example, we now convert

$$(p \to (q \to r)) \to ((p \wedge s) \to r)$$

$$\begin{array}{lll} 1. & \Rightarrow (p \to q) \to r & \\ 2. & \dfrac{(p \to q) \Rightarrow r}{} & \text{right} \to \text{on 1} \\ 3. & \Rightarrow r, p \quad q \Rightarrow r & \text{left} \to \text{on 2} \end{array}$$

Figure 1.14 Finding a CNF of $(p \to q) \to r$

to normal form. A truth table approach is possible, but this proposition has four distinct statement symbols and would require a table of 16 rows. The substitution of equivalent propositions followed by an algebraic manipulation of the substituted symbols provides the following conversion:

$(p \to (q \to r)) \to ((p \wedge s) \to r)$	
$\neg(\neg p \vee (\neg q \vee r)) \vee (\neg(p \wedge s) \vee r)$	equivalences
$(\neg\neg p \wedge \neg(\neg q \vee r)) \vee ((\neg p \vee \neg s) \vee r)$	De Morgan
$(p \wedge (q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r)$	De Morgan
$(p \wedge q \wedge \neg r) \vee (\neg p \vee \neg s \vee r)$	brackets

In fact, the resulting formula is not only in negation normal form but also in disjunctive normal form. Applications of the distributive rule allow a further conversion to CNF:

$$(p \vee (\neg p \vee \neg s \vee r)) \wedge (q \vee (\neg p \vee \neg s \vee r)) \wedge (\neg r \vee (\neg p \vee \neg s \vee r))$$
$$\mathit{true} \wedge (q \vee (\neg p \vee \neg s \vee r)) \wedge \mathit{true}$$
$$q \vee \neg p \vee \neg s \vee r$$

Clearly the direct use of equivalences followed by an algebraic simplification has become rather more complex than was the case in the earlier example. Although the algebraic method looked attractive with a small proposition, the size of that example disguised the exponential complexity of this procedure. Like the truth table approach, it quickly becomes intractable as the size of the problem increases.

Negation normal forms for this proposition are obtained when a counterexample tree is constructed using the proposition as either the antecedent or succedent. Figure 1.15 shows the tree obtained when this formula is taken as the antecedent, leading to four leaf sequents of the form

$$p, q \Rightarrow r \qquad \Rightarrow p \qquad \Rightarrow s \qquad r \Rightarrow$$

from which a DNF is read as

$$(p \wedge q \wedge \neg r) \vee \neg p \vee \neg s \vee r$$

A counterexample tree constructed with the proposition as succedent is shown in Figure 1.16, producing the leaf sequents

$$p, s \Rightarrow r, p \qquad r, p, s \Rightarrow r \qquad p, s \Rightarrow r, q$$

but the first two of these sequents are axioms and the remaining sequent produces a single clause of CNF

$$\dfrac{\dfrac{\dfrac{(p \to (q \to r)) \to ((p \wedge s) \to r) \Rightarrow}{\Rightarrow p \to (q \to r)}}{p \Rightarrow (q \to r)}}{p, q \Rightarrow r} \qquad \dfrac{\dfrac{(p \wedge s) \to r \Rightarrow}{\Rightarrow p \wedge s \qquad r \Rightarrow}}{\Rightarrow p \qquad \Rightarrow s}$$

Figure 1.15 A deduction tree to obtain DNF

$$\dfrac{\dfrac{\dfrac{\dfrac{\Rightarrow (p \to (q \to r)) \to ((p \wedge s) \to r)}{(p \to (q \to r)) \Rightarrow ((p \wedge s) \to r)}}{(p \to (q \to r)), p \wedge s \Rightarrow r}}{(p \to (q \to r)), p, s \Rightarrow r}}{\dfrac{p, s \Rightarrow r, p}{\times} \qquad \dfrac{q \to r, p, s \Rightarrow r}{\dfrac{p, s \Rightarrow r, q \quad r, p, s \Rightarrow r}{\times}}}$$

Figure 1.16 A deduction tree to obtain CNF

$$\neg p \vee \neg s \vee r \vee q$$

A small increase in the size of the problem and in the number of statement symbols has made the deduction tree approach much more attractive compared to the alternatives. Perhaps more important, the rules of the G system are easily implemented in computer programs, so the process is easily mechanised.

1.6.3 Normal forms in proofs

Propositional normal forms have characteristic features that make them more suitable for certain purposes than propositions in general form. However, the concept of a normal form may be extended to proofs arising from the propositions: specific normal forms of proof are defined by characteristic deduction trees. To illustrate such normal forms, we first find an equivalent normal form for the proposition

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$$

This proposition is an instance of Hilbert's second axiom and is therefore valid, i.e. it is a propositional tautology, *true* in all valuations. A deduction tree developed with this formula as its initial succedent has only leaf axioms, confirming that the formula cannot be falsified. A tree developed from a sequent taking this proposition as antecedent is shown in Figure 1.17 and it generates the leaf sequents

$$p, q \Rightarrow r \qquad p \Rightarrow q \qquad \Rightarrow p \qquad r \Rightarrow$$

This information allows an equivalent DNF proposition to be written as

$$(p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \vee \neg p \vee r$$

The resulting proposition is in both NNF and DNF, and since it is equivalent to the formula from which it was derived, it should behave equivalently. In particular, a sequent of the form

$$\Rightarrow (p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \vee \neg p \vee r$$

should produce a proof tree when subjected to G system inference rules, and Figure 1.18 shows this is indeed the case. The proof in Figure 1.18 uses a generalised

$$\dfrac{\dfrac{(p \to (q \to r)) \to ((p \to q) \to (p \to r)) \Rightarrow}{\Rightarrow (p \to (q \to r))} \qquad \dfrac{(p \to q) \to (p \to r) \Rightarrow}{\dfrac{\Rightarrow p \to q}{p \Rightarrow q} \qquad \dfrac{p \to r \Rightarrow}{\Rightarrow p \quad r \Rightarrow}}}{\dfrac{p \Rightarrow (q \to r)}{p, q \Rightarrow r}}$$

left $\to$ on 1

left $\to$ on 2

Figure 1.17 A deduction from Hilbert's second axiom

$$\dfrac{\dfrac{\dfrac{\Rightarrow (p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \vee \neg p \vee r}{\Rightarrow (p \wedge q \wedge \neg r), (p \wedge \neg q), \neg p, r}}{\dfrac{\Rightarrow (p \wedge q \wedge \neg r), \neg p, r, p}{p \Rightarrow (p \wedge q \wedge \neg r), r, p}\ \times} \qquad \dfrac{\Rightarrow (p \wedge q \wedge \neg r), \neg p, r, \neg q}{\dfrac{\Rightarrow p \wedge q, \neg p, r, \neg q}{\dfrac{\Rightarrow p, \neg p, r, \neg q}{p \Rightarrow p, r, \neg q}\ \times} \qquad \dfrac{\Rightarrow q, \neg p, r, \neg q}{q \Rightarrow q, \neg p, r}\ \times} \qquad \dfrac{\Rightarrow \neg r, \neg p, r, \neg q}{r \Rightarrow \neg p, r, \neg q}\ \times}$$

Figure 1.18 A normal form of proof

"right $\vee$" inference rule that removes all disjunction symbols in one step. This might appear simply as a form of shorthand, since the individual steps could have been written, but they would have produced exactly the same sequent. After this step, the only rules that can be applied are the "right $\wedge$" inference rule and negation inference on the literals.

Suppose more generally that a formula is the succedent in a sequent:

$$\Rightarrow \text{formula}$$

and the application of inference rules generates a proof tree. Suppose further that the formula is converted to disjunctive normal form and again taken as the succedent of a sequent:

$$\Rightarrow \text{formula}_{\text{dnf}}$$

It too must lead to a proof tree, though different inference rules are required to produce axioms. If a particular formula is valid, any other formula claiming to be its equivalent must also be valid. Reasoning from the G system "not" rule, it is clear that the sequent

$$\neg(\text{formula}_{\text{dnf}}) \Rightarrow$$

must also lead to a proof tree because the first rule to be applied would be the "left $\neg$" rule and the sequent would be returned to its previous form. However, the negation of a DNF formula is an easily derived CNF formula, so a sequent of the form

$$\text{formula}_{\text{cnf}} \Rightarrow$$

must also lead to a proof tree (provided the original formula is valid).

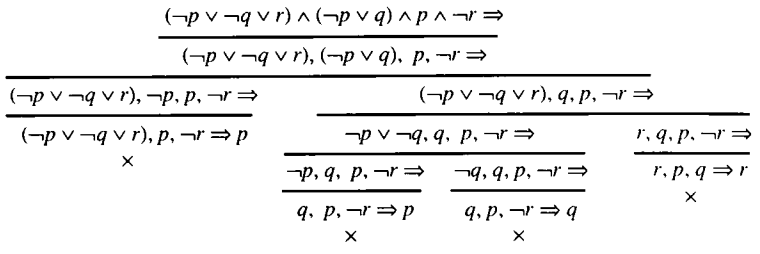

Figure 1.19 A deduction tree from a CNF antecedent

Returning to the example above, we negate and simplify the DNF to obtain an equivalent CNF formula:

$$\neg((p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \vee \neg p \vee r)$$
$$\neg(p \wedge q \wedge \neg r) \wedge \neg(p \wedge \neg q) \wedge \neg\neg p \wedge \neg r \quad \text{De Morgan}$$
$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p \wedge \neg r \quad \text{De Morgan}$$

Note how this is the CNF of the negated formula and could have been obtained from the DNF by inspection, since one is the dual of the other. Having converted the formula to the CNF style of NNF, it might now be made the antecedent of a sequent

$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p \wedge \neg r \Rightarrow$$

and inference rules applied to this sequent should produce a proof tree. The resulting tree is shown in Figure 1.19 and from this we see an interesting property: the whole proof appears to be the dual of that given in Figure 1.18. First of all, a generalised "left $\wedge$" rule is used to remove all conjunctions then a number of "left $\vee$" rules are applied until axioms are obtained through negation inferences. The important point here is that proofs have dual forms related to the dual forms obtainable in NNF.

The familiar style of proof, producing axioms from formulas taken as the succedent of a sequent, is called proof normal form. This procedure systematically attempts to falsify the formula until every branch of the deduction tree produces axioms or counterexamples. If every branch produces an axiom, the falsification attempt has failed and the formula must be valid. The alternative style introduced above, taking the negation normal form of the negated proposition as antecedent, is called refutation normal form. Here the objective is to satisfy the negated form, i.e. to show that the formula has a satisfying valuation. A systematic attempt is made to find such satisfactions, but if every branch of the tree is an axiom, the negated formula represents a contradiction and the original unnegated formula is valid. This technique of demonstrating the validity of a formula by refuting its negation is more characteristic of semantic tableaux and is developed in the resolution method of Chapter 3.

EXERCISES 1.6

1. Use truth tables, equivalences and G system proofs to deduce DNF and CNF equivalents from the following contingent propositions:
   a. $p \rightarrow (q \rightarrow r)$
   b. $(p \rightarrow q) \wedge (q \rightarrow r)$
   c. $(p \rightarrow q) \rightarrow (r \rightarrow \bot)$
2. Use equivalences and G system proofs to produce DNF and CNF equivalents of the following contingent propositions:
   a. $((p \rightarrow q) \rightarrow r) \rightarrow s$
   b. $p \rightarrow (q \rightarrow (r \wedge s))$
   c. $(\neg p \vee q) \rightarrow (\neg r \rightarrow s)$
3. Find disjunctive normal forms for the following tautologies by making each one in turn the antecedent of a G system deduction tree:
   a. $(p \rightarrow q) \wedge (p \rightarrow r) \rightarrow (p \rightarrow (q \wedge r))$
   b. $(p \rightarrow (q \rightarrow r)) \rightarrow (q \rightarrow (p \rightarrow r))$
   c. $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r))$

   Check that the equivalent DNFs obtained are still tautologies by making each one the succedent in a proof tree. Show that the dual CNF propositions produce refutation trees.
4. Find negation normal forms for each of the propositions in the previous exercise by substituting equivalences and simplification.

1.7 A HILBERT PROOF SYSTEM FOR PROPOSITIONS

The Gentzen proof system contained just one axiom scheme and eight rules of deduction. It is easily related to the semantic tableau method because its inference rules are chosen to reflect the intended semantics of the formal system. A Hilbert proof system does not relate to the semantics in such an obvious way but we shall see that the theorems it proves are exactly those proved by the G system. First of all, an alphabet and a set of rules for combining elements of this alphabet are provided as in Section 1.1 then a proof system is defined by giving three axioms and a single deduction rule. Whereas the Gentzen system described earlier has one axiom and eight inference or deduction rules, the Hilbert system defined here has three axioms and just one deduction rule. The structural rules, axioms and deduction rule are as follows:

a. An alphabet of symbols:

$$\neg, \rightarrow, (,) \ , \ , \ , p, q, r, s, \ \ldots$$

b. Rules for building up propositions from the alphabet:

1. Atoms such as p, q, r are propositions.
2. If A and B are both propositions then $\neg A$ and $A \rightarrow B$ are propositions.
3. Nothing else is a proposition.

c. The following axiom schemata:

1. $(A \rightarrow (B \rightarrow A))$
2. $((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$
3. $((\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A))$

d. A rule of deduction called *modus ponens* (MP):

$A, (A \rightarrow B) \vdash_{H} B$

This alphabet obviously contains fewer symbols than the alphabet in Section 1.1, but it was clear from Section 1.2 that the original alphabet contained several connectives that are redundant in the usual interpretation of these symbols. In fact, distinguishable but equivalent Hilbert proof systems may be constructed from any adequate set of connectives. If interpretations are provided by truth tables, there is no great disadvantage in using a more than adequate set of connectives. On the other hand, a Hilbert system is a pure proof system that encapsulates the traditional meaning of logical connectives in a number of axioms, so a larger alphabet requires a greater number of axioms. Although a reduced number of connectives makes the formal system less expressive, it is easier to prove properties for those that are defined. Connectives not contained in this alphabet may then be defined as abbreviations for propositional fragments using elements within the alphabet.

The fact that A can be proven from $\neg\neg A$ within the Hilbert system is indicated by a syntactic turnstile with an H subscript

$\neg\neg A \vdash_{H} A$

A proof of the above statement runs as follows:

1. $\neg\neg A$	assumption
2. $\neg\neg A \rightarrow (\neg\neg\neg\neg A \rightarrow \neg\neg A)$	axiom 1
3. $\neg\neg\neg\neg A \rightarrow \neg\neg A$	1,2 MP
4. $(\neg\neg\neg\neg A \rightarrow \neg\neg A) \rightarrow (\neg A \rightarrow \neg\neg\neg A)$	axiom 3
5. $\neg A \rightarrow \neg\neg\neg A$	3,4 MP
6. $(\neg A \rightarrow \neg\neg\neg A) \rightarrow (\neg\neg A \rightarrow A)$	axiom 3
7. $\neg\neg A \rightarrow A$	5,6 MP
8. A	1,7 MP

A propositional statement to the left of the turnstile is seen as a hypothesis, a basic point from which reasoning begins. The proposition in line 2 is obtained by substituting $\neg\neg A$ for A and $\neg\neg\neg\neg A$ for B in axiom 1 and is therefore an instance of the axiom. *Modus ponens* is then applied to the formulas in lines 1 and 2 to obtain the result in line 3. Every line in a proof of this sort contains a formula proven on the basis of some assumptions, the axioms and *modus ponens*. Further substitutions using axiom 3 are both followed by applications of *modus ponens*, leading eventually to the desired result. It is clear that Hilbert-style proofs are much more difficult and far less intuitively reasonable than corresponding G system proofs. Worse still, apparently trivial relationships sometimes require inordinately many lines to prove them. To counter this problem, it is usual to work from a stock of proven relations, substituting them in later proofs as required. This procedure is justified by a law usually described as the deduction rule.

1.7.1 The deduction rule

The deduction rule states that if a proposition B is deduced from proposition A and a set of propositions M (possibly empty), the proposition $A \rightarrow B$ may be deduced directly from the set M. Symbolically

If $M \cup \{A\} \vdash B$ then $M \vdash A \rightarrow B$

In practice this means that propositions may be taken from the left of the syntactic turnstile and made the antecedent of a new propositional implication on the right. Already we have shown a deduction of the formula A from the formula $\neg\neg$A and this is represented as follows:

$\neg\neg A \vdash_{H} A$

According to the deduction rule, this result might equally well be shown as

$\vdash_{H} \neg\neg A \rightarrow A$

and we have a proposition that may be assumed without any hypotheses. A proposition with this property is called a theorem, and one of the purposes of the deduction rule is to build up a stock of theorems that may be used in deductions in much the same way that axioms are used. This particular example is called the double-negation theorem and its existence permits an occurence of $\neg\neg A$ in a proof to be replaced by A through *modus ponens*. Notice that the implication is only proved in one direction and a further proof is required before it can be used in the other direction:

1. $\neg\neg\neg A \rightarrow \neg A$	dubneg of $\neg A$
2. $(\neg\neg\neg A \rightarrow \neg A) \rightarrow (A \rightarrow \neg\neg A)$	axiom 3
3. $A \rightarrow \neg\neg A$	1, 2 MP

As a result of this deduction, we are able to state another theorem:

$\vdash_{H} A \rightarrow \neg\neg A$

Another useful theorem can be derived from the following deduction:

$A, A \rightarrow B, B \rightarrow C \vdash_{\mathrm{H}} C$

This deduction is easily proved without any axioms:

1.	A	assumption
2.	$A \rightarrow B$	assumption
3.	$B \rightarrow C$	assumption
4.	B	1, 2 MP
5.	C	4, 3 MP

Having proven this result, the deduction theorem can now be applied to give a very useful theorem called the chain rule:

$A \rightarrow B, B \rightarrow C \vdash_{\mathrm{H}} A \rightarrow C$

As the name implies, this rule allows implications to be chained along a series. A glance at the proof should be enough to show that the method could be applied to chains longer than three symbols and is really just an extended version of *modus ponens*.

In order to derive another rule, we prove the following deduction:

$(B \rightarrow A) \vdash_{\mathrm{H}} (\neg A \rightarrow \neg B)$

Taking the single proposition from the left as an assumption, the derivation is as follows:

1.	$B \rightarrow A$	assumption
2.	$\neg\neg B \rightarrow B$	dubneg
3.	$\neg\neg B \rightarrow A$	2, 1 + chain
4.	$A \rightarrow \neg\neg A$	dubneg
5.	$\neg\neg B \rightarrow \neg\neg A$	3, 4 + chain
6.	$((\neg\neg B) \rightarrow (\neg\neg A)) \rightarrow (\neg A \rightarrow \neg B)$	axiom 3
7.	$\neg A \rightarrow \neg B$	5, 6 MP

This is modified through the deduction rule to give

$\vdash_{\mathrm{H}} (B \rightarrow A) \rightarrow (\neg A \rightarrow \neg B)$

and is usually called the contrapositive rule.

One more useful rule is obtained after we prove the deduction

$A \rightarrow (B \rightarrow C) \vdash_{\mathrm{H}} B \rightarrow (A \rightarrow C)$

1.	$A \rightarrow (B \rightarrow C)$	assumption
2.	$(A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$	axiom 2
3.	$(A \rightarrow B) \rightarrow (A \rightarrow C)$	1, 2 MP
4.	$B \rightarrow (A \rightarrow B)$	axiom 1
5.	$B \rightarrow (A \rightarrow C)$	4, 3 + chain

After an application of the deduction theorem, the "exchange of antecedent rule" is obtained:

$\vdash_{\mathrm{H}} (A \rightarrow (B \rightarrow C)) \rightarrow (B \rightarrow (A \rightarrow C))$

Finally we show that $A \rightarrow A$ is a theorem in the Hilbert proof system with the following deduction:

1.	$A \rightarrow ((A \rightarrow A) \rightarrow A) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$	axiom 2
2.	$A \rightarrow ((A \rightarrow A) \rightarrow A)$	axiom 1
3.	$(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)$	1, 2 MP
4.	$(A \rightarrow (A \rightarrow A))$	axiom 1
5.	$(A \rightarrow A)$	3, 4 MP

No assumptions have been made in the above deduction and we can state the final result in the form

$\vdash_{\mathrm{H}} A \rightarrow A$

This theorem is equivalent to the law of the excluded middle in classical logic, i.e. the proposition $\neg A \vee A$ is also a theorem.

1.7.2 Soundness and completeness

A deductive system is sound if every theorem proven in the system is in fact valid. For the propositional Hilbert deduction system this means that a proven theorem is a tautology, true in all valuations, and this requirement may be expressed as

$\vdash_{\mathrm{H}}$ proposition implies $\models$ proposition

As a first step, we note that every theorem is derived from assumptions, axioms and the rule *modus ponens*. Assumptions are discharged at the point where the deduction rule is applied and are in a sense built into the resulting theorem. However, we do have to be sure that the steps leading from assumptions to conclusion preserve the meanings of the assumptions. This will be the case if we can show that the axioms are universally valid, i.e. they are tautologies, and that this validity is preserved by the deduction rule. It is relatively easy to show in a semantic tableau that each of the Hilbert axioms is a tautology and an example for one of the axioms was given earlier. Next we note that the *modus ponens* rule is equivalent to an assumption that implication is a tautology, i.e. in this usage we assume the truth of implication. Now, if A is always true and $A \rightarrow B$ is always true, it follows that B is always true, i.e. it too is a tautology. Thus a Hilbert deduction system is sound.

A formal system is complete if every valid proposition is derivable within the system, essentially the converse of soundness, i.e.

$\models$ proposition implies $\vdash_{\mathrm{H}}$ proposition

If a proposition is a tautology it must be derivable within the deduction system. Luckily, we have already shown that the G proof system is complete, so all we need to show is that a proof in this system can be converted to a Hilbert proof. A

deduction above showed that $\neg A \vee A$ is a theorem in the Hilbert system and we know that it is also equivalent to an axiom in the G system, so any proof in G may be converted to an equivalent proof in H. Thus the Hilbert system is complete.

EXERCISES 1.7

1. Use the Hilbert calculus to prove the following propositions:
   a. $\neg p \rightarrow (q \rightarrow \neg p)$
   b. $\neg(p \rightarrow q) \rightarrow p$
   c. $(p \rightarrow \neg p) \rightarrow \neg p$
   d. $(p \rightarrow q) \rightarrow ((\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p))$
2. Using metasymbols A, B, C, . . . to represent any proposition, show that the following formulas represent theorems:
   a. $\neg(A \rightarrow A) \rightarrow B$
   b. $\neg(A \rightarrow B) \rightarrow \neg B$
   c. $\neg B \rightarrow (B \rightarrow A)$
   d. $A \rightarrow (\neg B \rightarrow \neg(A \rightarrow B))$
   e. $\neg(A \rightarrow B) \rightarrow (B \rightarrow A)$

CHAPTER TWO

First-order logic

Atomic propositions are non-decomposable statements that have to be interpreted as a single entity by either a *true* or *false* valuation. Symbols such as p, q and r might represent the following statements:

p john is taller than mary
q mary is taller than tim
r john is taller than tim

and appropriate valuations for the statements are decided by a little extralogical activity such as applying a tape-measure to the people concerned. A brief examination of the statements reveals that the truth of statements p and q implies the truth of statement r, and this observation might be expressed as an implication

$$p \wedge q \rightarrow r$$

This statement encodes something we know about the property of tallness: if it is *true* that john is taller than mary and that mary is taller than tim, it follows that john is taller than tim. There is nothing wrong with descriptions of this sort, but a problem becomes apparent when the same reasoning is applied to greater numbers of people. Separate atomic statements have to be written for another comparison, say s, t and u, and another formula is required to bind these statements together in an implication relationship like the one above.

Propositional logic by itself has limited expressiveness because each statement has to be accepted as a whole, even though the statement has an obvious internal structure. The property or predicate "is taller than" relates two objects in a sentence and, although the objects being compared might differ in each sentence, the predicate has a common interpretation that might be applied to many pairs of people. Obviously, a human reader decides the truth of a statement such as $p \wedge q \rightarrow r$ by

examining the names of the individuals in the atomic statements while interpreting the predicate "is taller than" in the accepted way. Predicate logic extends propositional reasoning by defining a new form of statement that allows a property or predicate to be separated from the objects to which it is applied. The statements above may be written in this more flexible notation as

Is_taller_than(john, mary)
Is_taller_than(mary, tim)
Is_taller_than(john, tim)

When the person named in the first argument is indeed taller than the person in the second argument, the atomic formula is interpreted as *true* in much the same way as for the propositions above. Furthermore, the transitive nature of tallness is equally well expressed in the predicate form

$$Is\_taller\_than(john, mary) \wedge Is\_taller\_than(mary, tim) \rightarrow Is\_taller\_than(john, tim)$$

conveying the same information as before. However, there is a major difference in that the predicated statements are now connected to each other through their arguments. Each of the argument objects, *john*, *mary* and *tim*, appears twice in the formula above, connecting atomic predicates in a way not possible with propositions. The real advantage of this separation into predicates and objects is that statements may be generalised by introducing variables that represent arbitrary individuals:

$$Is\_taller\_than(x,y) \wedge Is\_taller\_than(y,z) \rightarrow Is\_taller\_than(x,z)$$

This is far more satisfactory because it expresses the relationship in a general form.

None of the propositional work described in Chapter 1 is wasted because that form of logic occurs as a subset of the more extensive logic now described. Predicated statements of the kind described above might contain variables such as x and y, but these variables have to be instantiated with values such as mary and tim, returning us to statements that are essentially propositional in nature. All that is required now is the addition of extra symbols, interpretations and rules that describe the more complex structures outlined above.

2.1 SYNTAX FOR FIRST-ORDER LOGIC

First-order logic introduces strings of symbols called terms and predicates that do not occur in propositional logic, and these new strings have to be correctly formed in just the same way that propositions have to be well formed. Correctly formed terms may be embedded as arguments within predicates that are then joined together by the propositional connectives described earlier.

2.1.1 Terms

A term is defined to be one of the following:

a. Zero-arity symbols called constants, often represented by lower case letters from the beginning of the alphabet, i.e. a, b, c, ... or any one of these letters with numeric subscripts. An infinite number of such constants is available, but the examples that follow use only a small number of constants.

b. Symbols called variables that may be substituted by any other term, usually represented by lower case letters from the end of the alphabet, typically w, x, y and z.

c. Constant symbols that have arity greater than zero and so require other terms as arguments before they themselves are terms. Symbols of this kind, called functions, are given lower case letters in the range f, g, h, ...

Terms are easily constructed from constants and variables because these symbols are themselves terms and any more complex term has to be constructed with the aid of function symbols. If function symbols f and g have arities of one and two, each of the following strings of symbols represents a term:

$f(a)$ $f(f(c))$ $g(b,c)$ $g(f(c), f(a))$ $g(g(b,a), f(b))$

so the procedure for constructing a term is very similar to the procedure for constructing a proposition in the previous chapter. Terms constructed with just constants and function symbols are called ground terms because they represent unchangeable forms. Each one of the examples above is a ground term.

Variables in terms may be substituted with ground terms or even with a different variable. The substitution of term t for variable x is generally shown as $\{t/x\}$, so the substitution of constant c for variable x in a term may be shown as

$$g(f(x), h(b,y))\{c/x\} = g(f(c), h(b,y))$$

Only variable x is replaced by term c; other constants and variables remain unchanged. This pattern of substitutions is summarised in the following rules:

$$b\{a/x\} = b$$
$$y\{a/x\} = y$$
$$y\{a/y\} = a$$
$$f(t_1, t_2, \ldots, t_n)\ \{a/x\} = f(t_1\{a/x\}, t_2\{a/x\}, \ldots, t_n\{a/x\})$$

which tell us that the result of substituting a for x in constant b is to leave b unchanged. Variable y is similarly unchanged when another variable is substituted, but is replaced by the new term when it is the object of substitution. Substitutions in functions are achieved by making the same substitutions in each function argument.

2.1.2 Predicates

First-order logic extends the simple notion of statement symbols to the concept of predicate symbols, usually represented by the upper case letters P, Q, R, ... Each

predicate symbol has an associated arity or rank indicating a number of terms required as arguments to make it into a well-formed formula. A predicate might have an arity of zero and is then equivalent to the simple proposition described in Chapter 1, but upper case symbols are used in full first-order logic. Predicate symbols P and R with arities of two and three have correctly formed strings $P(a,f(b))$ and $R(g(a),b,c)$ and, since the arguments used here are all ground terms, these strings may be described as ground predicates or ground formulas. Variable terms that occur within predicate arguments may be substituted as in the following example:

$$R(g(x),\ h(f(y,b),c))\{a/y\} = R(g(x),\ h(f(a,b),c))$$

In general, the substitution of term t_s for variable x in a predicate R results in the substitution being made in each argument term:

$$R(t_1,t_2,\ \ldots\ ,t_n)\{t_s/x\} = R(t_1\{t_s/x\},\ t_2\{t_s/x\},\ \ldots\ ,\ t_n\{t_s/x\})$$

Terms and relations might differ in each first-order language, depending on the intended interpretation of the language. In addition to this differing base of symbols, there exists a fixed set of logical symbols consisting largely of the propositional symbols described earlier. As we might expect, a small number of additional symbols are required to extend logical reasoning to term symbols.

2.1.3 Logical symbols

In addition to the term and predicate symbols defined above, first-order logic has the following alphabet of logical symbols:

$\bot$
$\neg$
$\wedge, \vee, \rightarrow, \leftrightarrow$
$(,)$,
$\forall, \exists$

and it is clear that all of these symbols except the last two are inherited from propositional logic. At the moment, we are only concerned with the syntax of first-order logic, but it might be helpful to note that the interpretations (semantics) we shall give to the symbols $\forall$ and $\exists$ are respectively "for all" and "there exists". The symbols are usually read as such, but are also described as the universal and existential quantifiers.

Given correctly formed terms and predicates, a formula is defined from the logical symbols by the following rules:

a. Predicates are formulas and the constant $\bot$ is a formula.
b. If A is a formula then $\neg A$ is also a formula.
c. If A and B are formulas then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$ are also formulas.
d. Given a variable x and a formula A then $\forall xA$ and $\exists xA$ are also formulas.

e. A string of symbols not constructed in accordance with these rules is not a formula.

Thus the simplest formula consists of either the constant $\perp$ or a single predicate symbol with an appropriate number of terms as arguments, e.g. $P(b)$, $Q(a,b)$ and $R(a,b,c)$ are formulas. Predicates such as these may be joined together by the logical symbols from propositional logic in much the same way as simple statements were constructed from proposition statements. Thus formulas $\neg Q(a,b)$ and $P(b) \wedge Q(b,c)$ are well formed because they comply with the above rules and are ground formulas because they contain only ground terms.

A formula with unquantified variable terms such as $P(x,y)$ is said to contain free variable arguments that could be substituted with other terms, whereas a formula of the type

$$\forall x \forall y P(x,y)$$

has two variables bound by universal quantifiers. A variable in a formula is bound by a quantifier if it lies within the scope of an appropriate quantification, otherwise it is free. For example, the formula

$$\forall x P(x,y) \vee Q(x)$$

contains one occurrence of x bound by a universal quantifier and a second occurrence of x that lies outside the scope of the quantifer and is therefore free. The single occurence of y is also free. Increasingly large formulas are constructed from smaller ones according to the four rules for creating formulas. Looking in the other direction, we see that a large formula contains correctly formed subformulas and the behaviour of the whole depends on the behaviour of these subformulas.

2.1.4 Substitutions in formulas

Substitutions in formulas are really substitutions in the arguments of predicates within formulas and they follow the pattern of term substitutions described earlier. An attempt to substitute a variable in the constant $\perp$ or in ground formulas has no effect:

$$\perp \{a/x\} = \perp$$
$$Q(b,c)\{a/x\} = Q(b,c)$$

but a substitution applied to a formula with unbound variables reduces to substitutions in the appropriate arguments of each predicate, thus

$$P(x) \wedge Q(x,y)\{a/x\} = P(a) \wedge Q(a,y)$$

Substitutions in a quantified formula are similar, except that only free occurrences of the variable are substituted.

Sets of substitutions are usually labelled with lower case letters from the Greek alphabet, giving the general form

$$\tau = \{t_1/x_1, t_2/x_2, \ldots, t_n/x_n\}$$

For example, the substitution

$$\sigma = \{y/x, d/y\}$$

is applied to formula $P(f(x,a), h(y,z))$ as follows:

$$P(f(x,a), h(y,z))\ \sigma = P(f(y,a), h(d,z))$$

the important point being that constant d is only substituted for the existing y variable, not for the first substituted item, i.e. the substitutions do not "chain" along.

Any term, including another variable, can be substituted in the place of a variable; it amounts to renaming the variable. However, a substituted variable should not become captured on substitution, i.e. it should not be substituted within the scope of a quantification for that symbol. This possibility is illustrated in the substitution

$$\forall x P(x,y)\{x/y\} = \forall x P(x,x)$$

in which free variable argument y is replaced by variable x within the scope of the $\forall x$ quantification, binding the variable and preventing any further substitution. A substituted variable should not change the properties of the formula in which it is substituted and should not become bound on substitution. A substituted term should be free for the variable it replaces, but in this example the variable x is not free for y in the initial formula.

2.1.5 Compositions of substitutions

Multiple substitutions carried out separately might produce a different result from that obtained when the same individual replacements are contained in a single substitution. For example, two substitutions defined as

$$\pi = \{y/x\}$$
$$\rho = \{d/y\}$$

might by applied to a formula one after the other. The composition of substitutions $\pi \circ \rho$, read as "π followed by ρ" indicates that substitution π is followed by ρ, giving the result

$$\begin{aligned} P(f(x,a), h(y,z))\ \pi \circ \rho &= P(f(y,a), h(y,z))\ \rho \\ &= P(f(d,a), h(d,z)) \end{aligned}$$

In this case the second substitution is able to use the previously substituted variable to obtain a result different from that obtained in a single substitution σ.

EXERCISES 2.1

The following substitution sets are used in the examples below:

$\pi = \{a/x, y/z\}$ $\theta = \{b/y\}$ $\rho = \{a/x, y/z, b/y\}$ $\sigma = \{c/x, d/y, d/z\}$

1. Carry out the following term substitutions:
   a. $g(x, f(y,x), h(z))\ \pi$
   b. $g(x, f(y,x), h(z))\ \pi \circ \theta$
   c. $g(x, f(y,x), h(z))\ \rho$
   d. $P(h(x,y,z), f(x))\ \rho$
2. Carry out the following substitutions in formulas:
   a. $(P(x,a) \wedge \neg Q(y,z))\ \pi$
   b. $(P(y) \rightarrow \exists y Q(x,y))\ \theta$
   c. $\forall y(P(y) \rightarrow Q(x,z))\ \pi$
   d. $(\forall y P(f(x,y))) \rightarrow Q(g(y), h(z)))\ \rho$
   e. $\forall x(\exists y P(f(x,z),y) \vee \exists z R(z,h(x,y)))\ \sigma$
   f. $Q(x,y,z,f(x,y))\ \theta \circ \rho$
   g. $Q(x,y,z,f(x,y))\ \rho \circ \theta$

2.2 SEMANTICS FOR FIRST-ORDER LOGIC

An interpretation of a set of propositional logic symbols amounts to no more than a valuation that assigns one of the truth values *true* or *false* to each symbol. Interpretations in first-order logic extend the simple valuations of propositional logic to cover predicates that include argument terms. Just as we assigned a truth value to a single proposition P, we have to assign one to the predicate $P(a)$, but such an assigned value may be different from that of formula $P(b)$. The truth value of a predicate clearly depends on the argument to which it is applied; we need to know all of the truth values for all possible arguments. If the number of arguments is small, it might be possible to list truth values of the predicate applied to each possible argument. More often it is necessary to depend on some understanding of the interpretation of the predicate P. Once a truth value has been found for each predicate in a formula, the truth tables given earlier are used to derive a truth value for the formula as a whole.

A first-order logic interpretation must provide a non-empty universe or domain of discourse D with elements representing constants of the syntax. It must also provide

a. A mapping from constants $\{a, b, c, \ldots\}$ in the formal system to elements of the domain D in the interpretation.

b. A mapping from function symbols $\{f, g, h, \ldots\}$ in the formal system to operations of the same arity in the interpretation.

c. A mapping from predicate symbols $\{P, Q, R, \ldots\}$ in the formal system to relations of the same arity using arguments from the domain D.

A set of constants $\{a, b, c, d\}$ in a formal language might be interpreted by a domain of pet animals $\{rover, pixie, tiddles, fido\}$ as follows:

$$I(a) = rover \qquad I(b) = pixie \qquad I(c) = tiddles \qquad I(d) = fido$$

whereas a set of arity-one predicates $\{P, Q, R\}$ might be interpreted by the relations

$$I(P) = \{(rover), (fido)\} \qquad I(Q) = \{(tiddles)\} \qquad I(R) = \{(pixie)\}$$

An interpretation of the predicates such as this is simply a list of the arguments for which the predicate is *true*, and we conclude that formulas $P(a)$, $P(d)$, $Q(c)$ and $R(b)$ are all *true* in this interpretation. These are the only predicates that are *true* in this interpretation, ensuring that any other combination of predicate and domain element is *false*. As a result, $P(b)$ is *false* because b is interpreted as *pixie* and this domain element does not appear in the set of elements interpreting P.

An alternative approach might map each predicate symbol to another symbol of known meaning, thus the interpretation of the predicate symbols is now

$$I(P) = Dog \qquad I(Q) = Cat \qquad I(R) = Parrot$$

whereas the interpretation of the domain element set $\{a, b, c, d\}$ remains as above. Given an interpretation in terms of a meaningful name and the domain element assignment above, we would like to deduce that formulas $P(a)$, $P(d)$, $Q(c)$ and $R(b)$ are *true* in this interpretation, but here the method does not work very well. Although the predicate names clearly define a distinguishable set, it is not obvious which of the named animals belong in each set. Nevertheless, such an approach works well in applications where the predicate property may be deduced from the domain object, e.g. the prime number predicate applied to a number, $Prime(5)$. Similarly, formulas $Odd(x)$ and $Even(x)$ are predicates applied to natural numbers and their meaning is understood without an explicit listing of all satisfying constants. In this and in many other examples drawn from arithmetic, such an explicit listing is impossible.

Logical connectives are interpreted in exactly the same way as described in Chapter 1 and are necesssary to decide the truth of formulas constructed according to the syntax rules. Consider the interpretation given above applied to the formula

$$P(a) \rightarrow \neg Q(a)$$

observing that constant a is interpreted by *rover* and this domain element appears in the relation interpreting P but not in the one interpreting Q. Thus $P(a)$ is *true* in this interpretation whereas $Q(a)$ is *false*, and the atomic formulas may be replaced by truth values to give a proposition

$$true \rightarrow \neg false$$

An intepretaton of this formula reduces to the following valuation of the proposition:

val(*true* → ¬*false*)
true implies not false
true implies true

which is obviously *true*. As a result, we conclude that the formula is *true* in this interpretation. Distinguishing between the syntactic and semantic forms in this way is a heavy burden, so it makes sense to use just the syntactic form and decide how it is to be used from the context.

Formulas may be manipulated with propositional equivalences in just the same way as simple propositions. Thus the formula above might be expressed in either of the following forms:

$\neg P(a) \vee \neg Q(a)$
$\neg(P(a) \wedge Q(a))$

using a logical equivalence and the De Morgan relation. The last formula might be interpreted as a more recognisable statement that a pet may not be both a dog and a cat.

Variables allow formulas such as the one above to be expressed in a more general way, allowing any constant to replace symbol x in the formula

$$P(x) \rightarrow \neg Q(x)$$

Each different mapping of variables to constants is called an assignment, but in this particular case only the assignment of x influences the value of the formula. When x is assigned to constant b it produces an instantiated formula

$$P(b) \rightarrow \neg Q(b)$$

and in the interpretation given above this formula evaluates to true. In fact, this formula evaluates to *true* "for all" assignments of the variable, making the quantified expression $\forall x(P(x) \rightarrow \neg Q(x))$ *true* in this interpretation.

An arity-one function f in the formal language might be interpreted by the operation *father*, producing as its result the single domain object that is the father of the argument domain object. Such an interpretation may be written as

$$I(f) = \{(rover \mapsto fido)\}$$

showing that *fido* is the (only) father of *rover*. A formula may be defined with interpretations in the domain of pets in mind, thus

$$\forall x(P(x) \rightarrow P(f(x)))$$

and is interpreted as a requirement that the father of any dog is also a dog. This interesting biological fact applies to all animals, not just dogs, and the formula

$$\forall x(Q(x) \rightarrow Q(f(x)))$$

is interpreted to restrict fathers of cats to be cats. Since the same property applies to all predicates, we might be tempted to write a general formula as follows:

$$\forall Animal \forall x(Animal(x) \rightarrow Animal(f(x)))$$

intending to range over both types of animals and instances of animals of one type. Quantifications over instances of the predicate belong to first-order logic. Quantifications over the predicates themselves would require the use of second-order logic and this introduces many unwanted complications. In practice the first-order form is sufficient to express anything of interest and the second-order form is not required.

Two quantifier symbols have been defined, but one of them is redundant in the same sense that many of the propositional connectives in Chapter 1 are redundant. Symbol $\forall$ is usually taken as the more fundamental quantifier and is interpreted as a requirement that a predicate is *true* for all domain elements. Symbol $\exists$ requires that "there exists one" domain element satisfying the predicate, though there might be more, and is defined in terms of the universal quantifier as

$$\exists x P(x) \equiv \neg \forall x \neg P(x)$$

There exists one domain element that satisfies the predicate if it is not the case that all domain objects falsify the predicate. If both sides of this equivalence are negated and the right-hand side simplified with the double-negation rule, the following dual equivalence is obtained:

$$\neg \exists x P(x) \equiv \forall x \neg P(x)$$

Alternatively, if the predicate in the first formula above is replaced by a negated predicate, the following equivalence arises after an application of the double-negation rule:

$$\exists x \neg P(x) \equiv \neg \forall x P(x)$$

It is clear from this equivalence that negations may be "passed over" quantifiers provided the quantifer is changed to its dual form, converting existential quantifiers to universal forms and vice versa.

2.2.1 Some formulas involving natural numbers

Consider the formula

$$\forall x \exists y P(x,y)$$

with an interpretation in which the arguments are natural numbers and predicate P is the relation "greater than". Since this relation is usually represented by the more meaningful symbol >, the formula might be written in the more familiar form

$$\forall x \exists y(y > x)$$

indicating that for all x there exists a number y that is greater than x. Clearly this is *true* because there is no largest number, so a bigger one can always be found. But if the intended interpretation of predicate P is the relation "less than", the following modified version of the formula might be used:

$\forall x \exists y(x > y)$

This formula is *false* in such an interpretation: there does not exist a smaller number for every other natural number, so this interpretation is not a model of the formula. In moving directly from variables to domain elements, we have cheated a little by not describing the syntactic constants associated with natural numbers. Although a more detailed treatment of this relationship is postponed until Chapter 5, we note here that the familiar Arabic representation of natural numbers is an interpretation of the abstract series *zero*, *succ*(*zero*), *succ*(*succ*(*zero*)), . . . described later. Every number is either *zero* or the successor of some other number.

Suppose that we have a language with predicates P and Q of arity one and two together with constants *zero*, *succ*(*zero*), *succ*(*succ*(*zero*)) and as many variables as required. An interpretation of this language is provided as follows:

$I(zero) = 0,$
$I(succ(zero)) = 1,$
$I(succ(succ(zero))) = 2$

$I(P) = \{(0), (1)\},$
$I(Q) = \{(0,1), (0,2), (1,2)\}$

Thus $P(zero)$, $P(succ(zero))$, $Q(zero, succ(zero))$, $Q(zero, succ(succ(zero)))$, and $Q(succ(zero), succ(succ(zero)))$ are *true* whereas all other constants make these predicates *false*. In the light of this interpretation, we now decide the truth of formula

$\forall x \exists y(P(x) \rightarrow Q(x,y))$

by checking every possible assignment of x. In order to be *true* "for all" values of x, this formula has to be *true* for the three constants defined in the formal system. In other words, we have to show that each of the formulas

$\exists y(P(zero) \rightarrow Q(zero,y))$
$\exists y(P(succ(zero)) \rightarrow Q(succ(zero),y))$
$\exists y(P(succ(succ(zero))) \rightarrow Q(succ(succ(zero)),y))$

is *true* in the interpretation. Taking the first of these, we note that $P(zero)$ is *true* in the interpretation because 0 occurs in $I(P)$. As a result, the formula is only *true* if there exists a y such that $Q(zero,y)$ is *true* in this interpretation. Looking at the interpretation $I(Q)$ supplied for Q, we see there are in fact two possible pairs of arguments that satisfy the requirements: (0,1) and (0,2). The second formula is satisfied in a very similar manner when the y variable is assigned to the constant $succ(succ(zero))$ because $Q(zero, succ(succ(zero)))$ is *true* in this interpretation. Finally the third formula is *true* because its antecedent $P(succ(succ(zero)))$ is *false* and this is sufficient to satisfy the implication, regardless of the consequent. There are only three domain elements and the formula is *true* "for all" of them, making the universally quantified formula itself *true* in this interpretation.

2.2.2 Formulas with function symbols

An interpretation of a language with functions must provide an interpretation for each function, providing either an explicit listing or a known meaning. Consider a language with an arity-one predicate P, a function symbol f and with constants *zero*, *succ*(*zero*), *succ*(*succ*(*zero*)) and *succ*(*succ*(*succ*(*zero*))). The constants are given the usual interpretation 0, 1, 2, 3, . . . and function f is given the interpretation

$$I(f) = \{(0 \mapsto 1)\ (1 \mapsto 2)\ (2 \mapsto 3)\ (3 \mapsto 0)\}$$

which is just a modulo 3 increment operation. Predicate P has the interpretation

$$I(P) = \{(0), (2)\}$$

indicating that $P(zero)$ and $P(succ(succ(zero)))$ are *true* in the interpretation whereas $P(succ(zero))$ and $P(succ(succ(succ(zero))))$ are *false*. A truth value for the formula

$$\forall x(P(f(f(x))) \rightarrow \neg P(f(x)))$$

is then deduced by evaluating the formula for all domain elements. For example, substituting the first domain element, we obtain

$$P(f(f(zero))) \rightarrow \neg P(f(zero))$$

and this is evaluated only when the functions themselves have been evaluated. Element *zero* is interpreted as 0 and a single application of function f converts this to element 1. A further application of the function to element 1 results in element 2, so the interpretations are as follows:

$$I(f(zero)) = 1$$
$$I(f(f(zero))) = 2$$

Relation P given above contains element 2 but not 1, so $P(f(f(zero)))$ is *true* in this interpretation whereas $P(f(zero))$ is *false* and the formula above evaluates as follows:

$$true \rightarrow \neg false$$
$$true \rightarrow true$$
$$true$$

If this procedure is repeated for the other three domain elements, the universally quantified formula is also found to be *true*.

A formula is said to be satisfiable in an interpretation if there is some assignment that makes it *true* in that interpretation. A formula is *true* in an interpretation if every assignment makes it *true* in that interpretation. In this case the interpretation is said to be a model of the formula. It is *false* in an interpretation if there is no assignment that makes it *true*. Finally a formula is valid if it is *true* in every assignment in every interpretation; it is a contradiction if *false* in every assignment of every interpretation.

EXERCISES 2.2

1. A formal system has constants a, b, c and d with predicates P, Q, R and S with the following interpretation:

   $I(a) = huey \quad I(b) = duey \quad I(c) = luey \quad I(d) = donald$

   $I(P) = \{(huey), (duey)\}$
   $I(Q) = \{(duey), (luey)\}$
   $I(R) = \{(donald), (mickey)\}$
   $I(S) = \{(donald,huey), (mickey,duey), (mickey,luey)\}$

   Decide whether this interpretation is a model for the following formulas:

   a. $\exists x(P(x) \wedge Q(x))$

   b. $\exists x(P(x) \wedge \neg Q(x))$

   c. $\forall x \forall y(R(x) \wedge (P(y) \vee Q(y)) \rightarrow S(x,y))$

2.3 SEMANTIC TABLEAUX

The move from propositional to predicate logic introduced existentially and universally quantified formulas that may be either *true* or *false* in a particular interpretation. Remember that the objective of the semantic tableau approach is to break down formulas into fragments that would have to be *true* in order to make the root formula *true*. For this reason, we need to know the conditions under which quantified formulas will be *true*. Taking first the existentially quantified formula $\exists xP(x)$, we claim it is *true* if there exists an instantiated predicate $P\{a/x\}$ that is *true*, i.e. if there is a ground formula $P(a)$ that is *true*. This is quite reasonable: the statement that there exists a domain element x which makes predicate $P(x)$ *true* is replaced by a formula containing an element that actually does so. As a result, we have the "left $\exists$" rule shown as one of the semantic tableau rules in Figure 2.1 and the fact that the truth of $P(a)$ establishes the truth of $\exists xP(x)$ is shown by placing it directly along a tableau line below the earlier formula.

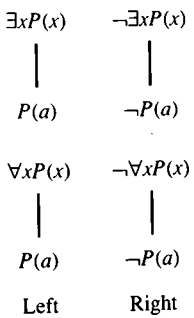

Figure 2.1 Tableau rules for quantifiers

If the left existential rule is used more than once in a given tableau, a fresh constant must be introduced with each use, otherwise a constant is endowed with properties that it might not possess. If the fragment $\exists x P(x)$ is instantiated to give $P(b)$ then $\exists y Q(y)$ is instantiated to give $Q(b)$ in the same tableau, a claim that object b has both properties P and Q is made. Such a claim is unjustified and is only avoided if a fresh constant is introduced with every use of the rule.

A similar line of reasoning leads to the establishment of a left universal rule, but there is a crucial difference in the way this rule is used in comparison with the left existential rule. An existential formula is of no further interest after a left existential inference rule has been applied and is therefore discharged in the same way that propositional formulas are discharged. The fact there exists one object that satisfies a predicate is demonstrated by instantiating just one constant. In contrast, a universally quantified formula is not discharged by the use of the left universal rule, because the quantified formula is *true* if and only if the predicate is *true* "for all" domain elements. As a result, a tableau with fragment $\forall x P(x)$ should have descendents $P(a)$, $P(b)$, etc., exhausting all the domain elements $a, b, c, \ldots$, but the rule instantiates them one at a time. Hopefully, the instantiated formulas allow every branch of the tableau to close, removing the need for any further instantiations. Since a universally quantified formula must be true for all domain elements, a left universal inference may use a constant previously introduced by other left universal or existential inference rules.

As in the propositional case, "right" rules show the conditions under which negated formulas are *true*, but two logical equivalences given earlier show a diagonal relation between left and right rules:

$$\neg \forall x P(x) \equiv \exists x \neg P(x)$$

$$\neg \exists x P(x) \equiv \forall x \neg P(x)$$

A "right $\forall$" inference is equivalent to a "left $\exists$" acting on a negated formula, so this rule has the properties of an existential rule. In particular, a formula is discharged when the rule has been used once, and fresh variables must be introduced with each usage. Similarly, the right existential rule has all the properties of the left universal rule.

As a first example, we demonstrate the validity of the formula

$$\exists x P(x) \rightarrow \exists y P(y)$$

by constructing a semantic tableau with the negated formula at its root. Figure 2.2 shows how a propositional "right $\rightarrow$" rule is first applied to decompose the formula into two separate quantified subformulas. Then, in line 4, the constant a is introduced through an application of the left existential rule. When an instance of an existential formula has been introduced, the original formula can be ticked as having been used. An application of the right existential rule to the formula in line 3 then produces the result on line 5 and it is clear that the single tableau path is now closed. The negated existential formula in line 3 is not discharged by the application of the rule, but the occurrence of $P(a)$ and $\neg P(a)$ on a single path ensures the

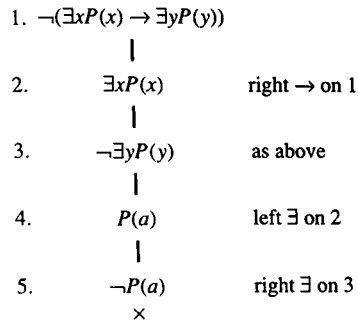

Figure 2.2 A proof tableau

closure of that path. Additions of $\neg P(b)$, $\neg P(c)$ and others by further applications of the rule would not change the result.

In a slightly more ambitious example, we use the same approach to prove the formula

$$\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$$

An application of the "right $\rightarrow$" rule discharges the formula at the root of the tableau in Figure 2.3, producing a choice of two formulas that could be subjected to further inference rules. A left universal inference could be applied to line 2 or

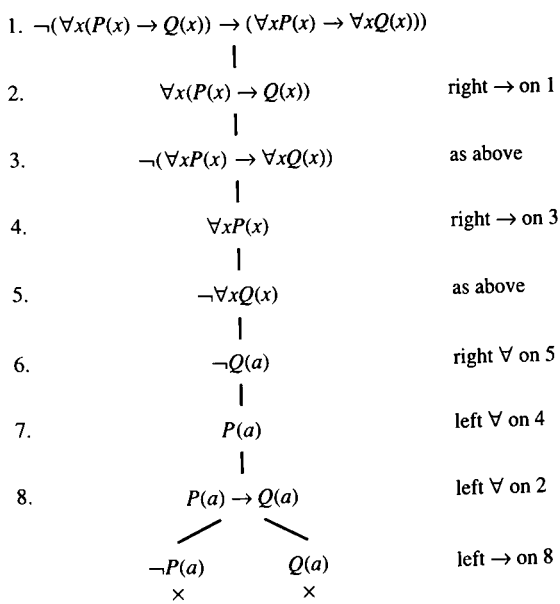

Figure 2.3 A closed semantic tableau

a right implication to line 3, the latter option being taken in this demonstration. A "right $\forall$" rule is then used to discharge the formula in line 5, introducing constant a into line 6. Once introduced, this constant is reused when the "left $\forall$" rule is applied to the formula in line 4, but the formula itself is not discharged. The same constant is used yet again when the "left $\forall$" rule is applied to line 2, producing the formula in line 8. A single propositional "left $\rightarrow$" inference then closes the tableau. Two universally quantified formulas in lines 2 and 4 remain undischarged and could be used to introduce further constants into the tableau, but both paths are closed and further ground formulas would not change anything. Just as it is wise to apply non-splitting rules first, in the propositional case it is equally wise to apply existential rules (either "left $\exists$" or "right $\forall$") first to quantified formulas.

In propositional examples, a tableau might close before all of its formulas have been discharged; but if this does not occur, the tableau terminates when it runs out of connectives to decompose. The introduction of universally quantified formulas generates an endless "self-generated universe" of constants that might continue indefinitely. Closure might now be the only way of obtaining a certain result. A termination problem arises in the following non-valid formula:

$$\exists x P(x) \wedge \exists x Q(x) \rightarrow \exists x (P(x) \wedge Q(x))$$

According to this formula, the existence of objects that separately satisfy predicates P and Q implies the existence of a single object that satisfies both P and Q. This is in fact the misconception avoided by instantiating existential formulas in a given tableau with different constants. Since the formula is not universally *true*, we would not expect its negation to produce a closed tableau, and as shown in Figure 2.4, this

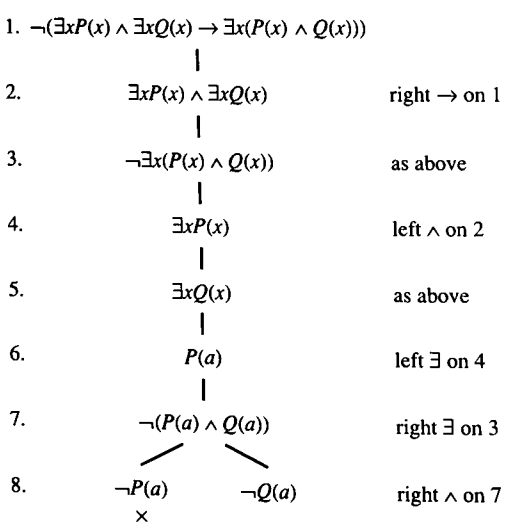

Figure 2.4 An open tableau

is indeed the case. The first part of the tableau proceeds in much the same way as the previous example, but fails to generate ground atoms capable of closing every path. The two existential subformulas in lines 4 and 5 are discharged to give ground formulas $P(a)$ and $Q(a)$, then the right existential rule is applied to the formula in line 3 to give the result in line 8. A propositional rule then produces one closed path and one open path. Since the formula in line 3 is not discharged by one use, it may be used to generate further ground formulas such as $\neg(P(b) \wedge Q(b))$ but this causes further branching. Note that further instantiations using constants c, d, e will not solve the problem, though this is less obvious than in the propositional case.

EXERCISES 2.3

1. Use semantic tableaux to demonstrate the following tautologies:

   a. $\forall xP(x) \rightarrow P(a)$

   b. $\exists x(P(x) \vee Q(x)) \rightarrow (\exists xP(x) \vee \exists xQ(x))$

   c. $(P \rightarrow \forall xQ(x)) \leftrightarrow \forall x(P \rightarrow Q(x))$

   d. $\forall x(P(x) \rightarrow Q(x)) \rightarrow \neg\exists x(P(x) \wedge \neg Q(x))$

   e. $\forall xQ(x) \rightarrow \neg\exists x\neg Q(x)$

   f. $\forall x(P(x) \wedge Q(x)) \leftrightarrow (\forall xP(x) \wedge \forall xQ(x))$

   g. $\forall x(P(x) \rightarrow Q(x)) \wedge \exists x(R(x) \wedge \neg Q(x)) \rightarrow \exists x(R(x) \wedge \neg P(x))$

2. Brackets are not absolutely necessary when writing a formula because predicates such as $P(x)$, $Q(x)$ and $R(x)$ may be written more simply as Px, Qx and Rx. Use this abbreviated notation in a semantic tableau to show that the following formula is valid:

   $$\exists xPx \wedge \forall x(Px \rightarrow Qx) \wedge \forall x(Qx \rightarrow Rx) \rightarrow \exists x(Px \wedge Rx)$$

   Predicates of greater arity, such as $Q(x,y)$ and $R(x,y,z)$ may be shown in a similar style as Qxy and $Rxyz$.

3. Show that the following formula is a contradiction by attempting (and failing) to establish its truth in semantic tableaux:

   $$\forall x\forall y\forall z(Pxy \wedge Pyz \rightarrow Pxz) \wedge \forall x\neg Pxx \wedge \exists x\exists y(Pxy \wedge Pyx)$$

2.4 SEMANTIC ENTAILMENT

Semantic entailment or logical consequence is applied to predicated formulas in exactly the same way as the propositional case. An entailment

$A_1, A_2, \ldots, A_m \models B$

is *true* if there is no interpretation that makes the formula on the right *false* when every formula on the left is *true*. Consider a potential entailment using just one predicate symbol P:

$\neg P(x) \models \forall x \neg P(x)$

in all possible interpretations with the two-element domain $\{1, 2\}$. Each domain element leads to two possible interpretations of the predicate $P(1)$ and $P(2)$, both of which might be *true* or *false*, producing the four possible interpretations in the table:

	*I*1	*I*2	*I*3	*I*4
$P(1)$	*true*	*true*	*false*	*false*
$P(2)$	*true*	*false*	*true*	*false*
$\neg P(1)$	*false*	*false*	*true*	*true*
$\neg P(2)$	*false*	*true*	*false*	*true*
$\forall x \neg P(x)$	*false*	*false*	*false*	*true*

A negation symbol in front of a predicate inverts the truth value obtained for the whole formula, just as for a propositional formula. Since there are only two domain elements in this case, the formula is *true* "for all" elements if it is *true* for both elements 1 and 2. In order to confirm the entailment, we have to show that the truth of $\forall x \neg P(x)$ follows from the truth of $\neg P(x)$. In other words, we have to show that if $\neg P(x)$ evaluates to *true* for any one domain element, the formula must evaluate to *true* for all elements. A glance at the table shows this is not the case: interpretations *I*2 and *I*3 have entries where $\neg P(x)$ evaluates to *true* but in which $\forall x \neg P(x)$ evaluates to *false*.

Turning our attention to a second possible entailment

$\neg P(x) \models \neg \forall x P(x)$

we have to show that formula $\neg \forall x P(x)$ is *true* whenever the formula $\neg P(x)$ is *true*. To do this, we first extend the table above to include two more lines:

	*I*1	*I*2	*I*3	*I*4
$\forall x P(x)$	*true*	*false*	*false*	*false*
$\neg \forall x P(x)$	*false*	*true*	*true*	*true*

This time, a glance at the formula shows the entailment does follow: the right-hand formula is *true* in the three interpretations that make the left-hand side true.

The truth values of formulas $\neg \forall x P(x)$ and $\forall x \neg P(x)$ are clearly different in each interpretation, confirming that these formulas are not equivalent. Remembering the

earlier identity relating universal and existential quantifiers, this second entailment could be written as

$$\neg P(x) \models \exists x \neg P(x)$$

and the entailment seems much more obvious. When a particular instantiation makes $\neg P(x)$ *true*, there must exist a domain element that makes the formula $\exists x \neg P(x)$ *true*.

The number of cases that has to be considered increases with the number of domain elements, so this method of demonstrating entailments is very limited. An alternative approach depends on generating just those constants necessary to demonstrate the entailment, the so-called self-generated universe of the entailment. As an example, we take an entailment with quantified formulas but with a similar structure to one of the propositional entailments examined in Section 1.4:

$$\forall x(P(x) \rightarrow Q(x)), \forall x(Q(x) \rightarrow R(x)) \models \forall x(P(x) \rightarrow R(x))$$

As in the propositional examples, we have to show there is no interpretation that makes all the formulas on the left *true* while making the entailed formula on the right *false*. As a result, the entailment is only made invalid when the entailed formula $\forall x(P(x) \rightarrow R(x))$ is *false*, and this only occurs when there is some constant c that makes formula $P(c) \rightarrow R(c)$ *false*. There might be more, but one example is enough to falsify the entailment. Having generated the constant c in this way, we now note that both formulas on the left are universally quantified and should therefore be *true* for all domain elements. This being the case, they must be *true* for c, since it is one of the domain elements, and the quantified entailment above reduces to the following ground formula entailment:

$$P(c) \rightarrow Q(c), Q(c) \rightarrow R(c) \models P(c) \rightarrow R(c)$$

One constant might not seem enough to demonstrate the entailment, but constant c might be taken as any one constant that makes the entailed formula *false*. Both formulas on the left are universally quantified and therefore *true* for such an arbitrary constant, since they are *true* for all such constants.

An entailment containing only ground formulas is equivalent to a propositional form and is treated in exactly the same way. Formula $P(c) \rightarrow R(c)$ is *false* only when $P(c)$ is *true* and $R(c)$ is *false*, so these are the only valuations that need to be considered. This leaves only $Q(c)$ undecided, but a small truth table shows the consequences of assigning each truth value to this formula:

$Q(c)$	$P(c) \rightarrow Q(c)$	$Q(c) \rightarrow R(c)$	$P(c) \rightarrow R(c)$
false	*false*	*true*	*false*
true	*true*	*false*	*false*

It appears that no valuation of $Q(c)$ makes both of the formulas on the left of the entailment *true* while making the formula on the right *false*; the entailment is

proven. As in the propositional case, the failure of a systematic attempt to falsify the right-hand side of the entailment while satisfying the left-hand side proves validity.

A slightly different strategy is required to demonstrate the following entailment:

$$\exists x P(x), \forall x(P(x) \rightarrow Q(x)) \models \exists x Q(x)$$

Again we aim to prove the entailment by failing to find a counterexample in a systematic search for interpretations that satisfy every formula on the left while falsifying the formula on the right. In this particular example, the right-hand formula is *false* if none of the domain elements makes $Q(x)$ *true*. This means that "for all" domain elements the predicate $Q(x)$ is *false*, giving the formula a universally quantified nature. An existentially quantified formula on the right-hand side of an entailment clearly behaves like a universally quantified formula on the left and is not a suitable starting-point.

However, the left-hand formula $\exists x P(x)$ is *true* if there exists a single constant c making $P(c)$ *true*, and this makes a better starting-point in the search for a counterexample. Given such a constant, we then reason that the right-hand formula $\exists x Q(x)$ is *false* if and only if formula $Q(x)$ is *false* for all domain elements. This being the case, the formula must be *false* for the element c that satisfies formula $P(x)$, since this is just an arbitrarily generated argument. The universally quantified formula on the left must also be *true* for the element c, since it has to be *true* for all domain elements. As a result, the search for a counterexample reduces to a check of the following propositional entailment:

$$P(c), P(c) \rightarrow Q(c) \models Q(c)$$

There is no valuation that makes both formulas on the left *true* while making the one on the right *false*, because the implication must be *false* when $P(c)$ is *true* and $Q(c)$ is *false*. The search for a counterexample fails and the entailment is proven.

EXERCISES 2.4

1. Use the truth value reasoning techniques of the preceding section to demonstrate the following entailments:
   a. $\forall x(P(x) \rightarrow Q(x)), \forall x P(x) \models \forall x Q(x)$
   b. $\forall x(P(x) \rightarrow Q(x)), \forall x(Q(x) \rightarrow \neg R(x)) \models \forall x(P(x) \rightarrow \neg R(x))$
   c. $\forall x(P(x) \vee Q(x) \rightarrow R(x)), \forall x \neg R(x) \models \forall x \neg P(x)$
   d. $\forall x(P(x) \rightarrow Q(x)) \models \forall x P(x) \rightarrow \forall x Q(x)$
2. Use truth value reasoning to demonstrate the following entailments containing existential quantifiers:
   a. $\exists x(P(x) \rightarrow Q(x)), \exists x P(x) \models \exists x Q(x)$
   b. $\forall x(P(x) \vee Q(x) \rightarrow R(x)), \exists x \neg R(x) \models \exists x \neg P(x)$
   c. $\forall x(P(x) \rightarrow Q(x)), \exists x(R(x) \wedge P(x)) \models \exists x(R(x) \wedge Q(x))$

2.5 A GENTZEN PROOF SYSTEM FOR FORMULAS

G system inference rules operate on two sets of formulas in a sequent with the form

antecedent $\Rightarrow$ succedent

and aim to show antecedent formulas to be *true* while showing succedent formulas to be *false*. G system rules demonstrate validity by failing to falsify the sequent itself, leading to a collection of subsequents called axioms.

The Gentzen G proof system described in Chapter 1 is extended to first-order logic by introducing the four new rules shown in Figure 2.5. These rules have a similar form to the semantic tableau rules, including the diagonal relationship between the rules. Both the "left $\exists$" and "right $\forall$" rules are considered to be existential inferences that cause formulas to be discharged after one application, leaving only the instantiated formula. The fact that the "left $\forall$" and "right $\exists$" rules do not discharge their formulas is shown by reproducing the formula below the line in the deduction tree. These four rules have to be used in conjunction with those already given for propositions in Figure 1.10.

As a first example, we consider the formula

$$\exists x(P(x) \wedge Q(x)) \rightarrow \exists xP(x) \wedge \exists xQ(x)$$

The existence of a single object making both predicates P and Q *true* implies the existence of an element making P *true* and the existence of one making Q *true*. This formula is certainly valid, unlike the reverse implication, for which the semantic tableau is given in Figure 2.4. In order to demonstrate its validity, the formula is made the succedent in a sequent and G inference rules are applied as appropriate. Figure 2.6 shows how a right implication rule first generates a sequent to which either a "left $\exists$" or "right $\wedge$" might be applied. Since the second option causes the proof tree to divide, it is delayed and the left existential formula instantiated. A dividing inference has to be applied at line 4, at which point the "right $\wedge$" rule is the only rule that can be applied. Once this is done, the right existential formulas may be instantiated with the previously introduced constant to produce axioms. Notice that the right existential formulas remain in the final sequents, but this is not a problem because these sequents are axioms, and the original formula is proven.

$$\frac{X, \exists xP(x) \Rightarrow Y}{X, P(a) \Rightarrow Y} \qquad \frac{X \Rightarrow \exists xP(x), Y}{X \Rightarrow \exists xP(x), P(a), Y}$$

$$\frac{X, \forall xP(x) \Rightarrow Y}{X, \forall xP(x), P(a) \Rightarrow Y} \qquad \frac{X \Rightarrow \forall xP(x), Y}{X \Rightarrow P(a), Y}$$

Left Right

Figure 2.5 G system quantifier rules

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Rightarrow \exists x(P(x) \wedge Q(x)) \rightarrow \exists x P(x) \wedge \exists x Q(x)}{\exists x(P(x) \wedge Q(x)) \Rightarrow \exists x P(x) \wedge \exists x Q(x)}\ \text{right} \rightarrow \text{ on } 1}{P(a) \wedge Q(a) \Rightarrow \exists x P(x) \wedge \exists x Q(x)}\ \text{left} \exists \text{ on } 2}{P(a), Q(a) \Rightarrow \exists x P(x) \wedge \exists x Q(x)}\ \text{left} \wedge \text{ on } 3}{P(a), Q(a) \Rightarrow \exists x P(x)}\ \text{right} \wedge \text{ on } 4 \qquad \dfrac{P(a), Q(a) \Rightarrow \exists x Q(x)}{P(a), Q(a) \Rightarrow Q(a), \exists x Q(x)}\ \text{right} \exists \text{ on } 4$$

$$\dfrac{P(a), Q(a) \Rightarrow P(a), \exists x P(x)}{\times} \qquad \dfrac{P(a), Q(a) \Rightarrow Q(a), \exists x Q(x)}{\times}$$

Figure 2.6 A G system proof tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Rightarrow \forall x(P(x) \vee Q(x)) \rightarrow \forall x P(x) \vee \forall x Q(x)}{\forall x(P(x) \vee Q(x)) \Rightarrow \forall x P(x) \vee \forall x Q(x)}\ \text{right} \rightarrow \text{ on } 1}{\forall x(P(x) \vee Q(x)) \Rightarrow \forall x P(x), \forall x Q(x)}\ \text{right} \vee \text{ on } 2}{\forall x(P(x) \vee Q(x)) \Rightarrow P(a), \forall x Q(x)}\ \text{right} \forall \text{ on } 3}{\forall x(P(x) \vee Q(x)) \Rightarrow P(a), Q(b)}\ \text{right} \forall \text{ on } 4}{\forall x(P(x) \vee Q(x)), P(a) \vee Q(a) \Rightarrow P(a), Q(b)}\ \text{left} \forall \text{ on } 5$$

$$\dfrac{\forall x(P(x) \vee Q(x)), P(a) \Rightarrow P(a), Q(b)}{\times}\ \text{left} \vee \text{ on } 6 \qquad \forall x(P(x) \vee Q(x)), Q(a) \Rightarrow P(a), Q(b)$$

Figure 2.7 A non-terminating tree

As a second example, we try to prove the validity of the formula

$$\forall x(P(x) \vee Q(x)) \rightarrow (\forall x P(x) \vee \forall x Q(x))$$

Figure 2.7 shows how two right propositional inference rules are followed by two applications of the right universal rule, generating two differently labelled constants. Fresh constants are required for each application of either the left existential or right universal rules, leaving two different predicates instantiated with two different constants, $P(a)$ and $Q(b)$. Finally the left universal rule is applied, generating ground atoms that close one branch of the tree. The remaining branch of the tree is the non-axiom sequent

$$\forall x(P(x) \vee Q(x)), Q(a) \Rightarrow P(a), Q(b)$$

and further instantiations of the universally quantified formula do not lead to a proof tree. The situation is similar to Figure 2.4, because a deduction has not terminated and the whole matter is left unresolved.

EXERCISES 2.5

1. Show that the following formulas are valid by constructing a proof tree, taking each formula as the initial succedent:

   a. $\forall x P(x) \rightarrow \exists x P(x)$

   b. $\forall x(P(x) \rightarrow Q) \rightarrow (\exists x P(x) \rightarrow Q)$

   c. $\neg\exists x(P(x) \wedge \neg Q(x)) \rightarrow \forall x(P(x) \rightarrow Q(x))$

   d. $\forall x P(x) \vee \exists x Q(x) \rightarrow \exists x(P(x) \vee Q(x))$

2. By constructing a proof tree, demonstrate the validity of the following formulas:

   a. $\forall x \forall y R(x,y) \leftrightarrow \forall y \forall x R(x,y)$

   b. $\exists x \exists y R(x,y) \leftrightarrow \exists y \exists x R(x,y)$

   c. $\exists x \forall y R(x,y) \rightarrow \forall y \exists x R(x,y)$

   Two of these formulas are mutual implications and one is a simple conditional. Explore the result when the conditional of the last example is reversed.

3. Prove that the following formulas are valid:

   a. $\forall x(P(x) \rightarrow \neg Q(x)) \wedge \exists x(R(x) \wedge P(x)) \rightarrow \exists x(R(x) \wedge \neg Q(x))$

   b. $\exists x(P(x) \rightarrow \neg Q(x)) \rightarrow (\forall x(R(x) \wedge P(x)) \rightarrow \exists x(R(x) \wedge \neg Q(x)))$

2.6 NORMAL FORMS

Normal forms are standard methods of writing formulas or proofs that reveal properties not obvious in the unnormalised form. A number of such forms were defined for the propositional subset of first-order logic and further definitions connected to quantified formulas are now given.

2.6.1 Prenex normal form

A formula in prenex normal form has all its quantifier symbols to the left of a collection of predicate and logical connective symbols called the matrix. If symbol Q represents either an existential or universal quantifier with its variable and M represents the matrix, a prenex normal formula has the following general form:

$$Q_1 Q_2 Q_3 \ldots M$$

Consider first a formula that is not in prenex form because two quantifiers appear within a propositional subformula

$$\forall x(P(x) \wedge \forall y \exists x(Q(x,y) \vee R(x,y)))$$

Notice further that symbol x is used for two distinct variables: the inner disjunction has variable x bound by an existential quantifier and the variable in $P(x)$ is bound by the outer universal quantifier. Since the precise symbol given to a variable in a quantified formula is unimportant, one of the occurrences of x may be relabelled with symbol z to give

$$\forall x(P(x) \wedge \forall y \exists z(Q(z,y) \vee R(z,y)))$$

Quantifiers can then be moved to the left, producing a formula in prenex form:

$$\forall x \forall y \exists z(P(x) \wedge (Q(z,y) \vee R(z,y)))$$

Notice that, in moving the quantifiers to the left, we have assumed the two quantifiers can be "carried over" a predicate such as $P(x)$. This repositioning is justified by the following logical equivalences:

$$\begin{aligned} Qx(A \wedge B) &\cong A \wedge QxB \\ Qx(A \vee B) &\cong A \vee QxB \\ Qx(A \rightarrow B) &\cong A \rightarrow QxB \end{aligned}$$

Provided x does not occur free in A, the quantifier can be carried over this formula to the left, increasing its scope. Similar logical equivalences occur when the quantifier occurs to the left of a conjunction or disjunction symbol:

$$\begin{aligned} Qx(A \wedge B) &\cong QxA \wedge B \\ Qx(A \vee B) &\cong QxA \vee B \end{aligned}$$

Again the scope of a quantifier is extended to the whole formula, so variable x should not occur free in B. Provided the movement of a quantifier is not allowed to capture a free variable on moving to the left, the change to prenex form is straightforward for conjunctions and disjunctions.

Slightly more care is necessary when a quantification on the antecedent of an implication is to be applied to the whole formula rather than just the antecedent. In this case the quantifier changes from existential to universal, or vice versa, as its scope changes:

$$\begin{aligned} \forall x(A \rightarrow B) &\cong \exists xA \rightarrow B \\ \exists x(A \rightarrow B) &\cong \forall xA \rightarrow B \end{aligned}$$

Confirmation of these identities is easily obtained by replacing implications with disjunctions before increasing the scope of the quantifier:

$$\begin{aligned} \exists xA \rightarrow B &\cong \neg \exists xA \vee B \\ &\cong \forall x(\neg A \vee B) \\ &\cong \forall x(A \rightarrow B) \end{aligned}$$

Nevertheless, formulas with implications require greater care than those with only conjunctions and disjunctions. Consider the following conversion of formula $\forall xP(x,y) \rightarrow \neg \exists yQ(y)$ to prenex form:

$\forall x P(x,y) \rightarrow \neg\exists y Q(y)$
$\forall x P(x,y) \rightarrow \neg\exists z Q(z)$ rename variable
$\forall x P(x,y) \rightarrow \forall z \neg Q(z)$ move negation inwards
$\forall z(\forall x P(x,y) \rightarrow \neg Q(z))$ extract consequent quantifier
$\forall z \exists x(P(x,y) \rightarrow \neg Q(z))$ extract antecedent quantifier

2.6.2 Normal forms of writing proofs

A formula expressed in prenex normal form is equivalent to the formula from which it was derived, but has some useful features for a task in which it is to be used. Similarly, a sequent proof has normal forms equivalent to an unnormalised proof, but the application of inferences in the normal form illustrates features not obvious in the initial proof. In earlier examples, quantifier inferences or propositional inferences were used in a sequent proof in the order that each one appeared in the decomposition of the formulas. A Gentzen normal form of proof requires that all quantifier inferences are applied before any propositional inferences are used, dividing the proof into two separate parts above and below a line called the midsequent. Working down from the root formula, the midsequent is the first sequent that does not contain any quantifier symbols. Such a normal form of proof has the general appearance shown in Figure 2.8: a formula to be proven occurs at the top of a deduction tree followed by a number of quantifier inferences, leading at some point to the midsequent. Below this, only propositional inference rules are applied until axioms are obtained.

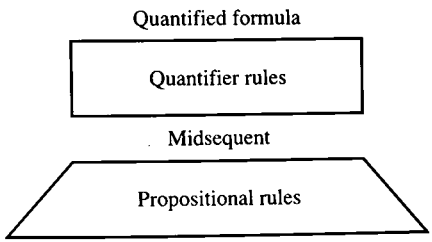

Figure 2.8 Midsequent proof pattern

Propositional and quantifier inferences have so far been applied in the order in which they arise. Applications of the two sorts of rules have been interleaved and the first inference rule to be applied has often been a propositional one. If we wish to apply every quantifier inference before any propositional rule, some modification of the procedures described in previous sections is required. One possible solution is to express all formulas in prenex normal form so that quantifier inferences are naturally the first to arise.

The following formula has already been validated by the Gentzen proof of Figure 2.6:

$$\exists x(P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x))$$

Using the identities given earlier, the formula is converted to prenex normal form:

$$\begin{array}{ll}
\exists x(P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x)) & \\
\forall x(P(x) \wedge Q(x)) \rightarrow (\exists y P(y) \wedge \exists z Q(z)) & \text{rename variables} \\
\forall x((P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x))) & \text{extract quantifier} \\
\forall x \exists y \exists z((P(x) \wedge Q(x)) \rightarrow (P(y) \wedge Q(z))) & \text{extract quantifier}
\end{array}$$

and a proof tree for the modified formula is shown in Figure 2.9. Since the prenex formula is equivalent to the original one, the fact that it is proven valid should not be a great surprise. The real point to be observed is that quantifier inferences are now applied first, leading to the midsequent line, then propositional rules are used to produce axioms. Only "right" quantifier rules are required in the proof and this will be the case in all proofs arising from a prenex form succedent. There can be no "right" negation or propositional rule that could move quantifiers over to the left because these connectives have been moved into the matrix. If the matrix had additionally been converted to negation normal form only, "right" propositional rules would be required.

As a larger example, we take the two formulas already in prenex form:

$$\forall x \forall y \forall z(P(x,y) \wedge P(y,z) \rightarrow P(x,z)) \qquad \forall x(\neg P(x,x))$$

and show that the prenex formula

$$\forall x \forall y(P(x,y) \rightarrow \neg P(y,x))$$

is a consequence. This might be achieved by using the semantic entailment arguments from the previous section or by building a semantic tableau from the first two formulas and a negation of the third. Equivalently, we might show that a Gentzen proof taking the first two formulas in its antecedent and the entailed formula as succedent leads to a proof tree. Figure 2.10 shows that the entailment does in fact hold and that a midsequent again divides quantifier inferences from propositional inferences.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow \forall x \exists y \exists z(P(x) \wedge Q(x) \rightarrow P(y) \wedge Q(z))}}{\Rightarrow \exists y \exists z(P(a) \wedge Q(a) \rightarrow P(x) \wedge Q(z))}\ \text{right } \forall \text{ on } 1}{\Rightarrow \exists z(P(a) \wedge Q(a) \rightarrow P(a) \wedge Q(z))}\ \text{right } \exists \text{ on } 2}{\Rightarrow P(a) \wedge Q(a) \rightarrow P(a) \wedge Q(a)}\ \text{right } \exists \text{ on } 3}{P(a) \wedge Q(a) \Rightarrow P(a) \wedge Q(a)}\ \text{right } \rightarrow \text{ on } 4}{P(a), Q(a) \Rightarrow P(a) \wedge Q(a)}\ \text{left } \wedge \text{ on } 5}{P(a), Q(a) \Rightarrow P(a) \quad \times \qquad P(a), Q(a) \Rightarrow Q(a) \quad \times}\ \text{right } \wedge \text{ on } 6$$

midsequent

Figure 2.9 Proof of a prenex formula

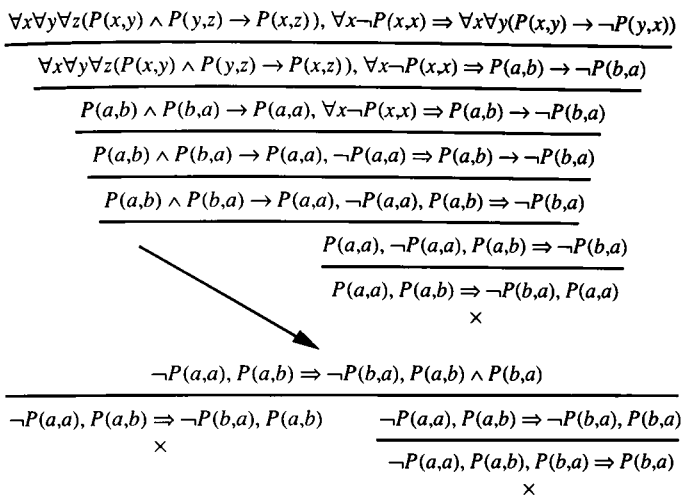

Figure 2.10 A deduction from prenex formulas

2.6.3 Negation normal form

A formula is said to be in negation normal form (NNF) if its logical symbols are restricted to the set $\{\exists, \forall, \wedge, \vee, \neg, \bot, (,)\}$ and every occurrence of the symbol $\neg$ stands directly before an atomic formula. Thus, the formula $\neg\forall x P(x,y)$ is not in NNF, but the equivalent form $\exists x \neg P(x,y)$ has the desired property. In fact, every first-order formula can be expressed in an equivalent negation normal form and the procedure for achieving this is an extension of that described in Chapter 1.

a. Use equivalences to replace unacceptable symbols by those in the above set. In practice this usually means the replacement of implication and mutual implication symbols.

b. Use further equivalences to drive inwards all $\neg$ symbols that do not stand directly before atomic formulas.

As a first example, we convert the formula

$$\exists x(P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x))$$

to negation normal form as follows:

$\exists x(P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x))$	
$\neg(\exists x(P(x) \wedge Q(x)) \vee (\exists x P(x) \wedge \exists x Q(x))$	propositional identity
$\forall x \neg(P(x) \wedge Q(x)) \vee (\exists x P(x) \wedge \exists x Q(x))$	quantifier identity
$\forall x(\neg P(x) \vee \neg Q(x)) \vee (\exists x P(x) \wedge \exists x Q(x))$	De Morgan

A formula F in NNF has a dual form that is no more than the negated formula $\neg F$ adjusted so that it too is in NNF. The simplest negation normal forms are atomic

formulas with or without preceding negation symbols such as $P(x)$, $\neg Q(x,y)$ and $\neg R(x,y,z)$. Such formulas are usually called literals and have dual forms that are obtained by adding or removing negation symbols, producing the dual forms $\neg P(x)$, $Q(x,y)$ and $R(x,y,z)$ from the examples above. Quantified formulas have dual forms that are obtained by the equivalences given earlier:

Formula	Dual
$\forall x P(x,y)$	$\exists x \neg P(x,y)$
$\exists x P(x,y)$	$\forall x \neg P(x,y)$
$\neg \forall x P(x,y)$	$\forall x P(x,y)$
$\neg \exists x P(x,y)$	$\exists x P(x,y)$

Dual forms are equivalent to the negated formulas and each one of the tabulated duals could have been obtained by simplification of the negated formula. The dual of formula $\forall x P(x,y)$ is obtained as follows:

$$\neg \forall x P(x,y) \equiv \exists x \neg P(x,y)$$

using the earlier identity.

The dual form of the NNF formula

$$\forall x(\neg P(x) \vee \neg Q(x)) \vee (\exists x P(x) \wedge \exists x Q(x))$$

is obtained by negation and simplification as follows:

$$\neg(\forall x(\neg P(x) \vee \neg Q(x)) \vee (\exists x P(x) \wedge \exists x Q(x)))$$
$$\neg \forall x(\neg P(x) \vee \neg Q(x)) \wedge \neg(\exists x P(x) \wedge \exists x Q(x))$$
$$\exists x(P(x) \wedge Q(x)) \wedge (\forall x \neg P(x) \vee \forall x \neg Q(x))$$

but this result might have been written by inspection of the previous formula: universal and existential symbols are exchanged, atomic formulas are replaced by their dual forms and disjunctions and conjunctions are exchanged.

A formula might be in prenex form without also being in NNF; this is certainly the case when a prenex formula contains implication connectives or if negation symbols have scope over more than one predicate. Conversely, a formula might be in NNF but not in prenex form. Given a formula in either of these forms, it should not be difficult to adjust the formula so that it is in both forms; the following prenex formula is modified to ensure it is also in NNF:

$$\forall x \exists y \exists z((P(x) \wedge Q(x)) \rightarrow (P(y) \wedge Q(z)))$$
$$\forall x \exists y \exists z(\neg(P(x) \wedge Q(x)) \vee (P(y) \wedge Q(z))) \quad \text{identity}$$
$$\forall x \exists y \exists z(\neg P(x) \vee \neg Q(x) \vee (P(y) \wedge Q(z))) \quad \text{De Morgan}$$

A formula in NNF is very easily adjusted to put it in prenex form. An NNF formula might have several quantified subformulas within a propositional statement and each subformula might use the same variable symbol. This layout appears in the following formula:

$\exists x(P(x) \vee Q(x)) \vee (\forall x \neg P(x) \vee \forall x \neg Q(x))$

but the variables could be renamed and the quantifiers then moved to the left to give a prenex form

$\exists x \forall y \forall z((P(x) \vee Q(x)) \vee (\neg P(y) \vee \neg Q(z))$

Later we shall see that there are sometimes advantages in retaining an NNF formula in non-prenex form.

2.6.4 Refutation normal form

A formula, or set of formulas, is proven valid by making it the succedent of a sequent and applying inference rules until a proof tree is obtained. Thus the starting-point of the proof is the sequent

$\Rightarrow$ formula

but the left negation rule allows this sequent to be identified with another sequent

$\neg$formula $\Rightarrow$

A given formula may be proven from either starting-point, but in the second case the first rule to be applied is the left negation inference and the sequent is returned to the succedent form. However, if the negated antecedent formula is converted to negation normal form, the sequent becomes

$(\neg\text{formula})_{\text{nnf}} \Rightarrow$

and applications of negation rules are delayed until the very last steps of the proof.

Earlier we saw how prenex formulas lead to special normal forms of proof in which all the applications of quantifier inference rules are discharged before any of the propositional inferences. As a result, a special sequent without any quantifier symbols called the midsequent appears to divide the two different kinds of inference rules. This is possible because the quantifier inferences are naturally the first to be used in a sequent containing only prenex formulas. A second method allows all quantifier inferences to be discharged before any propositional ones are used. Especially useful in refutation form, this second method allows quantifier inference rules to be applied to quantified subformulas within sequent formulas, provided the sequent formulas are expressed in negation normal form. The need to have formulas in NNF is clear if we consider the apparent need to apply a "left $\forall$" inference rule to the subformula $\forall x P(x)$ in the following sequent:

$Q(a,b) \vee \neg(P(b) \wedge \forall x P(x)) \Rightarrow$

but when this formula is converted to NNF it becomes

$Q(a,b) \vee \neg P(b) \vee \exists x \neg P(x) \Rightarrow$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists x(P(x) \wedge Q(x)) \wedge (\forall x \neg P(x) \vee \forall x \neg Q(x)) \Rightarrow}{(P(a) \wedge Q(a)) \wedge (\forall x \neg P(x) \vee \forall x \neg Q(x)) \Rightarrow}}{(P(a) \wedge Q(a)) \wedge (\neg P(a) \vee \forall x \neg Q(x)) \Rightarrow}}{\text{midsequent} \quad (P(a) \wedge Q(a)) \wedge (\neg P(a) \vee \neg Q(a)) \Rightarrow}}{P(a) \wedge Q(a), \neg P(a) \vee \neg Q(a) \Rightarrow}}{P(a), Q(a), \neg P(a) \vee \neg Q(a) \Rightarrow}$$

$$\dfrac{P(a), Q(a), \neg P(a) \Rightarrow}{P(a), Q(a) \Rightarrow P(a)} \qquad \dfrac{P(a), Q(a), \neg Q(a) \Rightarrow}{P(a), Q(a) \Rightarrow Q(a)}$$

$$\times \qquad \times$$

Figure 2.11 A refutation tree

and it is clear that what is really required is an application of the "left $\exists$" rule to give the subsequent

$$Q(a,b) \vee \neg P(b) \vee \neg P(c) \Rightarrow$$

Formulas in NNF only contain negation symbols within literals and we can be certain that in this case the subformula quantifiers really are what they appear to be.

Earlier it was shown that the formula

$$\exists x(P(x) \wedge Q(x)) \rightarrow (\exists x P(x) \wedge \exists x Q(x))$$

is valid by making it the succedent in the proof tree of Figure 2.6. This formula was converted to NNF as an example and its dual form was derived as

$$\exists x(P(x) \wedge Q(x)) \wedge (\forall x \neg P(x) \vee \forall x \neg Q(x))$$

If this formula is taken as the antecedent in a root sequent and quantifier rules are applied to quantified subformulas, the deduction tree in Figure 2.11 is obtained. All of the quantifier inferences can be applied before any of the propositional inferences; this is because subformulas can be instantiated when the formula is in NNF. The tree is divided by a line called the midsequent, above which there are only quantifier inferences and below which there are only propositional inferences. Every branch of this tree terminates with an axiom, indicating that an attempt to demonstrate the truth of the negated formula has failed. This in turn indicates that the negated formula is a contradiction and the original unnegated formula must be a tautology, i.e. it is a valid formula. Proofs of this kind are called refutations because they achieve their objective by refuting a dual, negated formula.

2.6.5 Skolem functions

Skolem functions may be used to replace existentially quantified variables in a formula such as

$\exists x(Woman(x) \wedge Loves(mike,x))$

simply by replacing the quantified variable by a specific though unknown object

$(Woman(a) \wedge Loves(mike,a))$

The claim that "there exists" such an object is replaced by a label for the object itself in a process resembling an application of the "left $\exists$" inference rule. Suppose, however, that we try to extend this reasoning to the following formal expression of "every man loves a woman":

$\forall x(Man(x) \rightarrow \exists y(Woman(y) \wedge Loves(x,y)))$

by instantiating a specific object in the place of the existentially quantified variable

$\forall x(Man(x) \rightarrow (Woman(a) \wedge Loves(x,a)))$

Unfortunately, constant a represents a specific though unknown woman, so the formula suggests that every man loves the same woman. The problem arises because the variable of the existential quantifier occurs within the scope of a universal quantifier and the simple constant a above does not take this into account. Every man loves a woman, but the woman may be different for each man and any substitution for the existentially quantified variable must reflect this. Skolem solved the problem by introducing a function $f(x)$ to represent an existentially quantified object within the scope of a universal quantifier. Function $f(x)$ is substituted in place of the simple constant a to give the formula

$\forall x(Man(x) \rightarrow (Woman(f(x)) \wedge Loves(x, f(x))))$

In this Skolemised form, $Man(a)$ loves $Woman(f(a))$, $Man(b)$ loves $Woman(f(b))$ and so forth, allowing the individual instances of men $a, b, c, \ldots$ to be mapped to distinct women $f(a), f(b), f(c), \ldots$.

Going still further, we formalise the well-known claim that every sailor loves a woman in every port he visits:

$\forall x(Sailor(x) \rightarrow \forall y(Port(x,y) \rightarrow (\exists z Woman(z) \wedge Loves(x,z))))$

Now the woman who is loved depends not only on the sailor but also on the port, i.e. it is a function of two variables and the Skolem function $f(x,y)$ used to replace the existentially quantified variable takes this into account:

$\forall x(Sailor(x) \rightarrow (\forall y Port(x,y) \rightarrow (Woman(f(x,y)) \wedge Loves(x, f(x,y)))))$

A simple rule emerges from these examples: the arity of a Skolem function used to replace an existentially quantified variable depends on the number of universally quantified variables within whose scope the existential variable is placed. An existentially quantified variable that is not within the scope of any universally quantified variable is replaced by a simple constant, an arity-zero function. Each additional universally quantified variable increases the arity of the Skolem function by one.

There is an implicit assumption that the formula being Skolemised appears in the antecedent of a sequent. Thus the sequent

$$\forall x \exists y P(x,y) \Rightarrow$$

is Skolemised to give

$$\forall x P(x, f(x)) \Rightarrow$$

However, the initial sequent might equally well be written in the following succedent form:

$$\Rightarrow \exists x \forall y \neg P(x,y)$$

and the Skolemised form could also be transformed to the succedent form

$$\Rightarrow \exists x \neg P(x, f(x))$$

The replacement of existential quantified variables in an antecedent is clearly equivalent to the replacement of universally quantified variables in a succedent. The procedure for antecedent formulas outlined above is now inverted: universally quantified variables are now replaced by functions of arity equal to the number of existential quantifiers within whose scope they are positioned. There is a dual system of Skolem functions corresponding to the dual forms of formulas described earlier. Skolem functions applied to succedent formulas are sometimes called Herbrand functions because they were used by Herbrand in the theorem described below.

The following example shows the care necessary in inserting Herbrand or Skolem functions in a formula:

$$\Rightarrow \forall x(\exists y P(x,y) \rightarrow \forall z Q(x,z))$$

Here the inner universally quantified variable might appear to be within the scope of an existential quantifier, but the scope of the y variable is limited to the first subformula. As a result, both universally quantified variables are replaced by constants

$$\Rightarrow (\exists y P(a,y) \rightarrow Q(a,b))$$

The following sequent formula requires the use of arity-one and arity-two Skolem functions:

$$\Rightarrow \exists w \forall x \exists y \forall z((\neg P(w,x) \vee Q(w)) \rightarrow R(y,z))$$

because variable x is in the scope of $\exists w$ and variable z is within the scope of both $\exists w$ and $\exists y$. This gives the formula

$$\Rightarrow \exists w \exists y((\neg P(w, f(w)) \vee Q(w)) \rightarrow R(y, g(w,y)))$$

2.6.6 Herbrand's theorem

A sequent containing only prenex formulas can be rearranged to place all such formulas in the succedent of a sequent:

$$\Rightarrow \text{formulas}$$

and the first inferences to be applied will then be the right existential and universal rules. Herbrand's theorem states that a quantified formula is provable if and only if the quantifier-free formula obtained by the application of quantifier inference rules is provable. In terms of normal form proofs, the initial sequent is provable if and only if its midsequent is provable. New constants are introduced into succedent formulas by each application of the right universal rule, then existential inferences are free to reuse any such constants. As a proof proceeds towards its midsequent, the order in which these introductions occurred is lost, so the form of the original formula is lost. Herbrand compensated for this loss of information by instantiating Skolem functions rather than simple constants, thus recording the order in which introductions are made. In the simplest case, represented by the sequent

$$\Rightarrow \forall x \exists y \exists z((P(x) \wedge Q(x)) \rightarrow (P(y) \wedge Q(z)))$$

the replacement of an existentially quantified variable with a constant produces a result identical to that obtained from the right universal rule:

$$\Rightarrow \exists y \exists z((P(a) \wedge Q(a)) \rightarrow (P(y) \wedge Q(z)))$$

Only functions with an arity of one or more have a significant effect in the Herbrand proof procedure and the following sequent is therefore of more interest:

$$\Rightarrow \exists t \forall u \neg P(t,u),\ \exists v \forall w \neg Q(v,w),\ \forall x \exists y \exists z(P(x,y) \wedge Q(y,z))$$

A deduction tree for this formula is constructed as far as the midsequent in Figure 2.12a, then propositional inferences may be used to produce a proof tree. This sequent may be Skolemised to give

$$\Rightarrow \exists t \neg P(t,f(t)),\ \exists v \neg Q(v,g(v)),\ \exists y \exists z(P(a,y) \wedge Q(y,z))$$

and a proof tree tree derived from the Skolemised form is shown in Figure 2.12b.

Axioms can be produced from the midsequent of Figure 2.12b using the same two propositional steps required to produce axioms from the midsequent in Figure 2.12a. The important difference between the two approaches is that the original root formula can be reconstructed from the Skolemised midsequent, allowing the direct connection between a formula and its midsequent required for Herbrand's theorem.

2.6.7 Skolem–Herbrand–Gödel theory

Herbrand's theorem asserts a claim that the provability of a formula rests on the provability of a quantifier-free formula derived from the quantified form. An alternative approach called the Skolem–Herbrand–Gödel (SHG) theory makes a similar claim for the semantic concept of unsatisfiablity. According to this theory, a formula is unsatisfiable if and only if a quantifier-free formula derived by the application of quantifier rules is itself unsatisfiable. A formula may be shown to be valid through the refutation of its negation, i.e. by the failure of a systematic attempt to

$$\Rightarrow \exists t\forall u\neg P(t,u), \exists v\forall w\neg Q(v,w), \forall x\exists y\exists z(P(x,y) \wedge Q(y,z))$$
$$\Rightarrow \exists t\forall u\neg P(t,u), \exists v\forall w\neg Q(v,w), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \forall u\neg P(a,u), \exists v\forall w\neg Q(v,w), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,b), \exists v\forall w\neg Q(v,w), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,b), \forall w\neg Q(b,w), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,b), \neg Q(b,c), \exists y\exists z(P(a,y) \wedge Q(y,z))$$

(a) $$\Rightarrow \neg P(a,b), \neg Q(b,c), (P(a,b) \wedge Q(b,c))$$

$$\Rightarrow \exists t\neg P(t,f(t)), \exists v\neg Q(v,g(v)), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,f(a)), \exists v\neg Q(v,g(v)), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,f(a)), \neg Q(f(a),g(f(a))), \exists y\exists z(P(a,y) \wedge Q(y,z))$$
$$\Rightarrow \neg P(a,f(a)), \neg Q(f(a),g(f(a))), \exists z(P(a,f(a)) \wedge Q(f(a),z)$$

(b) $$\Rightarrow \neg P(a,f(a)), \neg Q(f(a),g(f(a))), P(a,f(a)) \wedge Q(f(a),g(f(a)))$$

Figure 2.12 Midsequents: (a) with simple constants and (b) with Skolem functions

satisfy that formula. However, as explained in Chapter 1, an attempt to produce a deduction tree from the succedent

$$\neg(\text{formula}) \Rightarrow$$

would quickly reproduce the proof form. But if the antecedent is converted to NNF, the left subformula quantifier rules can be directly applied to the formula. Like Herbrand's original theorm, the SHG theorem depends on the separation of quantifier and propositional rules in the deduction tree; but if the antecedent is in NNF, only left rules are required. The SHG theorem uses Skolem functions to replace existential quantifiers in a dual approach to that taken in the Herbrand theorem. It is only of real interest when existentially quantified variables lie in the scope of universal quantifiers, generating Skolem functions of arity one or more. Consider as an example the formula

$$\exists x((P(x) \vee \exists y Q(x,y)) \rightarrow (\exists z Q(x,z) \vee P(a)))$$

from which an equivalent NNF is derived as follows:

$$\exists x((P(x) \vee \exists y Q(x,y)) \rightarrow (\exists z Q(x,z) \vee P(a)))$$
$$\exists x(\neg(P(x) \vee \exists y Q(x,y)) \vee (\exists z Q(x,z) \vee P(a))) \quad \text{remove} \rightarrow$$
$$\exists x(\neg P(x) \wedge \forall y\neg Q(x,y)) \vee (\exists z Q(x,z) \vee P(a)) \quad \text{De Morgan}$$

Since we wish to work in refutation form, we derive the dual of the above NNF by inspection:

$$\forall x(P(x) \vee \exists y Q(x,y)) \wedge (\forall z\neg Q(x,z) \wedge \neg P(a))$$

$$\frac{\forall x(P(x) \vee \exists y Q(x,y)) \wedge (\forall z \neg Q(x,z) \wedge \neg P(a)) \Rightarrow}{(P(a) \vee \exists y Q(a,y)) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}$$

$$\frac{(P(a) \vee \exists y Q(a,y)) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}{(P(a) \vee Q(a,b)) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}$$

$$\text{(a)} \qquad \frac{(P(a) \vee Q(a,b)) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}{(P(a) \vee Q(a,b)) \wedge (\neg Q(a,b) \wedge \neg P(a)) \Rightarrow}$$

$$\frac{\forall x(P(x) \vee Q(x,f(x))) \wedge (\forall z \neg Q(x,z) \wedge \neg P(a)) \Rightarrow}{(P(a) \vee Q(a,f(a))) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}$$

$$\text{(b)} \qquad \frac{(P(a) \vee Q(a,f(a))) \wedge (\forall z \neg Q(a,z) \wedge \neg P(a)) \Rightarrow}{(P(a) \vee Q(a,f(a))) \wedge (\neg Q(a,f(a)) \wedge \neg P(a)) \Rightarrow}$$

Figure 2.13 Midsequents: (a) with simple constants and (b) with Skolem functions

and the deduction of a midsequent from this formula as an antecedent is shown in Figure 2.13a. When the formula above is Skolemised, we obtain

$$\forall x(P(x) \vee Q(x,f(x))) \wedge (\forall z \neg Q(x,z) \wedge \neg P(a))$$

and a deduction tree produced when this formula is taken as the initial antecedent is shown in Figure 2.13b. Both of these deduction trees terminate in refutations, but the Skolemised version allows the original formula to be reconstructed.

EXERCISES 2.6

1. Convert the following formulas to prenex normal form:
   a. $\exists x(P(x) \vee \forall y(Q(x,y) \wedge \forall x R(x)))$
   b. $\forall x(\forall y R(x,y) \wedge \exists y(S(x,y) \vee \forall x T(x)))$
2. Express the following valid formulas in equivalent prenex normal form:
   a. $\forall x(P(x) \wedge Q(x)) \rightarrow \forall x P(x) \vee \forall x Q(x)$
   b. $\forall x P(x) \wedge \forall x Q(x) \rightarrow \forall x(P(x) \vee Q(x))$
   c. $\forall x(P(x) \wedge Q(x)) \wedge \forall x P(x) \rightarrow \forall x Q(x)$
   d. $\forall x(P(x) \rightarrow \neg Q(x)) \rightarrow (\exists x(R(x) \wedge Q(x)) \rightarrow \exists x(R(x) \wedge \neg P(x)))$
3. Show normal form proof trees for each of the formulas derived in the previous exercise, indicating the midsequent in each tree.
4. Convert each of the formulas of Exercise 2 into equivalent negation normal forms.
5. Take the dual forms of each of the formulas resulting from Exercise 4 and produce a refutation tree for each one.
6. Prove the following sequent is valid:

   $$\forall x \exists y \forall z R(x,y,z) \Rightarrow \forall x \forall z \exists y R(x,y,z)$$

Skolemise both sides of the sequent and repeat the proof.

7. Convert each of the following formulas to NNF then Skolemise out any existentially quantified variables:

   a. $\forall x \exists x P(x,y)$

   b. $\forall x \forall y \exists z R(x,y,z) \wedge \forall x \exists y \forall z R(x,y,z)$

   c. $\forall x \exists y (P(x,y) \rightarrow \exists z Q(x,y,z))$

   d. $\exists x (\forall y P(x,y) \rightarrow \exists z Q(x,z))$

   e. $\forall x \exists y \forall z \exists w ((\neg P(x,y) \vee Q(x)) \rightarrow R(x,z))$

2.7 A HILBERT PROOF SYSTEM FOR FORMULAS

The Hilbert proof system described in Chapter 1 is expanded to first-order logic in each of the four components described earlier:

a. An alphabet consisting of the following logical symbols:

   $\neg, \rightarrow, \forall, (,),$

   and the following non-logical symbols:

$a, b, c, \ldots$	constants
$x, y, z \ldots$	variables
$f, g, h,$	function symbols
$P, Q, R, \ldots$	atomic formulas

b. Rules for building formulas from the alphabet:

   1. Every atomic formula is a formula.
   2. If A and B are formulas then so are $\neg A$, $A \rightarrow B$ and $\forall x A$ where x is any variable.
   3. Nothing else is a formula.

c. Five axioms, three of which are inherited from the propositional subset.

   1. $(A \rightarrow (B \rightarrow A))$
   2. $((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$
   3. $((\neg A) \rightarrow (\neg B)) \rightarrow (B \rightarrow A)$
   4. $\forall x A(x) \rightarrow A(a)$
   5. $\forall x (A \rightarrow B(x)) \rightarrow (A \rightarrow \forall x B(x))$

d. Two rules of deduction:

   1. *Modus ponens* (MP): from A and $(A \rightarrow B)$ deduce B.
   2. Generalisation: if A is a formula and x is any variable from A, deduce $\forall x A$.

It is possible to recognise a subset of symbols, rules and axioms identical to those defined for the propositional Hilbert proof system within the proof system described above. The previous alphabet is extended with terms, formulas and quantifier symbols to allow the construction of full first-order logic formulas. Notice, however, that the logical symbols available in the alphabet are restricted to a minimal set and that only a universal quantifier is defined. A Hilbert proof system may be defined with a larger alphabet, but more axioms are then required to encode the required properties into the system. In practice a limited set of symbols is not a major problem because the missing symbols can be introduced as abbreviations for formulas expressed within the above system. For example

$A \vee B$ abbreviates $\neg A \rightarrow B$
$A \wedge B$ abbreviates $\neg(A \rightarrow \neg B)$

Similarly, the existential expression $\exists x A x$ may be seen as an abbreviated method of writing a formula $\neg\forall x \neg A x$, using only symbols from the alphabet above. This is justified by the equivalences described earlier. The two additional axioms are, like the previous three, really axiom schemata in which metasymbols A and B represent any formula.

A Hilbert proof begins with a number of assumptions to which axioms and rules of deduction are applied until a formula of some interest is obtained. The fact that the resulting formula is derived from those assumptions using the Hilbert axioms and rules of deduction is then expressed by the syntactic turnstile:

assumptions $\vdash_{\mathrm{H}}$ formula

2.7.1 The deduction rule

The rule of generalisation may be expressed in a Hilbert deduction as

$A \vdash_{\mathrm{H}} \forall x A(x)$

but it does not generally follow that

$\vdash_{\mathrm{H}} A \rightarrow \forall x A(x)$

so the deduction theorem cannot be applied as simply as in the formal system of propositions. The use of the deduction theorem as above is only possible if there is no application of generalisation to a variable that occurs free in A. Certainly, if A is a closed formula, this problem does not arise.

A small proof using quantifier rules shows how formula $\forall x(A \rightarrow B(x))$ is proven from the assumption $A \rightarrow \forall x B(x)$:

1. $A \rightarrow \forall x B(x)$ assumption
2. $\forall x B(x) \rightarrow B(a)$ axiom 4
3. $A \rightarrow B(a)$ 1, 2 chain
4. $\forall x(A \rightarrow B(x))$ generalisation of 3

This deduction is then expressed through the turnstile:

$$A \rightarrow \forall x B(x) \vdash_H \forall x(A \rightarrow B(x))$$

and since x does not occur free in A, the deduction theorem can be applied to give

$$\vdash_H (A \rightarrow \forall x B(x)) \rightarrow \forall x(A \rightarrow B(x))$$

A second example seeks to prove the formula

$$\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$$

by noting that two applications of the deduction rule might produce such a form. For this reason, the two antecedents make suitable assumptions from which the right-hand subformula might be derived as follows:

1. $\forall x(P(x) \rightarrow Q(x))$	assumption
2. $\forall x P(x)$	assumption
3. $\forall x P(x) \rightarrow P(a)$	axiom 4
4. $P(a)$	2, 3 *modus ponens*
5. $\forall x(P(x) \rightarrow Q(x)) \rightarrow (P(a) \rightarrow Q(a))$	axiom 4
6. $P(a) \rightarrow Q(a)$	1, 5 *modus ponens*
7. $Q(a)$	4, 6 *modus ponens*
8. $\forall x Q(x)$	generalisation of 7

The deduction is then expressed in turnstile form:

$$(\forall x(P(x) \rightarrow Q(x))), \forall x P(x) \vdash_H \forall x Q(x)$$

then two applications of the deduction theorem give the desired result:

$$(\forall x(P(x) \rightarrow Q(x))) \vdash_H \forall x P(x) \rightarrow \forall x Q(x)$$
$$\vdash_H (\forall x(P(x) \rightarrow Q(x))) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$$

Existential quantifiers are so useful in practice that it is useful to have an existential equivalent of the generalisation property. Such a formula would appear as the theorem

$$\vdash_H P(a) \rightarrow \exists x P(x)$$

and this is easily proven:

1. $\forall x \neg P(x) \rightarrow \neg P(a)$	axiom 4
2. $(\forall x \neg P(x) \rightarrow \neg P(a)) \rightarrow (P(a) \rightarrow \neg \forall x \neg P(x))$	contrapositive
3. $P(a) \rightarrow \neg \forall x \neg P(x)$	1, 2 *modus ponens*
4. $P(a) \rightarrow \exists x P(x)$	definition

The strategy is to prove the deduction in terms of fragments such as $\neg \forall x \neg P(x)$ then to replace such fragments with the equivalent existential form. Such a strategy has to be adopted in proving the theorem

$$\vdash_H \forall x(P(x) \rightarrow Q(x)) \rightarrow (\exists x P(x) \rightarrow \exists x Q(x))$$

The two antecedents again provide some guidance on the assumptions to be used in proving the formula, but the reason for the second assumption is only clear at the end of the proof:

1. $\forall x(P(x) \rightarrow Q(x))$	assumption
2. $\forall x \neg Q(x)$	assumption
3. $\forall x(P(x) \rightarrow Q(x)) \rightarrow (P(a) \rightarrow Q(a))$	axiom 4
4. $P(a) \rightarrow Q(a)$	1, 3 *modus ponens*
5. $(P(a) \rightarrow Q(a)) \rightarrow (\neg Q(a) \rightarrow \neg P(a))$	contrapositive
6. $\neg Q(a) \rightarrow \neg P(a)$	4, 5 *modus ponens*
7. $\forall x \neg Q(x) \rightarrow \neg Q(a)$	axiom 4
8. $\neg Q(a)$	2, 7 *modus ponens*
9. $\neg P(a)$	6, 8 *modus ponens*
10. $\forall x \neg P(x)$	generalisation of 9

confirming the following Hilbert deduction:

$$\forall x(P(x) \rightarrow Q(x)), \forall x \neg Q(x) \vdash_{\mathrm{H}} \forall x \neg P(x)$$

from which a final result may be obtained after two applications of the deduction theorem:

$$\forall x(P(x) \rightarrow Q(x)) \vdash_{\mathrm{H}} \forall x \neg Q(x) \rightarrow \forall x \neg P(x) \quad \text{deduc}$$
$$\vdash_{\mathrm{H}} (\forall x(P(x) \rightarrow Q(x))) \rightarrow (\forall x \neg Q(x) \rightarrow \forall x \neg P(x)) \quad \text{deduc}$$

This is not quite the formula required, but the replacement of one subformula by its contrapositive, followed by a simplification, yields the desired result:

$$\vdash_{\mathrm{H}} (\forall x(P(x) \rightarrow Q(x))) \rightarrow (\neg \forall x \neg P(x) \rightarrow \neg \forall x \neg Q(x))$$
$$\vdash_{\mathrm{H}} (\forall x(P(x) \rightarrow Q(x))) \rightarrow (\exists x P(x) \rightarrow \exists x Q(x))$$

The formulation of proofs in a Hilbert system requires much more experience than the development of a proof for the same formula in the Gentzen G system described earlier. A G system proof arises from the decomposition of a formula whereas a Hilbert proof requires some foresight or experiment to arrive at suitable starting asumptions.

2.7.2 Soundness and completeness

It can be shown that any formula proven in Hilbert's system is valid and the proof system is therefore sound. Thus if M is some set of formulas and F is a formula proven in the deduction system, then

$$M \vdash_{\mathrm{H}} F \text{ implies } M \models F$$

Conversely, Gödel's completeness theorem assures us that any valid theorem is provable in the calculus, i.e.

$$M \models F \text{ implies } M \vdash_{\mathrm{H}} F$$

Since any valid formula may be proven and anything proven is valid, it might seem that any formula at all may be either proved or disproved. However, soundness and completeness are defined only with respect to valid formulas; they say nothing about an invalid formula. The real problem is that, unlike propositional logic, first-order logic is not decidable, i.e. there is no algorithm that can decide whether or not a formula is valid.

During the early part of the twentieth century, many researchers struggled to find algorithms that would decide the validity of first-order formulas, but none succeeded. Eventually in the mid 1930s Turing and Church separately showed that no such algorithm could ever be developed. Fortunately, this negative result was accompanied by a positive one: Church defined the limit of what could be computed in terms of partial recursive functions. The Church–Turing thesis tells us that first-order logic is partially decidable and that the parts that may be decided are formulated as partial recursive functions. This group of computable functions includes some expressions that are in principle computable, but in practice have exponential complexities, so they rapidly become intractable. A subgroup of primitive recursive functions contains all those computable functions that are needed in practice, and this subgroup is exactly what is required to produce the midsequent in Herbrand's theorem. Church originally formulated his arguments in a notation called lambda calculus, but this is equivalent to the (initial) semantics of terms in first-order logic. More important, this whole area of work gave rise to the functional programming languages described later in this book.

EXERCISES 2.7

1. Use Hilbert axioms to prove the following formulas:
   a. $\forall x P(x) \rightarrow \exists x P(x)$
   b. $\forall x P(x) \rightarrow \forall y P(y)$
   c. $\neg \forall x P(x) \rightarrow \exists y \neg P(y)$
   d. $\forall x(P(x) \wedge Q(x)) \rightarrow \forall x P(x) \wedge \forall x Q(x)$
   e. $\forall x \forall y R(x,y) \rightarrow \forall x R(x,x)$
   f. $\forall x \forall y(Q(x,y) \rightarrow \neg Q(y,x)) \rightarrow \forall x \neg Q(x,x)$

CHAPTER THREE

Principles of logic programming

3.1 REFUTING PROPOSITIONS

G system proofs may be constructed in either of the two normal forms described earlier: proof normal form and refutation normal form. The first of these approaches places formulas in the succedent position then endeavours to falsify the formula and thus the sequent. A deduction tree in which each branch terminates with an axiom, a valid sequent, is sufficient to prove the original formula valid. Refutation normal form, on the other hand, places a negation normal form of the negated formula in the antecedent then systematically attempts to satisfy the formula and thus the sequent. A deduction tree in which each branch terminates with an axiom now indicates contradiction and indirectly shows the validity of the original unnegated proposition.

In order to show the relationship between the normal forms described earlier and the resolution technique described in this chapter, we consider again the proposition

$$(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$$

which we already know to be a tautology from its semantic tableau given in Figure 1.8. Using the method of substituting equivalences described in Chapter 1, an NNF of the above formula is derived as follows:

$(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$	
$\neg((\neg p \vee q) \wedge (\neg q \vee r)) \vee (\neg p \vee r)$	remove implications
$\neg(\neg p \vee q) \vee \neg(\neg q \vee r) \vee (\neg p \vee r)$	De Morgan
$(p \wedge \neg q) \vee (q \wedge \neg r) \vee \neg p \vee r$	De Morgan

In fact, this result is also in DNF and, since the original formula is a tautology, a deduction tree taking either the original formula or its DNF equivalent as the initial succedent would produce only leaf axioms and is a proof tree.

$$\dfrac{\dfrac{\dfrac{(\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r \Rightarrow}{(\neg p \vee q), (\neg q \vee r), p, \neg r \Rightarrow}\ \text{gen left } \wedge}{(\neg p \vee q), \neg q, p, \neg r \Rightarrow}\ \text{left } \vee \text{ on 2} \qquad \dfrac{}{(\neg p \vee q), r, p, \neg r \Rightarrow}\ \text{left } \vee \text{ on 2}}{\neg p, \neg q, p, \neg r \Rightarrow} \qquad \dfrac{}{q, \neg q, p, \neg r \Rightarrow}\ \text{left } \vee \text{ on 3}$$

Figure 3.1 A refutation deduction

Proof normal form does not in fact require the proposition itself to be in any special form, but we saw earlier that an especially interesting case arises when the succedent formula is in DNF, i.e. has the form

$$\Rightarrow \text{formula}_{\text{dnf}}$$

In this case all disjunctions may be removed in a general "right $\vee$" rule, leaving only "right $\wedge$" rules to be applied. Refutation normal form requires the negated antecedent formula to be in NNF, but again an especially interesting property was observed when the formula was also in CNF:

$$(\neg\text{formula})_{\text{cnf}} \Rightarrow$$

Now all the conjunctions may be removed in a general "left $\wedge$" inference, leaving only "left $\vee$" and "left $\neg$" rules to be applied. Since we are more interested here in refutation normal form, an NNF of the negated formula is required, but this is easily obtained as the dual of the previous proposition:

$$(\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r$$

Remembering this formula is the negation of one that we wish to prove valid, it is made the antecedent of a sequent which then produces the refutation-style deduction shown in Figure 3.1. All the "left $\wedge$" inferences have been carried out in the first step, leaving an antecedent containing clauses that are then subjected to the "left $\vee$" rule. Subsequents with dual literals in their antecedents are eventually obtained and the "left $\neg$" rule might be applied to convert each of them to an axiom. Alternatively, we might simply accept that an antecedent containing these clashing pairs is equivalent to an axiom and stop the tree at this point. Although the form of the deduction tree is illustrated by a particular example, it should be clear that a deduction tree whose root is a CNF antecedent always follows this form. A generalised "left $\wedge$" inference followed by several "left $\vee$" produces a tree of leaf axioms if the root formula is inconsistent. Since the initial "left $\wedge$" and final "left $\neg$" operations are little more than formatting procedures, the only effective operation in these deductions is the "left $\vee$" inference. Thus the refutation is achieved with only one rule in a way similar to the resolution procedure described later.

CNF formulas have a particularly simple structure that admits a simplified notation called clausal form in which logical connectives are not shown. For example, the CNF formula above is represented by the clausal form

$$\{\neg p, q\}, \{\neg q, r\}, \{p\}, \{\neg r\}$$

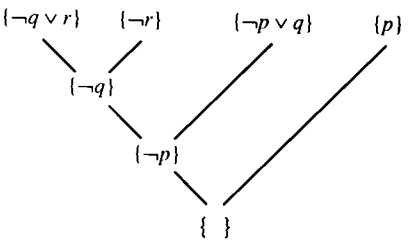

Figure 3.2 Resolution to produce an empty clause

with the understanding that literals within set brackets are joined by disjunctions whereas sets themselves are joined by conjunctions. A resolution proof shows this proposition to be inconsistent by removing clashing pairs of literals from clauses until an empty clause is obtained. Figure 3.2 shows how the dual pair of literals clashing in clauses $\{\neg q, r\}$ and $\{\neg r\}$ is removed to produce a new clause $\{\neg q\}$ that undergoes further resolution. Obtaining an empty clause through this procedure is equivalent to obtaining a tree containing only axioms in a G system refutation. In fact, the three clashing pairs based on r, q and p in Figure 3.2 are quickly related to the three axioms based on the same symbols in Figure 3.1.

In summary, the method of resolution proceeds as follows:

a. Convert the negated formula to CNF.

b. Rewrite the result in clausal form.

c. Apply the resolution step, i.e. from $\{A, X\}$ and $\{B, \neg X\}$ deduce $\{A, B\}$ until an empty clause is obtained.

The three Hilbert propositional axioms are certainly tautologies and this should be easily demonstrated with the resolution procedure. An instance of the first axiom is converted to NNF as follows:

$$\begin{array}{ll} p \rightarrow (q \rightarrow p) & \\ \neg p \vee (\neg q \vee p) & \text{equivalences} \\ \neg p \vee \neg q \vee p & \text{remove brackets} \end{array}$$

producing a DNF in which three unit cubes are joined by disjunctions. Notice that in this case the resulting proposition is also in CNF because the resulting proposition may be seen as a single clause. Negation then produces the dual of this proposition

$$p \wedge q \wedge \neg p$$

and when this is shown in clausal form as

$$\{p\}, \{q\}, \{\neg p\}$$

the production of an empty clause is obvious. Although a very small example, this refutation is interesting because it achieves its result without using all of its clauses. One form of a logic law called the compactness theorem states that a clausal form

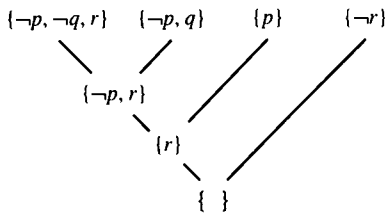

Figure 3.3 Refutation of the negated second axiom

is unsatisfiable if any subset of the clauses is unsatisfiable. Thus if any subproposition is unsatisfiable, the whole proposition is unsatisfiable.

Hilbert's second axiom has the form

$$(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$$

and the equivalent DNF of this proposition was derived in Section 1.6 as

$$(p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \vee \neg p \vee r$$

A negated form of this proposition is easily expressed through its dual, automatically in CNF:

$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p \wedge \neg r$$

which is expressed in clausal form as

$$\{\neg p, \neg q, r\}, \{\neg p, q\}\{p\}, \{\neg r\}$$

and a resolution diagram leading to an empty clause is shown in Figure 3.3.

Finally, Hilbert's third axiom $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$ has the negation normal form derived below:

$$\begin{array}{ll} (\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p) & \\ \neg(\neg\neg p \vee \neg q) \vee (\neg q \vee p) & \text{equivalences} \\ (\neg p \wedge q) \vee \neg q \vee p & \text{De Morgan} \end{array}$$

producing a result conveniently in DNF. The negation of this proposition is easily expressed in CNF by the following dual form:

$$(p \vee \neg q) \wedge q \wedge \neg p$$

and the clausal form $\{p, \neg q\}, \{q\}, \{\neg p\}$ has obvious clashing literals.

As one further example, consider the semantic tableau of the following proposition, already in CNF:

$$(p \vee \neg q) \wedge (\neg s \vee q) \wedge (\neg s \vee \neg q) \wedge s$$

Figure 3.4a shows that a semantic tableau built from this proposition closes before all of its clauses have been used. Similarly, the resolution diagram of Figure 3.4b produces an empty clause from the equivalent clausal form

$$\{p, \neg q\}, \{\neg s, q\}, \{\neg s, \neg q\}, \{s\}$$

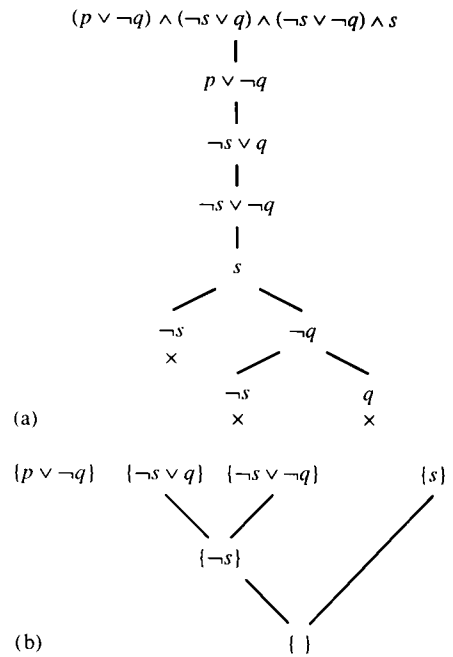

Figure 3.4 (a) Semantic tableau and (b) resolution of a negated formula

without using the clause $\{p, \neg q\}$. As noted above, any subset of clauses that produces the empty clause is sufficient to refute the whole formula, so the existence of $\{x\}$ and $\{\neg x\}$ in any clausal form is sufficient. When this occurs, the semantic tableau also closes without using all of its clauses.

A resolution step only depends on an implication from left to right:

$$((A \vee X) \wedge (B \vee \neg X)) \rightarrow (A \vee B)$$

and the validity of this formula is shown in the semantic tableau of Figure 3.5. The following proposition is, however, not valid:

$$((A \vee X) \wedge (B \vee \neg X)) \leftrightarrow (A \vee B)$$

Some valuations of the atoms in this mutual implication are a model for the formula, i.e. they make the overall formula *true*, but not all of them do so. It is a useful exercise to derive truth tables for the left- and right-hand sides of the mutual implication and to show that the two are not equivalent.

Figure 3.1 showed how a proposition in CNF is refuted when it is the antecedent of a root sequent in a deduction tree. The form of the example with its CNF antecedent is

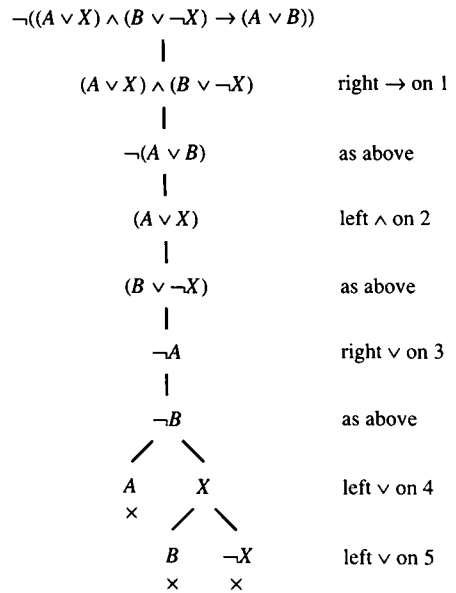

Figure 3.5 A tableau proof of the resolution principle

$$(\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r \Rightarrow$$

but we might be curious to know how the example would have proceeded if the antecedent had been in DNF rather than CNF. In order to find out, we have to find the equivalent formula, rather than the dual form, using algebraic techniques. Since the DNF is equivalent to the previous form, another refutation tree should result, but it is the form of this tree that is of interest. To convert CNF to DNF, the clauses are "multiplied out" rather like arithmetic formulas; for example, the first two clauses multiply out to give a product proposition

$$(\neg p \vee q) \wedge (\neg q \vee r) \equiv (\neg p \wedge \neg q) \vee (\neg p \wedge r) \vee (q \wedge \neg q) \vee (q \wedge r)$$

and this result is "multiplied" by proposition $p \wedge \neg r$:

$$\begin{aligned}&((\neg p \wedge \neg q) \vee (\neg p \wedge r) \vee (q \wedge \neg q) \vee (q \wedge r)) \wedge (p \wedge \neg r)\\ &\equiv (\neg p \wedge \neg q \wedge p \wedge \neg r) \vee (\neg p \wedge r \wedge p \wedge \neg r) \vee (q \wedge \neg q \wedge p \wedge \neg r)\\ &\quad \vee (q \wedge r \wedge p \wedge \neg r)\end{aligned}$$

A Gentzen style refutation taking the resulting proposition as antecedent immediately divides through a generalised "left ∨" into four subsequents of the form

$$\neg p \wedge \neg q \wedge p \wedge \neg r \Rightarrow$$

then a generalised "left ∧" applied to each subsequent produces sequents of the form

$$\neg p, \neg q, p, \neg r \Rightarrow$$

with a clashing pair of literals. Every cube in the DNF contains a pair of complementary literals that immediately produces an axiom, so a refutation of this kind always stops after the first step.

EXERCISES 3.1

1. Convert each of the following tautologies into disjunctive normal form then derive the dual to give a formula in conjunctive normal form:
   a. $(p \vee q) \rightarrow (q \vee p)$
   b. $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$
   c. $(p \rightarrow (q \rightarrow r)) \rightarrow (p \wedge r \rightarrow q)$
   d. $(p \rightarrow r) \wedge (\neg q \rightarrow \neg r) \rightarrow (p \rightarrow q)$
   e. $(p \rightarrow r) \wedge (q \rightarrow r) \rightarrow (p \vee q \rightarrow r)$
2. Produce refutation trees smilar to Figure 3.1 for each of the CNF propositions obtained in the previous exercise.
3. Express each of the CNF propositions from Exercise 1 in clausal form and show through resolution that the formula is unsatisfiable.
4. The converse of the resolution proposition is
   $$(A \vee B) \rightarrow (A \vee X) \wedge (B \vee \neg X)$$
   Show that this proposition is not valid, by tableau and by writing out the truth table.
5. Prove the following mutual equivalence by proving two separate implications in tableaux or G system proofs:
   $$((A \vee X) \wedge (B \vee \neg X)) \leftrightarrow (A \vee X) \wedge (B \vee \neg X) \wedge (A \vee B)$$

3.2 REFUTING FORMULAS

The previous section showed a close relationship between the refutation normal form of a G system deduction and resolution applied to the same proposition. Now we have to extend this connection to deductions involving the quantified formulas described in Chapter 2.

Figure 2.3 shows a semantic tableau for the negation of formula

$$\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$$

and it is clear from the tableau that every path is closed, the negated formula is a contradiction and the original formula is therefore valid. This formula might equally well have been made the succedent of a sequent and the same series of inference

rules applied as in the tableau construction. A proof tree in which every leaf is an axiom then shows the validity of the original formula. In the manner of the previous section, we now convert this formula to its equivalent negation normal form:

$$(\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x)))$$
$$\neg\forall x(\neg P(x) \vee Q(x)) \vee (\neg\forall x P(x) \vee \forall x Q(x)) \quad \text{identity}$$
$$\exists x\neg(\neg P(x) \vee Q(x)) \vee \exists x\neg P(x) \vee \forall x Q(x) \quad \text{identity}$$
$$\exists x(P(x) \wedge \neg Q(x)) \vee \exists x\neg P(x) \vee \forall x Q(x) \quad \text{De Morgan}$$

A G system proof taking the resulting NNF formula as its succedent produces a proof tree, but obviously the series of inferences required to achieve this is different from that required for the original formula. At this point, we are more interested in a refutation normal form of the Gentzen-style deduction, so the above NNF is negated and manipulated back into NNF as follows:

$$\neg(\exists x(P(x) \wedge \neg Q(x)) \vee \exists x\neg P(x) \vee \forall x Q(x))$$
$$\neg\exists x(P(x) \wedge \neg Q(x)) \wedge \neg\exists x\neg P(x) \wedge \neg\forall x Q(x) \quad \text{De Morgan}$$
$$\forall x\neg(P(x) \wedge \neg Q(x)) \wedge \forall x\neg\neg P(x) \wedge \exists x\neg Q(x) \quad \text{identities}$$
$$\forall x(\neg P(x) \vee Q(x)) \wedge \forall x P(x) \wedge \exists x\neg Q(x) \quad \text{De Morgan}$$

noting that the result could have been written directly as the dual of the earlier NNF formula. The resulting formula may be made the antecedent of a sequent and deduction might proceed using the subformula rules introduced in Chapter 2. A refutation shown in Figure 3.6 adopts this approach, creating a midsequent after three quantifier inferences and axioms after further propositional operations. Notice that after the midsequent is obtained, the process of deduction is exactly that described in the preceding section on propositions. A single generalised "left $\wedge$" rule may be used to convert the midsequent to a list of clauses and the "left $\neg$" rule might be completely avoided if we accept an antecedent with dual literals as an axiom. As a result, only the "left $\vee$" rule is significant in the production of the final result.

In order to develop a resolution refutation approach for quantified formulas, we first replace existentially quantified variables with Skolem functions. In this particular case, a single variable is replaced by a constant to give the formula

$$\forall x(\neg P(x) \vee Q(x)) \wedge \forall y P(y) \wedge \neg Q(a)$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x(\neg P(x) \vee Q(x)) \wedge \forall x P(x) \wedge \exists x\,\neg Q(x) \Rightarrow}{\forall x(\neg P(x) \vee Q(x)) \wedge \forall x P(x) \wedge \neg Q(a) \Rightarrow}\ \text{left } \exists \text{ on } 1}{\forall x(\neg P(x) \vee Q(x)) \wedge P(a) \wedge \neg Q(a) \Rightarrow}\ \text{left } \forall \text{ on } 2}{(\neg P(a) \vee Q(a)) \wedge P(a) \wedge \neg Q(a) \Rightarrow}\ \text{left } \forall \text{ on } 3}{(\neg P(a) \vee Q(a)), P(a), \neg Q(a) \Rightarrow}\ \text{gen left } \wedge \text{ on } 5}{\neg P(a), P(a), \neg Q(a) \Rightarrow \qquad Q(a), P(a), \neg Q(a) \Rightarrow}\ \text{left } \vee \text{ on } 6$$

midsequent

Figure 3.6 A refutation deduction

At the same time, the second use of variable x has been replaced by a distinct variable y. Since all remaining variables are represented by distinct symbols and must be universally quantified, there is no longer any need to show quantifier symbols and the formula appears as

$$(\neg P(x) \vee Q(x)) \wedge P(y) \wedge \neg Q(a)$$

or, if written in clausal form, as follows:

$$\{\neg P(x), Q(x)\}, \{P(y)\}, \{\neg Q(a)\}$$

Resolution steps can now be applied to the clausal form in a development of the procedure explained earlier. Again the object is to resolve out new clauses from pairs of existing clauses that contain clashing atoms, but the existence of terms in the atoms makes this more complicated than before. Although $\neg P(x)$ and $P(y)$ do not appear to clash, if x is substituted for y or vice versa, the atoms become identical and $Q(x)$ or $Q(y)$ may be resolved from the pair. Similarly $Q(x)$ and $\neg Q(a)$ do not immediately clash, but the constant a may be substituted for variable x and a clash obtained. Notice that the substitution may be carried out either way round when both are variables, but in only one direction if one is a constant. When a choice exists, it is better to take the option that leaves the greatest number of variables in the resolvant, allowing further resolutions to take place. Figure 3.7 shows how these interleaved substitutions and resolutions lead to an empty clause, proving that the formula being resolved is unsatisfiable. Since this formula is the negation of the one in which we are really interested, the original unnegated formula must be valid.

Gentzen-style proofs become increasingly difficult as the size and complexity of the sequent to be proven or refuted increases. Left universal quantifications present particular problems because they might have to be used several times to obtain different ground-state formulas that might then be subjected to propositional rules. It is difficult in such deductions to know which instantiations are required to close the final deduction tree, so much experimentation is required. The problem is one of needing to see ahead in order to instantiate universally quantified formulas with appropriate constants. Repeated resolution of Skolemised formulas, on the other hand, leads to a systematic method of attempting every possible instantiation until a refutation is achieved. Furthermore, resolution is mechanised in a fairly direct way to produce the logic languages described in the following sections.

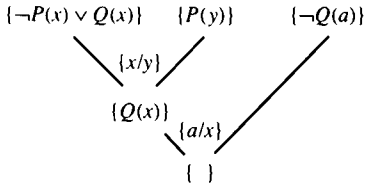

Figure 3.7 A resolution diagram with predicates

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall yP(a,y) \wedge \forall x\neg P(x,b) \Rightarrow}{P(a,b) \wedge \forall x\neg P(x,b) \Rightarrow}\ \text{left } \forall \text{ on 1}}{P(a,b) \wedge \neg P(a,b) \Rightarrow}\ \text{left } \forall \text{ on 2}}{P(a,b), \neg P(a,b) \Rightarrow}\ \text{left } \wedge \text{ on 3}}{P(a,b) \Rightarrow P(a,b)}\ \text{left } \neg \text{ on 4}$$

midseq

Figure 3.8 A refutation deduction

A hint of the problems that might be encountered in a G system proof appears in a deduction from the following small formula:

$$\exists x\forall yP(x,y) \rightarrow \forall y\exists xP(x,y)$$

This example is quickly translated into negation normal form

$$\forall x\exists y\neg P(x,y) \vee \forall y\exists xP(x,y)$$

Refutation requires that we take the dual of this formula which, by inspection, we are able to write as

$$\exists x\forall yP(x,y) \wedge \exists y\forall x\neg P(x,y)$$

Existentially quantified variables may then be replaced by Skolem constants to give the simpler form

$$\forall yP(a,y) \vee \forall x\neg P(x,b)$$

We can then show this to be unsatisfiable in the deduction tree of Figure 3.8. The deduction tree clearly demonstrates unsatisfiability, but the way in which this result is obtained indicates problems ahead. Notice that formula $\forall yP(a,y)$ is instantiated to $P(a,b)$, but it could have been instantiated with domain element a to $P(a,a)$ or with any other available domain element. Equally, $\forall x\neg P(x,b)$ might have been instantiated to $\neg P(b,b)$ but this was not done because it was obvious that $\neg P(a,b)$ was required to terminate the deduction. In this case there are just two formulas in the initial antecedent and the necessary instantiations are obvious; but as the examples get larger, the required instantiations become less obvious. If the above formula is drafted in clausal form as $\{P(a,y)\}$ $\{\neg P(x,b)\}$, it is clear that substitution $\{a/x, b/y\}$ produces a clash of atoms and no other substitutions need be considered. In larger examples the guidance provided by the need to produce clashing pairs makes problems tractable.

3.2.1 Substitution and resolution

Clauses derived from full first-order logic formulas are resolved when a clashing pair occurs in two different clauses in much the same way as clauses of propositions. A general scheme for resolving formula literals might be written as

$$\{A(x), P(z)\} \wedge \{B(y), \neg P(z)\} \rightarrow \{A(x), B(y)\}$$

However, unlike the propositional case, it is possible to have pairs of literals that clash only after appropriate substitutions of terms have been made in the atomic arguments. One approach is to instantiate all terms to ground states, i.e. to replace all variables by constants, and then attempt to resolve the resulting ground clauses. The problem here is that a great many different ground clauses are possible and it is not always easy to see which ones are required to produce a refutation. This is in fact much the same problem that arises in choosing suitable instantiations in a semantic tableau or G system tree deduction. A cleverer approach interleaves substitutions and resolutions so that the minimum necessary substitutions are made before each resolution step. An accumulation of each of these individual substitutions is then used to compute an equivalent single-step substitution.

The process of making two atoms the same through substitution is called unification. Taking the simplest example first, $P(z)$ unifies with $P(w)$ after the substitution $\{w/z\}$ or with $P(c)$ after substitution $\{c/z\}$. Atoms with different predicate symbols cannot be unified and cannot therefore be resolved out of a clause, e.g. $P(x)$ and $Q(y)$ cannot be unified. Substitutions only apply to variables appearing as arguments in atoms so that, even when predicate symbols are identical, differences of constants prevent unification, e.g. $P(a)$ and $P(b)$ cannot be unified. Atoms with two arguments might require a double substitution before unification occurs, e.g. $Q(a,x)$ and $Q(y,b)$ are unified by the substitution $\{a/y, b/x\}$. Sometimes one substitution forces another; for example, the unification of $Q(x,x)$ and $Q(a,y)$ requires substitution a/x, but since x occurs twice in the first atom, a further substitution a/y is also necessary.

Functions occurring as arguments are handled in much the same way as simple constants, e.g. $Q(x, f(a))$ and $Q(g(b), y)$ are unified by the substitution $\{g(b)/x, f(a)/y\}$. In a slightly more complicated example, $Q(x, f(x))$ and $Q(g(b), y)$ are unified by the substitution $\{g(b)/x, f(g(b))/y\}$. Notice that ground atoms result from unification in both of these examples and there is no alternative substitution. In contrast, the atoms $Q(x, f(x))$ and $Q(y, f(z))$ may be unified in two ways:

a. By $\{c/x, c/y, c/z\}$ to give $Q(c, f(c))$.

b. By $\{x/y, x/z\}$ to give $Q(x, f(x))$.

The second example admits a further substitution that produces the same result as the first:

$$Q(x, f(x))\ \{c/x\} = Q(c, f(c))$$

and is considered to be a more general unifier. In fact, this substitution is the most general unifier (mgu) possible for the atoms, leaving open the maximum possible number of subsequent substitutions. An mgu is obtained by avoiding the instantiation of constants wherever possible.

Atoms with more than two arguments are treated in exactly the same way as above, e.g. atoms $R(a, f(x), y)$ and $R(w, f(z), z)$ are unified in two ways:

a. By $\{a/w, a/x, a/y, a/z\}$ to give $R(a, f(a), a)$.

b. By $\{a/w, z/x, z/y\}$ to give $R(a, f(z), z)$.

The second method is the most general unifier for the two atoms. Notice that the most general unifier is not unique and $Q(a, f(x), x)$ or $Q(a, f(y), y)$ might have been obtained with a different strategy.

3.2.2 Robinson's algorithm

Unifications might be found by inspection, as in the examples above, but if the process is to be mechanised, they have to be discovered by a fixed algorithm. Robinson's algorithm for finding an mgu is fairly simple: just work from left to right through the arguments of a pair of atoms, making whatever substitutions are necessary to unify each individual argument. A composition of the substitutions then provides the unifier. The arguments in atoms $P(w, f(a), z)$ and $P(b,x,y)$ are unified as follows:

	$P(w, f(a), z)$	$P(b,x,y)$	
Arg 1	$P(b, f(a), z)$	$P(b,x,y)$	$\{b/w\}$
Arg 2	$P(b, f(a), z)$	$P(b, f(a), y)$	$\{f(a)/x\}$
Arg 3	$P(b, f(a), z)$	$P(b, f(a), z)$	$\{z/y\}$

When the last pair of arguments has been unified, the atoms themselves have been unified and, provided no unnecessary constants have been introduced, this procedure generates the most general unifier for the two atoms. A single equivalent substitution is obtained from the composition of individual substitutions

$$\{b/w\} \circ \{f(a)/x\} \circ \{z/y\} = \{b/w, f(a)/x, z/y\}$$

Here the single unifier is the sum of the individual substitutions, but this might not always be the case. A variable substituted into a term might itself be removed in a later substitution and this chaining of replacements has to be reflected when individual steps are combined. Such a problem occurs when atoms $Q(x, f(y,a))$ and $Q(z, f(z,z))$ are unified by Robinson's method:

	$Q(x, f(y,a))$	$Q(z, f(z,z))$	
Arg 1	$Q(z, f(y,a))$	$Q(z, f(z,z))$	$\{z/x\}$
Arg 2	$Q(z, f(z,a))$	$Q(z, f(z,z))$	$\{z/y\}$
Arg 3	$Q(a, f(a,a))$	$Q(a, f(a,a))$	$\{a/z\}$

An allowance has to be made for the fact that x and y were initally replaced by z, but this variable was then itself replaced by constant a, so the net effect is that all variables are replaced by constant a:

$$\{z/x\} \circ \{z/y\} \circ \{a/z\} = \{a/x, a/y, a/z\}$$

A final and larger example unifies atoms $S(x, g(f(z), v, a))$ and $S(f(y), g(x, h(x), y))$, demonstrating the approach when argument functions themselves contain functions:

	$S(x, g(f(z)), v, a))$	$S(f(y), g(x, h(x), y))$	
Arg 1	$S(f(y), g(f(z)), v, a))$	$S(f(y), g(f(y), h(f(y)), y))$	$\{f(y)/x\}$
Arg 2	$S(f(y), g(f(y)), v, a))$	$S(f(y), g(f(y)), h(f(y)), y))$	$\{y/z\}$
Arg 3	$S(f(y), g(f(y)), h(f(y)), a))$	$S(f(y), g(f(y), h(f(y)), y))$	$\{h(f(y))/v\}$
Arg 4	$S(f(a), g(f(a)), h(f(a)), a))$	$S(f(a), g(f(a), h(f(a)), a))$	$\{a/y\}$

and the unifying substitution is obtained from the sum of individual steps:

$$\{f(y)/x\} \circ \{y/z\} \circ \{h(f(y))/v\} \circ \{a/y\} = \{f(a)/x, a/z, h(f(a))/v, a/y\}$$

EXERCISES 3.2

1. Convert each of the following valid formulas into negation normal form and convert the resulting formula to its dual form:
   a. $\forall x P(x) \wedge \forall x Q(x) \rightarrow \forall x(P(x) \vee Q(x))$
   b. $\exists x(P(x) \wedge Q(x)) \rightarrow \exists x P(x) \vee \exists x Q(x)$
   c. $\forall x(P(x) \rightarrow Q(x)) \wedge \exists x(R(x) \wedge P(x)) \rightarrow \exists x(R(x) \vee Q(x))$
   d. $\forall x(P(x) \rightarrow Q(x)) \leftrightarrow \neg\exists x(P(x) \wedge \neg Q(x))$
   e. $\exists x(P(x) \wedge \forall y(Q(y) \rightarrow R(x,y))) \wedge$
      $(\forall x P(x) \rightarrow \forall y(S(y) \rightarrow \neg R(x,y))) \rightarrow \forall x(Q(x) \vee \neg S(x))$
2. Produce refutation trees similar to that shown in Figure 3.6 for each of the dual formulas obtained in the previous exercise.
3. Skolemise out any existentially quantified variables in the NNF dual formulas obtained from Exercise 1, convert the resulting formulas to clausal form and show they are unsatisfiable in a refutation diagram.
4. Produce unifying substitutions for the following pairs of atoms or explain why unification is not possible:
   $P(a)$ and $Q(x)$
   $R(f(a))$ and $R(f(b))$
   $Q(x)$ and $Q(f(a))$
   $Q(x)$ and $Q(f(y))$
   $R(x, f(x))$ and $P(f(a), y)$
   $R(x, f(a))$ and $R(g(b), f(y))$

5. Use Robinson's algorithm to unify the following pairs of atoms and so produce unifying substitutions:

   $P(w, f(a), z)$ and $P(b, x, g(x))$
   $R(x, z, f(a))$ and $R(y, g(b), x)$
   $Q(x, x, a)$ and $Q(y, f(z), z)$

3.3 HORN CLAUSES AND FORWARD CHAINING

Resolution might show that a collection of clauses is unsatisfiable by showing that successive removal of clashing literals from pairs of disjunctions leads to an empty clause. The technique can be applied to sets of clauses containing arbitrary numbers of literals with and without negation symbols. In order to convert logic statements into a form that may be animated, clauses have to be restricted to the Horn clause form described below. Statements in Horn clause form are equivalent to a logic program that may be used to answer questions on the basis of a number of axioms called facts and rules. This section is concerned with the generation of all theorems or *true* statements from program axioms through the process of forward chaining. The following section uses a technique called backward chaining to decide whether a specific statement is *true* in the environment created by a particular program.

3.3.1 Horn clauses

A Horn clause is a disjunction of literals containing at most one positive literal. Clauses containing this one allowed positive literal are called definite clauses whereas those without such a literal are called negative clauses. Definite clauses have the general form

$$R \vee \neg A \vee \neg B \vee \neg C \vee \ldots$$

but one application of the generalised De Morgan rule to the negated atoms in this formula introduces the following alternative formulations:

$$R \vee \neg(A \wedge B \wedge C \wedge \ldots)$$
$$R \leftarrow A \wedge B \wedge C \wedge \ldots$$

Notice that in logic programming it is convenient to write an implication from right to left and to read the reversed implication statement as "if". Thus R is *true* if A and B and $C \ldots$ are *true*. A reverse implication of this sort is related to disjunction through a variation of the familiar identity

$$Y \vee \neg X \equiv Y \leftarrow X$$

A definite clause must contain one positive literal, but need not contain a negative literal, so a single positive literal is also a Horn clause. Such a clause is usually shown in the form $R \leftarrow$ or more simply as just R.

A definite program is a collection of definite clauses written in both resolution form and as a logic program:

1. $p \wedge$	p
2. $q \wedge$	q
3. $r \vee \neg t \vee \neg s \wedge$	$r \leftarrow t, s$
4. $r \vee \neg q \vee \neg t \wedge$	$r \leftarrow q, t$
5. $t \vee \neg p \wedge$	$t \leftarrow p$
6. $t \vee \neg q \vee \neg s$	$t \leftarrow q, s$

The meaning or interpretation of a propositional definite program is a valuation of the atomic statements in the program, usually expressed as the set of those statements that are *true*. Thus an interpretation in which none of the program statements is *true* is shown as an empty set:

$$I0 = \{\ \}$$

but this cannot be a model for the program. A first pass through the program reveals that statements p and q occur as facts and any valuation that acts as a model must include them among its *true* statements. The next attempt at a model might therefore be the set

$$I1 = \{p, q\}$$

but this too proves to be inadequate. Passing down the program a second time, armed with the knowledge that elements p and q are *true*, we encounter the clause $t \leftarrow p$ and deduce by resolution that t must also be *true*. This leads to a further improved attempt:

$$I2 = \{p, q, t\}$$

but another pass through the program, starting with $I2$, establishes r as a consequence of the clause $r \leftarrow q, t$, forcing us again to expand the interpretation:

$$I3 = \{p, q, t, r\}$$

Further passes through the program do not produce any new *true* statements, so $I3$ is a "fixed-point" interpretation for the program. The meaning of a program is clearly obtained by forward chaining in repeated passes through the program, generating greater numbers of *true* statements until the fixed point is reached. The distinctive advantage of definite programs over arbitrary clauses is that a fixed point of this kind is always obtained.

Clauses derived from first-order logic formulas carry implicit universal quantifications and might be better shown as

$$\forall(R \leftarrow A \wedge B \wedge C \wedge \ldots)$$
$$\forall(R \leftarrow)$$

Once again, these clauses are equivalent to the axioms of a formal system, so the program is a theory. An interpretation that is a model for each of these axioms is also a model for the theory.

Unfortunately there exist an infinite number of possible interpretations when a program includes predicates, and some way of expressing every possible interpretation in a single form is required. One way of doing this is suggested by the Herbrand and Skolem–Herbrand–Gödel theorems of Chapter 2. It was shown there that quantified formulas are proven or refuted if and only if special kinds of quantifier-free formulas appearing in the midsequent are proven or refuted. The deduction trees used to demonstrate validity or contradiction use one particular interpretation to characterise the properties of an infinite number of other interpretations. In effect, a Herbrand interpretation uses the formal symbols themselves to characterise every possible interpretation. Thus, predicate symbols P, Q, ... are represented by the letters P, Q, ... and term symbols a, b, c, ... , f, g, h, ... are also represented by their own characters. Existentially quantified variables will have been replaced in refutation mode by Skolem functions and universally quantified variables will not introduce further constants into a definite program. As a result, the domain of a Herbrand interpretation is restricted to a special set called the Herbrand universe (sometimes called the Herbrand domain). This restriction on domain elements limits the number of possible interpretations of each predicate to a set called the Herbrand base; the set that can be constructed from objects in the Herbrand universe. A small definite program with predicates is now provided as an example:

$P(a)$
$Q(b)$
$P(x) \leftarrow Q(x)$
$R(y) \leftarrow P(y)$

This program contains no functions and thus has the simple Herbrand universe $\{a, b\}$ with just two constants. Its Herbrand base contains all the atoms of the program in every possible ground state, and since there are two constants and three arity-one predicates, this amounts to a set of six ground atoms:

$$\{P(a), P(b), Q(a), Q(b), R(a), R(b)\}$$

Herbrand interpretations of programs with predicates differ only in their valuations of atoms in the Herbrand base. An interpretation is usually described by the set of atoms mapped to *true*, all other atoms being assumed *false*. Clearly the simplest possible Herbrand interpretation ($I0$) for the above program is the empty set of base atoms

$$I0 = \{\ \}$$

but this is not a model for the program. No base atom evaluates to *true* in this interpretation, but the program requires that $P(a)$ and $Q(b)$ are *true* because these are facts in the program. Any interpretation capable of acting as a model for the program must at least contain these two atoms, so a next attempt might be

$$I1 = \{P(a), Q(b)\}$$

Here the two atoms shown are evaluated to *true* and the remaining atoms to *false*. Although this interpretation acts as a model for the first two clauses of the program,

it fails on the third. Interpretation $I1$ evaluates atom $Q(b)$ as *true* and the program contains formula $P(x) \leftarrow Q(x)$, making atom $P(b)$ *true* by resolution A Herbrand interpretation without this atom cannot act as a model for the program, forcing us to expand the previous attempt to

$$I2 = \{P(a), Q(b), P(b)\}$$

Similar reasoning based on the fourth clause demands a further expansion of the interpretation to include $R(a)$ and finally to Herbrand interpretation $I3$, which does act as a model for the program:

$$I3 = \{P(a), Q(b), P(b), R(a)\}$$

It should be clear that the method used to arrive at this interpretation is the forward-chaining procedure described earlier. An interpretation containing the least number of elements is always obtained when it is generated in this way, and any other interpretation that is a model for the program must contain all the elements of this set. Although they are models, interpretations taking further atoms from the Herbrand base, including interpretations taking the base itself, are less useful than the minimum model. In fact, the set of atoms obtained as the fixed point of forward chaining through a definite program defines the meaning of the program: it is a statement of all the atoms that may be proven *true*. Thus the least Herbrand model defines the semantics of a program.

As a second example of a logic program, consider the definite formulas describing the linkages shown in Figure 3.9. This diagram contains an example of a directed acyclic graph, directed because the arrows limit movements in one direction only, acyclic because it is not possible to return to the same point in the graph. An alternative representation of the information in the diagram is possible through the following facts:

$$Path(a,b),\ Path(b,d),\ Path(d,e),\ Path(a,c),\ Path(c,e)$$

Suppose now that we want to describe every pair of points that are connected by paths in the graph, listing all allowed routes in the graph. The simplest routes are the links themselves and this observation could be formalised by the rule

$$Route(x,y) \leftarrow Path(x,y)$$

There are also a number of routes passing over more than one link. For example, a route from b to e via d is possible and this may be expressed in the clause

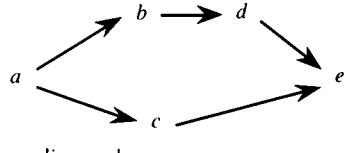

Figure 3.9 A directed acyclic graph

$Route(b,e) \leftarrow Path(b,d), Path(d,e)$

Routes over increasing numbers of intermediate points may be described by introducing variables into similar clauses:

$Route(x,z) \leftarrow Path(x,y), Path(y,z)$
$Route(w,z) \leftarrow Path(w,x), Path(x,y), Path(y,z)$

A route might pass overy many links in a larger graph and different rules would have to be used for routes with different numbers of intermediate stages. A more elegant solution than this encapsulates routes with differing numbers of intermediate points in a single rule:

$Route(x,z) \leftarrow Path(x,y), Route(y,z)$

There is a route from x to z if there is a path from x to some intermediate point y and a route from there to z. The final program then has the form

1. $Path(a,b)$
2. $Path(a,c)$
3. $Path(b,d)$
4. $Path(c,e)$
5. $Path(d,e)$
6. $Route(x,y) \leftarrow Path(x,y)$
7. $Route(x,z) \leftarrow Path(x,y), Route(y,z)$

and it would be of interest to find the least Herbrand model of this program. First of all, the Herbrand universe of this program is the set $\{a, b, c, d, e\}$ and the Herbrand base contains the two predicates in every possible ground state:

$\{Path(a,a), Path(a,b), Path(b,a), \ldots$
$Route(a,a), Route(a,b), Route(b,a), \ldots \}$

There are five constants in the Herbrand universe and thus $25 = 5 \times 5$ pairs of arguments that might appear in either of the two predicates, producing a Herbrand base of 50 atoms. An interpretation $I0$ in which none of these base atoms is assigned *true* is shown as the empty set

$I0 = \{\ \}$

but this is certainly not a model for the program. In order to find the minimal interpretation required for the least Herbrand model, a series of interpretations based on forward chaining through the program is explored. Interpretation $I1$ contains those base atoms known to be true from the facts of the program alone:

$I1 = \{Path(a,b), Path(a,c), Path(b,d), Path(c,e), Path(d,e)\}$

A second attempt adds the *Route* ground atoms generated by clause 6:

$I2 = I1 \cup \{Route(a,b), Route(a,c), Route(b,d), Route(c,e), Route(d,e)\}$

A further pass uses clause 7 to generate interpretation $I3$, containing routes with one intermediate point in addition to the atoms of $I2$:

$I3 = I2 \cup \{Route(a,d), Route(a,e), Route(b,e)\}$

and a final pass generates the one route spanning two intermediate points:

$I4 = I3 \cup \{Route(a,e)\}$

but this adds nothing new because $Route(a,e)$ already occurs in $I3$. No more ground states can be added in this way, so a fixed-point interpretation has been obtained. The model $I3$ obtained by forward chaining to the fixed point is called the least Herbrand model of the program and is a characteristic feature of a definite program, defining the meaning of the program. Different models may be obtained by adding further base elements to the least Herbrand model, but such models would not be particularly helpful or informative.

Both of the examples above have a finite universe, making the enumeration of the least Herbrand model a practical proposition, but the presence of functions makes this impossible. Interpretations for functions are considered in more detail in Chapter 6, so we just note the problem of an infinite Herbrand universe here. A formal system defining the natural numbers is provided by the constant *zero* and the successor function *succ* as follows

$zero$, $succ(zero)$, $succ(succ(zero))$, etc.

and a program to add such numbers may be written as follows:

$Add(x, zero, x)$
$Add(x, succ(y), succ(z)) \leftarrow Add(x,y,z)$

Adding x to *zero* gives x and adding x to $succ(y)$ gives $succ(z)$ if adding x to y gives z. The series of constants above is usually interpreted by the numbers 0, 1, 2, . . . , but a Herbrand interpretation takes the strings themselves as the interpretation. Hence the Herbrand universe of this program is an infinite but denumerable series of constants; attempts to define a fixed-point interpretation, as we have done earlier, will therefore not succeed.

3.3.2 Soundness and completeness

Logic programs are syntactic statements in a simplified form of first-order logic: they are collections of axioms from which further statements may be deduced according to a set of deduction rules. Since they consist entirely of definite clauses, only positive statements may be derived. At the same time, the least Herbrand model represents a particular interpretation of the program, equivalent to the intended interpretation. This representation is generated by a production rule that gradually increases the number of known *true* facts until a fixed point is obtained. Now if a given ground fact is derived from program P by proof theoretic reasoning, we write

$P \vdash F$

and if that fact is *true* in the least Herbrand interpretation (M_P) we write

$M_P \models F$

But the least Herbrand model is a collection of all the facts that can be derived from the program, so logical consequence must follow from derivability. A predicate that is derived from the program must be *true* and the method of deduction is therefore sound; conversely, the *true* statements are those that may be derived, so the method is complete.

3.3.3 The Datalog language

A forward-chaining strategy such as that described in connection with the least Herbrand model may be used as the basis of a logic language. Programs are divided into two parts: an extensional part containing factual information and an intensional part consisting of rules from which further ground states are derived. Such programs are seen as databases that store some of their information in the form of production rules but most of it in the form of an extensional database of facts. When presented with a query, the extensional database is first checked; if this check fails and there are rules that may be applied, further ground facts are produced. Repeated applications of the rules produce more ground facts until the fixed point is reached; if the desired predicate has not been found by this time, it will never be found. There are many different languages of this kind, but all of them are included in a general area described as Datalog languages. If we have a specific query that can be presented to a definite Datalog program, the question arising is, can this predicate be derived from the facts and rules of the program? Suppose we wish to know if *Route*(*a*,*d*) follows from the program *P* above, we are asking if it can be proven from the program that

$P \vdash Route(a,d)$

or, in terms of the least Herbrand interpretation M_P, if this atom is contained in the fixed point obtained by forward chaining:

$\{ \ldots, Route(a,d), \ldots \} \models Route(a,d)$

The basic principles of Datalog programs are clearly quite simple: just forward chain through the program until the desired result is obtained. If a fixed point is obtained without finding the result, it is assumed to be untrue.

A Datalog program has fairly simple semantics: its meaning is revealed by forward chaining to the fixed point and is therefore exactly that of the least Herbrand model. Intensional database rules may be applied in any order, but the same defining fixed-point set of predicates is always obtained. Forward chaining generates

duplicate copies of the same predicate, and implementations have to ensure that such duplicates are removed at each stage. Even so, the method generates many predicates that are of no interest to a user, and its attractive semantics is to some extent offset by problems in implementation. Much of the work in this area is carried out in connection with the set-oriented relational databases described in Chapter 10.

EXERCISES 3.3

1. Obtain the least Herbrand interpretation of the following definite program by forward chaining until a fixed point is obtained:

   p
   $s \leftarrow p$
   $s \leftarrow r, v$
   $r \leftarrow s$
   $t \leftarrow u, s$
   $v \leftarrow p, r$

2. Write down the Herbrand universe, Herbrand base and least Herbrand interpretation for the following program:

   $P(a)$
   $Q(b)$
   $R(x) \leftarrow Q(x)$
   $S(x) \leftarrow P(x), R(x)$

3. Write down the Herbrand universe, Herbrand base and least Herbrand interpretation for the following program:

   $R(a)$
   $P(a,b)$
   $P(b,c)$
   $Q(x,y) \leftarrow P(y,x)$
   $S(x) \leftarrow Q(x,y), R(y)$

3.4 BACKWARD CHAINING AND SLD RESOLUTION

Backward chaining does not require the generation and storage of large numbers of intermediate predicates because it works backwards from the goal using only the rules it needs. This technique is based on the simple observation that if a statement *S* can be derived from a program, i.e.

$Program \vdash S$

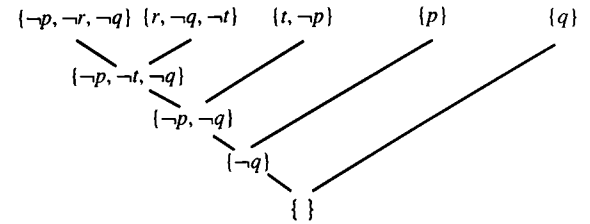

Figure 3.10 Backward chaining from a goal clause

a conjugation of the program and negated statement S must be inconsistent. To show that S follows from the program, we have to show that the formula *Program* $\wedge \neg S$ is inconsistent, or in terms of the interpretation, that it is a contradiction. To show that *Route(a,d)* can be derived from our earlier program P, we have to show that

$$\{P\} \wedge \neg Route(a,d)$$

must be a contradiction. Thus the production of a contradiction between the negated query and the program refutes the statement and indirectly shows the query to be *true*. This is of course the refutation style discussed earlier, so backward chaining is essentially a refutation argument.

The definite program given at the beginning of the previous section may be written in clausal form as

$$\{p\}, \{q\}, \{r, \neg t, \neg s\}, \{r, \neg q, \neg t\}, \{t, \neg p\}, \{t, \neg q, \neg s\}$$

Forward chaining revealed that this program is equivalent to the clausal form $\{p\}, \{q\}, \{t\}, \{r\}$. In other words, the meaning of the program may be expressed as $p \wedge q \wedge t \wedge r$ and a question like "is $p \wedge q \wedge r$ *true*?" is easily answered by checking that propositional statements in the query appear in the Herbrand model.

Adopting the backward-chaining approach, we show that proposition $p \wedge q \wedge r$ is *true* by showing that the negation of this query is inconsistent with the program. The dual of the query is $\neg p \vee \neg q \vee \neg r$ and this is expressed as a negative Horn clause $\{\neg p, \neg q, \neg r\}$. Figure 3.10 shows how this goal is resolved against succesive program clauses until the empty clause is obtained, proving that the negated query is inconsistent with the program. Goals are resolved against definite clauses to produce subgoals until the empty clause is obtained, refuting a conjugation of the negated clause with the program.

Such a procedure obviously has to begin with the goal clause, but there remains a choice of literals within the goal that might be selected for resolution and a choice of clauses against which the selected literal might be resolved. The method shown in Figure 3.10 selects the middle literal of the initial goal and resolves this against an appropriate clause:

$$\{\neg p, \neg r, \neg q\} \wedge \{r, \neg q, \neg t\} \rightarrow \{\neg p, \neg q, \neg q, \neg t\}$$

but the duplicated $\neg q$ literal is removed after resolution. Next the third literal of a modified goal is selected for resolution against an appropriate clause, then the first and finally the only remaining one. This ad hoc selection of literals has to be replaced by a fixed selection rule when the process of resolution is automated and a simple selection rule that always selects the leftmost literal is usually chosen. The process of resolving a goal with a set of definite clauses using a fixed selection rule is called SLD resolution, the acronym arising from Linear resolution for Definite clauses with Selection function. If the leftmost goal literal is always selected for resolution, the process is called normal SLD resolution and the result of resolution is described as a normal resolvant.

Not all the available clauses were used to produce the above refutation and it is clear that different initial choices might lead nowhere, i.e. they might fail to produce the empty clause. On the other hand, there might be other ways of producing an empty goal, allowing multiple refutations. Clauses are usually chosen for resolution in the order in which they appear in the logic program text, on a top-to-bottom basis when the program is written as

p
q
$r \leftarrow t \wedge s$
$r \leftarrow q \wedge t$
$t \leftarrow p$
$t \leftarrow q \wedge s$

Actually the choice of literals to be clashed in Figure 3.10 was based on the desire to produce a neat resolution diagram. An automated search using normal SLD resolvants traces out a rather less tidy path that can be displayed in the form of an SLD tree. A tree beginning with the goal $\neg(r \wedge q \wedge t)$ following the program above is shown in Figure 3.11. Here leftmost goal literals are chosen for resolution against program clauses and the first available clause in the program is taken for resolution. Following the leftmost branch of this tree to its leaf, we see that atom s needs to be resolved against a clause, but there is no appropriate clause in the program. At this point, the search has failed and has to backtrack to an earlier point where an alternative clause can be used. A second resolvant is possible for literal t, but though this is tried it also leads to a leftmost literal that cannot be resolved. The whole process continues in a very mechanical way, leading to an empty clause (shown as a box) and a further failed attempt.

Note that the SLD tree is explored from left to right so that two branches have been tried and failed before the empty clause is obtained. Furthermore, the SLD mechanism continues to explore alternative ways of showing the goal to be inconsistent with the program, even after it has already done so. This method of searching is described as depth-first because it descends to the tips of each branch before searching alternatives to the right.

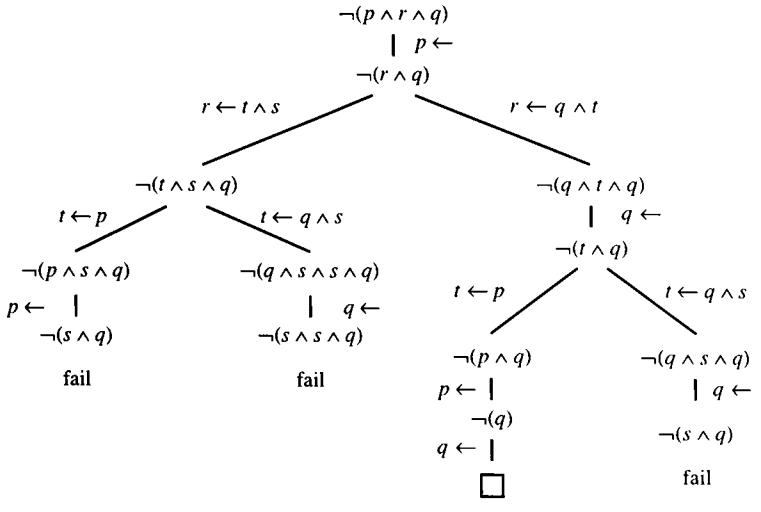

Figure 3.11 An SLD tree

3.4.1 SLD resolution for first-order formulas

Resolution is essentially a feature of propositional logic, so predicate literals are treated as simple statements. There exists, however, the extra complication that terms might have to be unified through substitutions before resolution is possible, and such substitutions have to be made throughout both goal and clause. Figure 3.12 shows how a goal and clause containing predicates are resolved through a most general unifier μ_1 to give a substituted negative clause for further resolution. A series of normal resolutions of this kind $\mu_1, \mu_2, \ldots, \mu_n$ applied to an initial goal G0 might eventually lead to an empty clause. If so, the substitutions made would then be the composition of a series of most general unifiers:

$$\mu = \mu_1 \circ \mu_2 \circ \mu_3 \circ \ldots \circ \mu_n$$

Consider the route procedure with the following path data:

C1 $Route(x,y) \leftarrow Path(x,y)$
C2 $Route(x,y) \leftarrow Path(x,z) \wedge Route(z,y)$
C3 $Path(a,b)$
C4 $Path(b,c)$

and take as a goal the predicate $Route(a,c)$. Normal resolution produces the SLD tree shown in Figure 3.13 with one successful refutation and two failed paths, so we deduce $Route(a,c)$. Each resolution step on the way to the empty clause required

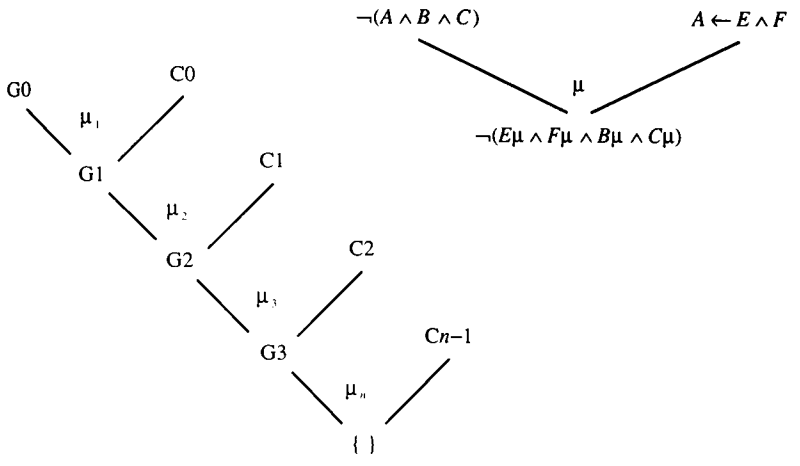

Figure 3.12 Backward chaining with substitutions

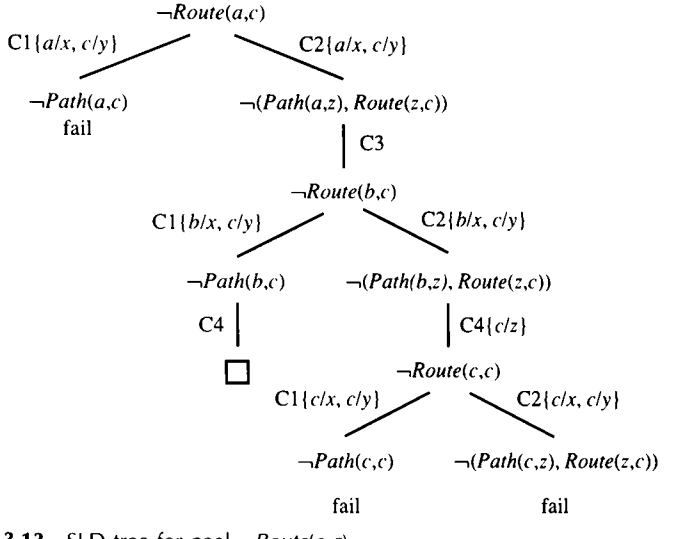

Figure 3.13 SLD tree for goal ¬*Route*(*a*,*c*)

a substitution and the composition of all such unifiers records a complete set of substitutions. However, these substitutions are of no interest because the goal contains no variables. In this case a refutation is itself the answer.

Suppose we begin the SLD tree with the goal containing two variable arguments, $Route(v,w)$, and apply clauses of the same program to produce the tree shown in

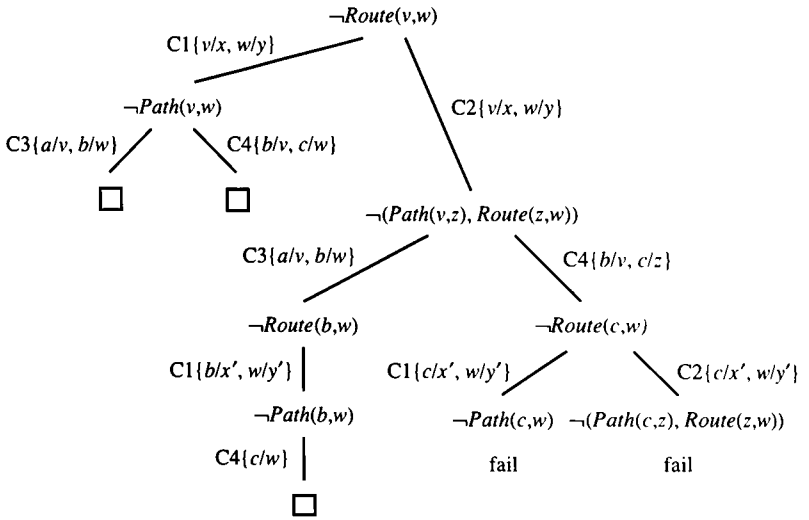

Figure 3.14 SLD tree for goal ¬*Route*(*v*,*w*)

Figure 3.14. This produces three successful refutation paths followed by two failed paths as the SLD mechanism attempts to find a route from point *c*. In this case the composition of unifiers required to achieve the empty clause is important: it provides the instantiated constants that cause refutation. Thus the first instantiation to produce a result is the substitution

$$\sigma = \{v/x, w/y\} \circ \{a/v, b/w\}$$
$$\sigma = \{a/v, b/w\}$$

Simplifications are possible here because *v* and *w* are substituted for *x* and *y* but the substituted variables are themselves subsequently replaced by *a* and *b*. Composition of these changes leads to a simpler substitution. Similarly, the series of substitutions required for the third refutation reduces to a much simpler single-stage substitution:

$$\rho = \{v/x, w/y\} \circ \{a/v, b/w\} \circ \{b/x', w/y'\} \circ \{c/w\}$$
$$\rho = \{a/v, c/w\}$$

The order of predicates within the body of a rule and the order of the clauses themselves may be changed to give a modified form of the program:

C1 $Route(x,y) \leftarrow Route(z,y) \wedge Path(x,z)$
C2 $Route(x,y) \leftarrow Path(x,y)$
C3 $Path(a,b)$
C4 $Path(b,c)$

If we now grow an SLD tree from the goal *Route*(*a*,*c*), we obtain Figure 3.15. Normal resolution combined with the top-to-bottom selection of clauses now produces

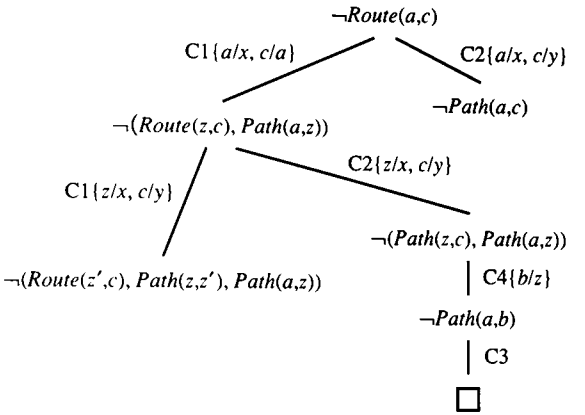

Figure 3.15 An SLD tree with an infinite branch

a path that neither fails nor succeeds, but instead creates an infinite series of increasingly large goals. Every attempt to resolve *Route*(*z*,*c*) with the first program clause results in the creation of a fresh variable *z'* and a further attempt to resolve *Route*(*z'*,*c*). Backtracking from this hopeless endeavour and resolving *Route*(*z*,*c*) against the second clause leads fairly quickly to a refutation that is never obtained.

An SLD tree appears to have three different types of path:

a. Successful paths that produce the empty goal and provide the constant instantiations required in the original goal.

b. Failing paths that terminate because the leftmost literal in the goal does not have a matching literal in a program clause.

c. Infinite paths that never produce a result.

Unfortunately, the mechanism we have described is equivalent to a depth-first search of the SLD tree, meaning that paths in a tree are explored to the maximum depth before any alternative to the right is considered. If, as in the above example, an SLD tree contains an infinite path to the left of a potentially successful path, the more productive path is never attempted. The technique fails to produce a result when one would have been expected and is therefore incomplete. An alternative breadth-first approach resolves the leftmost literal of a goal with every clause before proceeding to deeper levels and can be made complete. Unfortunately, the adoption of a breadth-first approach makes the implementation of backward-chaining systems rather more difficult.

We saw that forward chaining produces a large number of ground relations that will never be required, so backward chaining seemed an attractive alternative. Now we discover that a simple backward-chaining system depends on the order in which clauses occur in a program script. All of these problems can be avoided when more sophisticated algorithms are used.

EXERCISES 3.4

1. The clausal form of the definite program given in Exercise 1 of Exercises 3.3 is

   $\{p\}, \{s, \neg p\}, \{s, \neg r, \neg v\}, \{r, \neg s\}, \{t, \neg u, \neg s\}, \{v, \neg p, \neg r\}$

   Show that proposition $s \wedge v$ is true by resolving the dual of this expression in clausal form against the program clauses above. Similarly show that $r \wedge p$ is true.

2. Produce an SLD diagram tracing out the paths of attempts to prove propositions $p \wedge v$ and $s \wedge r \wedge v$ *true* in the definite program of Exercise 1 in Exercises 3.3.

3. A relation *Couple* has arguments naming a man and a woman; a relation *Mother* has arguments mother and child. A rule named *Father* relates fathers to their children through the *Couple* and *Mother* relations:

   Couple(*a*,*e*)
   Couple(*d*,*f*)
   Couple(*b*,*c*)
   Mother(*f*,*g*)
   Mother(*e*,*b*)
   Mother(*f*,*c*)
   Father(*x*,*z*) ← *Couple*(*x*,*y*), *Mother*(*y*,*z*)

   Draw SLD diagrams for the goals *Father*(*d*,*g*), *Father*(*d*,*x*) and *Father*(*x*,*y*).

CHAPTER FOUR

Prolog

4.1 PROLOG BASICS

At its simplest level, a Prolog system works by comparing facts in a database with a query called a goal presented at the prompt. Consider as an example some facts holding information about depositors and borrowers in a building society. A depositor fact contains an account number followed by the customer name and balance, whereas a borrower fact contains a loan number followed by a name and the loan amount:

```
depositor(123,smith,500).
depositor(234,brown,200).
depositor(345,patel,700).

borrower(735,jones,2000).
borrower(674,patel,6000).
borrower(865,evans,5000).
```

Collections of facts with the same name are called procedures or relations, and the individual instances are called clauses. Each fact consists of an identifier name followed by a number of arguments contained in brackets together with a terminating full stop. Prolog distinguishes between atomic and numeric constants by requiring atomic constants to begin with a lower case letter and numeric constants to begin with a digit. Thus, `depositor`, `borrower` and the names `smith`, `brown`, etc., are symbolic constants whereas the numbers are numeric constants. A string of characters beginning with a capital letter represents a variable, but when enclosed in single quotes, e.g. "Jones", the string is taken as an atomic symbol. Prolog will accept facts in which numeric and symbolic arguments have been entered inconsistently because it is not a typed language, so a great deal of care is required.

A database of information such as the facts above may be written into a text file then a Prolog system is instructed to "consult" that file. After consultation, Prolog is ready to answer queries presented at the query prompt. For example, a query might ask if the depositor relation contains an entry with the following specific arguments:

```
?-depositor(123,smith,500).
yes
```

Here the user supplies a goal at the question-mark prompt, and to achieve this goal, Prolog systematically attempts to match the goal name with a procedure name then to match the goal arguments with database arguments. This form of matching is a particularly simple form of a process called unification that checks goals against information in the database. If a goal can be unified with a clause in the database, the answer "yes" is obtained; otherwise the response is "no".

In practice the user is more likely to want to find information from a query rather than confirm known facts; this program finds the name and balance corresponding to a particular account number:

```
?-depositor(123,Name,Balance).
Name = smith
Balance = 500
yes;
no
```

Here a goal contains numeric constant `123` together with two variables distinguished from symbolic atoms by leading capital letters. Prolog again attempts to unify the goal with a fact in the database and, as before, matches the relation name and the first argument with constants. Since a variable can take any constant value, a unification is possible by setting `Name = smith` and `Balance = 500` and this satisfying instantiation of the two variables is reported. Any further depositor facts for account number 123 will be shown when the goal is resatisfied by typing a semicolon after the first result. As there is none, the system simply responds with the word "no".

Attempts at unification take place in order from the top to the bottom of the text file containing the clauses and satisfactory unifications are reported to the user in this order. A query with three variables, e.g. the query `depositor(X,Y,Z)`, would return all three arguments in the depositor clauses, producing three satisfactions in the order in which they occur in the text file.

Prolog can satisfy goals requiring more than one procedure, allowing questions like, "Is there anybody who is both a borrower and a depositor?"

```
?-depositor(A,N,B),borrower(L,N,S).
N = patel
```

A comma between the two parts of the goal acts as a logical "and", requiring truth in both relations: there must exist a triple `A,N,B` in the depositor relation and a

triple L,N,S in the borrower relation. Since N is the same variable in both relations, it is said to join the two procedures and only situations where the same name occurs in both are reported. One problem with the query above is that, in addition to reporting the name patel, Prolog also provides substitutions for A, B, L and S even though they are probably not required. A goal is only ever matched with facts having the same number of arguments, so arguments cannot simply be left out of goals. However, if some arguments are of no interest, they can be replaced by underscores:

```
?-depositor(_,N,_),borrower(_,N,_).
N = patel
```

and only instantiations of the named variables are reported. Underscores represent anonymous variables and might be seen as don't care or wild card variables. A semicolon represents logical "or" in a similar way to the logical "and" above, and we can discover if a named person is a customer, either a depositor or a borrower, with the following query:

```
?-depositor(_,smith,_);borrower(_,smith,_).
yes
```

In reality this goal is equivalent to two separate subgoals, one for depositor and one for borrower. It is true if depositor can be unified with a fact in the database or if borrower can be unified with a fact in the database. It is also true when both these subgoals are unified with clauses in their appropriate procedures, i.e. it is not an exclusive-or.

Rules may be added to the text file to provide a more permanant formulation of queries. For example, a rule to supply the balance for a depositor with a specific account number might be added to the facts in a database:

```
balance(AcctNo,Bal) :- depositor(AcctNo, _ ,Bal).
```

This type of clause is divided into a head on the left and a body on the right by the "if" symbol (:-) and the head of a rule is true if the body is true. When a text file of facts and rules is modified, the file must be "reconsulted" before the modification is incorporated. If the above rule is added to the initial database, the following query is possible:

```
?-balance(123,X).
X = 500
```

In practice this is very little help, except to demonstrate the use of a rule. A slightly more useful rule is possible for finding clients who are both depositors and borrowers:

```
both(Name) :- depositor(_,Name,_),borrower(_,Name,_).
```

which, after reconsultation, allows the following interaction:

```
?-both(X).
X = patel
```

Rules become more useful as queries become more complex. In particular, their usefulness increases when the heads of existing rules are used in the bodies of further rules, building up a series of references to rules.

Prolog contains a number of built-in predicates (BIPs) to carry out standard procedures such as numerical comparison. The "greater than" BIP may be used to find those depositors with balances greater than a certain value, e.g.

```
:- depositor(X,Y,Z), Z > 500.
X = 345, Y = patel
```

No introduction to Prolog would be complete without mentioning the relation `parent` and some of its family connections. Databases that include `parent` might contain the following facts and some comments distinguished by the leading % sign:

```
parent(anne,bob).      %examples of the parent relation
parent(john,jane).
female(anne).          %some gender relations
male(john).
```

They might also include rules to derive information from these simple facts, such as

```
mother(X) :- parent(X,Y),female(X).
```

indicating that a mother is at once a parent and a female. A variable `X` must be instantiated with the same value within a rule, but the value used in this rule has no connection with the use of the same variable name in another rule. Similar rules might be written for father, son and daughter.

Relation `grandparent` can be formed by conjugating `parent`:

```
grandparent(X,Z) :- parent(X,Y),parent(Y,Z).
```

A single variable `Y` now joins clauses in procedure `parent` to other clauses in the same procedure. A goal `grandparent(fred,tom)` generates subgoal `parent(fred,Y)` and Prolog systematically attempts to unify this subgoal with every fact in the procedure. Each time it instantiates `Y` to a constant, it searches the same procedure from top to bottom for the clause `parent(Y,tom)` and might report "yes" zero or more times.

A rule such as this can be used in the body of a further rule as follows:

```
grandma(X,Y) :- grandparent(X,Y),female(X).
sibling(X,Y) :- parent(Z,X),parent(Z,Y).
```

EXERCISES 4.1

1. A small library maintains the book number, author, title and price of its stock items in two Prolog relations such as

```
fiction(123,smith,dreaming,14.54)
non-fiction(467,jones,databases,20.30)
```

Enter six clauses for each relation into an editor and make Prolog consult the resulting file. Invent data for the relations, guided by the requirements of the following queries, then carry out the exercises at the prompt.

a. Find the price of any non-fiction book written by `patel`.

b. Show the author and title of all fiction books costing more than 20 currency units (cu).

Add rules to the file defining the following relations:

`books(B,A,T,P)`, describing all books, both fiction and non-fiction

`cheap(B,A,T,P)`, describing all books costing less than 10 cu

`authors(A)`, listing the names of all authors

`common(B1,B2)` where `B1` and `B2` are fiction and non-fiction book numbers of books with the same title

Use these rules to show the following information:

c. Names for the authors of all books costing less than 10 cu.

d. Author and title of books priced between 10 and 15 cu (inclusive).

e. Book numbers of fiction/non-fiction pairs with the same title when either one of them costs more than 30 cu.

4.2 FLAT TABLES

A flat table is a relation containing only simple atomic objects as opposed to structured objects such as functions and lists. Tables of this sort are used in relational databases together with a number of characteristic operations that define new tables or procedures from existing ones. For example, tables describing undergraduates, courses and their links can be written in Prolog as follows:

```
ugrad(123,wendy,f).          %students
ugrad(234,bill,m).
ugrad(345,norma,f).

course(c204,lisp,2,15).      %courses
course(c213,database,1,15).
course(c234,prolog,2,8).

link(123,c204,12).           %the connections
link(345,c213,10).
link(345,c234,11).
```

A `ugrad` relation contains arguments describing student number, name and gender, whereas a `course` relation has arguments for course number, title, semester (1 or 2) and credit value (15 for a full course, 8 for a half-course). A `link` relation contains student and course numbers from the previous two relations together with a grade point result (a figure between 0 and 16) for that combination of student and course. In relational database language the first two procedures are called entity relations and the third is called a relationship relation because its purpose is to connect other relations. Each of the procedures `ugrad`, `course` and `link` is equivalent to a base table of information in a relational database and is sometimes called the extensional database in Prolog.

A projection operation extracts arguments from one procedure to form a new procedure of fewer arguments. For example, a procedure of student names may be formed from the `ugrad` relation with the following rule:

```
names(Sn) :- ugrad(_, Sn, _).
```

so that a user can find the names of all undergraduates by typing `names(X)`. A very important point follows from this query: a user would never know whether the information obtained by typing `names(X)` had been obtained directly from facts or indirectly through rules. Instead of adding the rule above, we could have added a new procedure called `names` that contained only name facts. A virtual database obtained by adding rules in this way is called an intensional database.

A selection operation chooses tuples from a procedure on the basis of the values of its tuple arguments. Thus, instances of female undergraduates can be extracted from the `ugrad` procedure with the following rule:

```
f_ugrad(S,N,f) :- ugrad(S,N,f).
```

and the resulting `f_ugrad` relation has the same number of arguments but might have fewer instances. Since the third argument of `f_ugrad` must be `f`, it makes sense to combine the projection and selection operations, selecting females and projecting just the student number and name in one rule:

```
f_ugrad2(S,N) :- ugrad(S,N,f).
```

Only those tuples in which the third argument of ugrad is equal to `f` appear in the `f_ugrad2` relation. A selection based on a relation other than equality has to be made explicitly in the rule instead of placing the value in an argument position. For example, student number and course number combinations with grade points greater than 12 are revealed by the following rule:

```
high_marks(S,C) :- link(S,C,P),P>12.
```

4.2.1 Cartesian products and joins

The Cartesian product of two relations is a further relation containing ordered combinations of the arguments in the smaller relations. A Cartesian product can be formed from the `ugrad` and `link` relations through the following rule:

```
cprod(S1,N,G,S2,C,P) :- ugrad(S1,N,G),link(S2,C,P).
```

Remembering that Prolog works on a top-to-bottom and left-to-right basis, we should expect the `ugrad` variables' subgoal to be unified initially with the first `ugrad` fact appearing in the script file. The `link` subgoal is similarly unified with the first `link` fact appearing in the script, leading to a first satisfaction of the initial goal. Prolog then resatisfies the `link` subgoal as many times as possible before moving on to further instantiations of `ugrad`. Since there are three facts of each kind (in this example), the Cartesian product contains $3 \times 3 = 9$ entries, the first three of which are equivalent to

```
cprod(123,wendy,f,123,c204,12).
cprod(123,wendy,f,345,c213,10).
cprod(123,wendy,f,345,c234,11).
```

In practice a Cartesian product produces many more instances than are useful in any application. Two of the instances above contain information on different students, `123` and `345`, and are unlikely to be useful. Placing a common variable in the student number positions of both subgoals produces a special subset of the Cartesian product:

```
cprod2(S,N,G,S,C,P) :- ugrad(S,N,G),link(S,C,P).
```

but we now recognise that the duplicate copy of the student number in this relation is not necessary. When relations are joined through one or more attributes and duplicate copies of these attributes are removed from the resulting relation, a natural join is obtained. A join based on the student number is written as

```
join(S,N,G,C,P) :- ugrad(S,N,G),link(S,C,P).
```

A combination of this rule with the extensional database is equivalent to

```
join(123,wendy,f,c204,12).
join(345,norma,f,c213,10).
join(345,norma,f,c234,11).
```

but the rule is far more flexible. If the content of an extensional database changes, the rule uses modified data.

Joining operations are often accompanied by selections or projections. For example, the student numbers and names of female students who achieved more than 11 grade points in any course can be found with the rule

```
fover11(S,N) :- join(S,N,f,_,P),P > 11.
```

A join of the three relations in the extensional database may be written as

```
join2(S,N,G,C,P,T,L,V) :-
join(S,N,G,C,P),course(C,T,L,V).
```

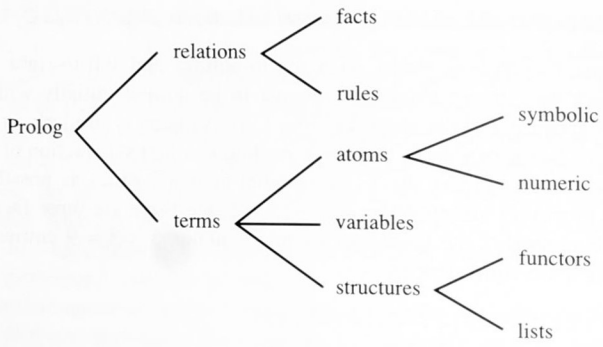

Figure 4.1 The parts of a Prolog program

A joined table can then be used to answer questions where information is spread over all three relations. For example, given a course title, provide the names of all students following that course:

```
members(Title,Name) :- join2(_,Name,_,Title,_,_,_,_).
```

Such a rule could have been written directly in terms of the extensional database relations with just enough variables to link the subgoal relations and produce the required results:

```
members2(Title,Name) :- course(C,Title,_),link(S,C,_),
ugrad(S,Name,_).
```

Prolog has the general form of first-order logic described in Chapter 2, but this form is restricted as described in Chapter 3 to give the language its operational features. Figure 4.1 shows that the objects making up a Prolog program are derived from the logic structure and are subdivided in much the same way. A program is composed of terms and relations; terms are subdivided into constants, variables and functions. Notice, however, that term constants are atoms in Prolog parlance and are further subdivided into symbolic and numeric forms. Functions are called structures and are subdivided into functors and lists. Functors look much the same as the functions described earlier and lists are a special functional notation. Terms are not evaluated and replaced by their denotations, as they would be in a functional language, so functors and lists have the more limited role of binding together other objects. Thus, the value of a function such as `square(4)` cannot be computed and its denotation `16` used to replace the function, because Prolog has a simple syntactic definition of equality. Nevertheless, functors are useful for binding together collections of objects into a single object, as in the date of birth functor `dob(12,june,1967)` or in the list structures `[1,2,3]` or `[mary, john,4]`. Since Prolog is not a typed language, a list may contain arbitrary atomic or structured objects. Similarly, any term may occur as an argument in a clause, but since we are limited to first-order logic, no clause can appear as an argument in another clause.

4.2.2 Tables with complex terms

Relational database systems accept the limitation to atomic terms in flat tables because this allows efficient implementation. Prolog allows more complex terms and therefore more expressive relations, but is rather less efficient in execution. However, some progress has been made in implementing databases that allow more complex terms and such systems could become commercially viable in the future. Consider as an example an extensional database containing information on winners of the Wimbledon men's singles tennis competition, with entries such as

```
mens_singles(player(stefan,edberg,dob(19,1,1966)),
[1988,1990],[becker,becker]).
```

Here a relation called `mens_singles` contains three arguments: a functor describing the player, a list of the years in which he won the title and a list of the defeated finalists in those years. A player functor contains the first name, surname and a further functor recording the date of birth of the player. Both lists are of variable length, depending on the number of times the player has won the competition, but the two have the same length. Entries only occur in the database when players have won the competition, so these lists cannot be empty. The `mens_singles` relation is still composed of symbolic and numeric atoms, but these atoms are gathered together into structures that allow them to be treated as a single entity when it would be useful to do so. For example, a simple query to the above database might be

```
?-mens_singles(X,Y,Z).
X = player(stefan,edberg,dob(19,1,1966))
Y = [1988,1990],
Z = [becker,becker]
```

and we see that complex terms rather than atoms have now been substituted for the variables. Complex terms might equally be substituted for anonymous variables, as in the following query that finds the years in which Borg won the men's singles competition:

```
?-mens_singles(player(_,borg,_),X,_).
X = [1976,1977,1978,1979,1980]
```

If we wish to know who won the competition exactly three times, we might write

```
?-mens_singles(player(Fname,Sname,_),[_,_,_],_).
```

Built-in predicates such as "greater than" may be included in a query to find surnames for all title winners born after 1950:

```
?-mens_singles(player(_,Sname,dob(_,_,Yr),_,_), Yr > 1950.
```

Obviously the use of complex terms makes the language far more flexible and expressive in comparison with relations limited to atomic terms.

Prolog terms may be tested for equality with the special built-in equality predicate, represented by the equals sign (=), e.g.

```
?-person(tom,13) = person(tom,13).
yes
```

If two terms can be unified, the appropriate substitution is made and reported:

```
?-player(stefan,edberg,X) = player(stefan,edberg,
dob(19,1,1966)).
X = dob(19,1,1966)
yes
```

Essentially the equals sign represents a special relation that is true when its two arguments are the same string of symbols or can be made the same by instantiation. A special strict equality is available (==) and this makes no attempt to unify the two terms; if they are not the same without unification, it fails.

EXERCISES 4.2

1. A small library contains the details of its stock and loans in the following relations:

```
book(123,smith,dreaming,14.54).
reader(894,palmer,finance).
serial(123,894,date(12,5,97)).
```

Book details remain as in Exercise 4.1 and the `reader` relation contains a reader number, reader name and reader department. Any book out on loan is connected to a reader number through the `serial` relation and is due back by the date shown in that relation.

Enter six clauses for each relation into a file through an editor and make Prolog consult this extensional database. Check the database by carrying out the following queries at the prompt:

a. Show names of all readers who currently have a book out on loan.

b. Show the book numbers and return dates of books on loan to `jones`.

Go back to the editor and define a relation that joins book and reader relations through the serial relation, then use this rule to answer the following queries:

c. Show title and price for all books currently on loan.

d. Show title for books currently on loan by `evans`.

e. Show title and return date for books on loan to `sales_dept`.

Define a relation called `later` that has two date arguments and is true if the second date is later than the first. Thus the goal `later(date(29,3,94), date(1,4,94))` elicits a "yes" response from the program.

f. Write a rule that allows a user to input a date and receive in return the reader names and titles of books on loan but due back by that date.

4.3 RECURSIVE QUERIES AND OPERATORS

The simple route diagram from the previous chapter is shown again in Figure 4.2 and used here as an example to illustrate recursive rules. Paths between various locations are shown in the form of a directed acyclic graph, a diagram of one-directional paths without any loops. Exactly the same information may be stored in a Prolog database as follows:

```
path(a,b).
path(a,c).
path(b,d).
path(d,e).
path(c,e).
```

and a query of the type "where can I go from d?" is an easily answered goal:

```
?-path(d,X).
X = e
```

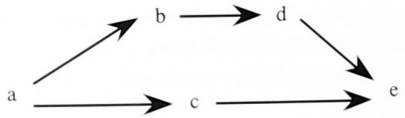

Figure 4.2 A directed acyclic graph

Single-step queries of this kind are not likely to be very useful. More often the user will want to know if a journey can be made between two points, perhaps with intermediate stages. Imagine that a user wants to know if a journey from b to e is possible and makes the following first attempt:

```
?-path(b,e).
no
```

followed by

```
?-path(b,X),path(X,e)
X = d
```

in which satisfaction is obtained. It might have taken many attempts with increasing numbers of intermediate points before a given goal succeeded. But some help could be provided by route rules such as

```
routeA(Start,Finish) :- path(Start,Finish).
routeB(Start,Finish) :- path(Start,X),path(X,Finish).
routeC(Start,Finish) :- path(Start.X),path(X,Y),
path(Y,Finish).
```

so that a user might try each rule in turn until one succeeds:

```
?-routeB(b,e).
yes
```

A more general approach follows from observing that a route consists of one path alone or one path followed by a further route, properties that are expressed in the rules

```
route(Start,Finish) :- path(Start,Finish).
route(Start,Finish) :- path(Start,X),route(X,Finish).
```

Suppose that the query `?-route(d,e).` is presented to Prolog with these rules and the above database. Goal `route(d,e)` is unified with the first clause of the `route` procedure, instantiating variables `Start` and `Finish` to constants `d` and `e`. Thus `route(d,e)` is true if `path(d,e)` is true and a systematic attempt to unify this subgoal with a database clause begins. Attempts to match goal against database fact proceed from the top of the text file and eventually succeed, causing Prolog to respond with the word "yes".

Suppose, however, that a query for a route with intermediate stages such as `?-route(b,e).` is presented. This goal is again unified with the head of the first rule and a systematic attempt to match subgoal `path(b,e)` with a database clause begins. This attempt fails, so variables `Start` and `Finish` in the second rule are instantiated to `b` and `e`, producing the following instantiation:

```
route(b,e) :- path(b,X),route(X,e).
```

Two subgoals now have to be satisfied, and Prolog works from left to right through the body of the rule attempting to unify subgoals with database clauses. Working from top to bottom through the text file, several attempts at unification are required before fact `path(b,d)` is unified with goal `path(b,X)` by instantiating `X` to `d`. This instantiation is transmitted to the following subgoal, and systematic attempts to unify `route(d,e)` with a database clause begin. Searching again begins at the top of the text file, leading to the first route rule and an instantiation of `d` and `e` for `Start` and `Finish`. A search for `path(d,e)` takes place and success is reported.

The second of the two route rules is said to be recursive because the head of the rule appears also in the body, causing it to make references to itself until a terminating condition occurs. Note that fresh variables are used each time a recursive reference occurs, even though the rule has a single name for each variable. The scope of variables in the head of the rule only extends to the body of that rule, not to other clauses used recursively by the procedure.

4.3.1 Operators and functors

Clauses might contain atoms, variables or functors as arguments, but Prolog functors serve a quite different purpose compared to the functions of a functional programming language. Although the two forms look the same, they behave differently because equality in Prolog means syntactic equivalence rather than denotational equivalence. Two terms are equal in Prolog if they are represented by the same string of symbols, whereas they are equal in a functional language if they can be reduced to the same canonical value. However, it is possible to define operators and provide Prolog rules that allow Prolog to discover canonical terms for complex terms expressed as relational arguments. As a particularly simple example, consider the fact

```
val0(and(true,true),true).
```

in which a fact of a relation called `val0` contains a functor and its denotation as first and second arguments. There is no difference in the form of a relation and a functor, so Prolog has to distinguish the two by their positions, thus in the example above, the `and` atom must be a functor because it occurs as an argument within a clause. Three more relations could be added to the one above, and the resulting procedure would completely define the effect of the `and` functor on two Boolean atoms. A particular functor may then be evaluated as

```
?-val0(and(true,false),X).
X = false
```

but queries that can be evaluated are limited to the four defined facts. However, four facts defining functor `and` in terms of constants may be replaced by just two rules using variables and taking advantage of Prolog's built-in operations, as in the `val` relation:

```
val(and(X,Y),true) :- val(X,true),val(Y,true).
val(and(X,Y),false) :- val(X,false);val(Y, false).
```

These rules express what we already know about the meaning of conjunction: the function `and(X,Y)` is true if `X` and `Y` are both true, but false if either `X` or `Y` is false or if both are false. If a database contained the rules above in a file together with facts

```
val(p,true).
val(q,false).
```

the following query response would be obtained:

```
?-val(and(p,q),X).
X = false
```

Relation `val` has the advantage over `val0` that it allows recursive references to decompose subterms within goal relations. A query in which a functor appears as an argument in another functor is possible:

```
?-val(and(p,and(q,r)),X).
X = false
```

allowing the evaluation of arbitrary Boolean expressions. Although this notation is perfectly correct, it suffers from two problems: the prefix use of `and` can become cumbersome in large expressions and it does not provide the usual precedence rules for propositional operators, i.e. it does not give conjunction priority over disjunction. Prolog allows users to define functions to be operations through operator definitions and, in the case of conjunction above, we could define an operator labelled `&` as follows:

```
:- op(600,xfy,&).
```

An operator declaration, sometimes called a directive, must precede any attempt to use the operator in a program. It consists of the atom name `op` with three arguments, the last of which is a symbol for the operator being defined.

The first argument is a number (usually between 1 and 1200) that defines a precedence level for the operator, lower numbers representing operators that bind most tightly to their arguments. For example, a Boolean expression `p v q & r` is assumed to represent $p \vee (q \wedge r)$. To ensure that the `&` operation is applied first, it is defined with a lower precedence number (giving it a higher precedence) than the `v` operation.

The second argument of the operator definition defines the position of the operation symbol in relation to its own arguments. Operator arguments are represented by the letters `x` and `y`, and the operator itself is represented by the letter `f`. Prefix, infix and postfix operators are then defined with the notations `fxy`, `xfy` and `xyf`. Argument `x` has a lower priority than `y`, so an operator defined in this way associates to the right when the arguments are themselves functors. Notice that an operator definition tells us nothing about the "meaning" of an operator, i.e. what it denotes, so this information has be be added with rules:

```
val(X & Y,true) :- val(X,true),val(Y,true).
val(X & Y,false) :- val(X,false);val(Y,false).
```

Except for the replacement of the prefix `and` symbol by an infix `&` symbol, these rules are the same as those above, but now the abbreviated symbol can be used at the prompt:

```
?-val(p & q, Z).
Z = false
```

Operator definitions for negation, disjunction and implication can be made in the same way:

```
:- op(500,fy,~).
:- op(700,xfy,v).
:- op(800,xfy,->).
```

Since all these symbols have to be typed on a normal keyboard, they are slightly different from equivalent symbols used earlier. Conjunction is represented by the symbol `&` because the usual $\wedge$ is not available on keyboards, though it would be possible to define $\wedge$ with two keystrokes. A negation function is represented by the tilde `~`, disjunction by a lower case letter `v` and implication by a combination of a minus sign with the chevron `->`.

Rules can be written for disjunction and implication using the operators defined above:

```
val(X v Y,true) :- val(X,true);val(Y,true).
val(X v Y,false) :- val(X,false),val(Y,false).
val(X -> Y,true) :- val(X,false);val(Y,true).
val(X -> Y,false) :- val(X,true),val(Y,false).
```

The valuation of a disjunction is `true` when either `X` or `Y` is `true`; it is `false` only when both arguments are `false`. Implication is slightly more difficult to follow; its valuation is `true` when the first argument (the antecedent) is `false` or when the second (the consequent) is `true`, or when both situations occur together. Only a `true` antecedent with a `false` consequent makes an implication `false`. Negation is expressed as a simple inversion:

```
val(~X, true) :- val(X,false).
val(~X, false) :- val(X,true).
```

Having entered these rules in the database, it is instructive to follow the evaluation of a query. Suppose that the following goal is presented to the database:

```
?-val((p v ~q) & (~p v q),true).
```

This proposition was written earlier as $(p \vee \neg q) \wedge (\neg p \vee q)$ and its semantic tableau is shown in Figure 1.3. A trace of the way in which Prolog attacks this problem follows the form of the semantic tableau and provides some insight into its search mechanism. The goal above is matched with the head of the `true &` rule, so that both subgoals have to evaluate to `true` in order to make the goal itself `true`. Since both have to be `true`, they are shown one directly below the other, leftmost first, in Figure 4.3. It is important that the leftmost subgoal of the rule appears highest in the diagram because subgoals are discharged in order from top to bottom. The goal in line 2 is now matched with the head of the `true v` rule, creating two further subgoals that have to be satisfied. A split in the diagram indicates that only one of these subgoals has to be satisfied in order to satisfy the parent goal. Now subgoal `val(p,true)` has to be satisfied, but it cannot be unified with a rule head, so the following possibilities arise:

a. No matching fact occurs in the database, so Prolog "backtracks" to its parent clause in order to check alternatives, in this case it would try `val(~q,true)`.

b. A matching fact does occur and Prolog continues with the current sequence of satisfactions, in this case unifying the subgoal in line 3 with a rule.

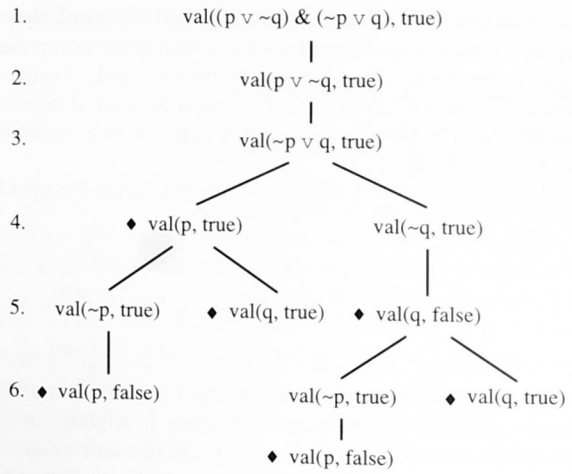

Figure 4.3 A Prolog search tree: backtracking points are marked with a diamond

Each backtracking point is marked with a diamond in the diagram; they are the make or break points where Prolog either finds a suitable fact in its database or it does not. Hopefully, `val(p,true)` and `val(p,false)` would not occur together in the database, so Prolog would have to backtrack from one of them. If the database contains both `val(p,true)` and `val(q,true)`, the bottom of the search tree is reached and the original goal is satisfied.

Whenever a branch occurs in a diagram such as Figure 4.3, it is the left-hand side that is explored first. If the system is unable to match a goal with the head of a rule or a fact, it backtracks to the point where it branched and tries the alternative route. This is called a depth-first strategy because the tip of the leftmost branch is reached before any alternative routes are explored. If the database contained facts `val(p,true)` and `val(q,true)`, the tableau would be explored to the left-hand tip, but `val(p,false)` would fail, causing backtracking to line 4 and an attempt to match the subgoal `val(q,true)` with a fact, which succeeds. Prolog reports this success, but continues to look for further matches, backtracking at each of the marked points.

An exclusive-or operation labelled `@` might be defined by the following program instructions:

```
:-op(700,xfy,@)
val(X @ X,true).
val(X @ Y,false) :- X \== Y.
```

EXERCISES 4.3

1. Create a database of parent facts of the type `parent(john,mary)`. Write a procedure called `ancestor` that succeeds if the person in the first argument

position is ancestor of the person in the second argument position. Add a relation `female` to the database that includes all the females in either argument positions of the parent relation. Write a modified version of `ancestor` called `maternal` that relates a person in the second argument position through a series of female parents to a female in the first argument position.

2. Declare operator symbols and write procedures that implement the *nand* (|), *nor* (↓) and *imp* operations described in Chapter 1. Enter atomic valuations `val(p,true)`, `val(q,false)` and `val(r,false)` into the database, then evaluate the following expressions, in which ⊕ is *niff* (exor) and ¬ is *not*; both will have to be translated to keyboard symbols.

   $p \mid q \oplus r \mid r$
   $p \mid q \downarrow \neg p \mid r$

3. Draw diagrams similar to Figure 4.3, showing the search path when the following goals occur at the prompt:

```
val(p & ~q v ~p & q,true)
val((p & ~q) -> r ,true)
```

4.4 OPERATORS AND ARITHMETIC

In the logical expressions above, we chose to write function `and(X,Y)` in the form `X & Y`, having defined an infix operator `&` with a specific precedence level. Remember that the operator definition did not explain how the result of the operation is deduced from its arguments. Similar considerations apply to the arithmetic operators used in Prolog except that both the operator definitions and their meanings are predefined in a standard environment.

Armed with the knowledge that the necessary operator definitions are built into the Prolog system, a naive user might attempt the following calculation at the prompt:

```
?- X = 2 + 3 * 4.
X = 2 + 3 * 4
```

Such a user might have hoped that Prolog would evaluate the expression on the right then report an equivalent value for `X`. Unfortunately, the concept of equality in Prolog is that of simple syntactic equivalence, not of denotational equivalence, and in this light the response above is perfectly correct. The user has asked Prolog what sequence of symbols a variable `X` would need to represent in order to be equal to `2 + 3 * 4` and the system has obliged with the appropriate sequence. Prolog recognises arithmetic symbols as syntactic infix operator symbols and instantiates variables to make expressions equivalent:

```
?-X + 2 = 3 + Y.
X = 3
Y = 2
```

Arithmetic operations are so useful that they are allowed and activated by special symbols. In particular, the calculation intended above could be carried out with the `is` operator at the `?` prompt:

```
?-X is 2 + 3 * 4.
X = 14
```

Operator `is` represents a form of denotational equality that causes the evaluation of the arithmetic expression on its right-hand side and the substitution of the result into the variable `X`. In fact, the `is` keyword evaluates the expression then unifies the result with the term on its left. If the other atom is a variable, the unification amounts to a simple substitution, as in the example above; but if it is a numeric constant, the operation amounts to an equivalence test. For example, the following query results in a successful unification:

```
?- 14 is 2 + 3 * 4.
yes
```

All atoms in a numerical computation must be numeric atoms. The available range of arithmetic operations varies between implementations but always includes the familiar operations: +, –, *, /, mod, div. Arithmetic operators associate from left to right, so the expression `4 - 3 - 2` evaluates as (4 – 3) – 2 rather than 4 – (3 – 2). This is a direct consequence of the `x` and `y` arguments in the operator definition of the subtraction symbol:

```
op(500,yfx, -).
```

The keyboard symbol for a minus sign is a hyphen (-). Arguments `x` and `y` can themselves be expressions but the principal functor of `x` must have a lower precedence than `f`, whereas the principal functor of `y` might be either lower or equal in precedence to `f`. This means that only left association is allowed. Precedence numbers also ensure the usual evaluation priorities, i.e. an expression of the form `2 + a * b` is understood to represent 2 + (a * b).

4.4.1 Relational operators

An intended numerical comparison of the type

```
?-4 * 3 = 2 * 6.
no
```

again fails because the strings of symbols are different. Prolog is concerned with syntactic equality, so it does not recognise the fact that `4 * 3` and `2 * 6` denote the same value. However, numerical comparisons of arithmetic expressions are so useful that they too are included in the language by adding some special relational symbols. Operations of this kind employ a different equality symbol:

```
?- 4 * 3 =:= 2 * 6.
yes
```

This new symbol causes the numerical evaluation of the two sides before a comparison is made. Several other numerical relation operators are defined such that they force the evaluation of two arguments then perform the numerical comparison, reporting "yes" and "no" as though the answer were a Boolean result:

`>`	greater than
`<`	less than
`=<`	less than or equal to
`>=`	greater than or equal to
`=\=`	numerical inequality
`=:=`	numerical equality

Built-in relational operators are very useful in writing a procedure to find the larger of two numbers `X` and `Y`:

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y > X.
```

allowing the following interaction at the prompt:

```
?-max(2,5,B).
B = 5
```

Although Prolog is a syntactic reasoning language "without equality", it uses a suprisingly large number of different equality symbols and we need to be sure of their meaning. Predicates `=` and `\=` represent syntactic equality or inequality respectively, allowing substitutions to achieve a result. Predicates `==` and `\==` represent strict equality and inequality respectively; they do not permit substitutions in order to obtain a result. Finally, Prolog's numerical comparisons force evaluations of the arguments then compare the values denoted by the arguments.

4.4.2 Arithmetic in recursive procedures

Distances could be incorporated into some facts derived from Figure 4.3 as follows:

```
path(a,b,8).
path(b,d,7). etc.
```

A rule could then be written to show not only that routes are possible, but also to provide the distance of a given journey:

```
route2(X,Z,D) :- path(X,Z,D).
route2(X,Z,D) :- path(X,Y,D1),route2(Y,Z,D2),D is D1 + D2.
```

Procedure `route2` works in the same way as `route` itself, but adds the path distance of each individual step to the distance of the remaining route.

The factorial of a given natural number is the product of that number with all natural numbers smaller than itself; factorial 5 is equal to $5 \times 4 \times 3 \times 2 \times 1 = 120$. In order to express a factorial in recursive form, we note that the factorial of any number n is equal to n multiplied by the factorial of $n - 1$. From this simple observation comes the following recursive Prolog program:

```
fac(1,1).
fac(X,Y) :- X1 is X - 1,fac(X1,Y1),Y is X * Y1.
```

and factorial 1 is easily found to be 1 by pattern-matching the goal `fac(1,R)` with the first clause. Goals containing larger integers such as `fac(4,R)` can only be matched with the second clause, the number being instantiated into variable `X` and the result `R` sharing with rule variable `Y`. It is instructive to follow the steps required to produce a result from the goal `?-fac(4,R)` when it is unified with the rule head:

```
fac(4,R) :- 3 is 4 - 1,fac(3,Y1),R is 4 * Y1
fac(3,Y1) :- 2 is 3 - 1,fac(2,Y2),Y1 is 3 * Y2
fac(2,Y2) :- 1 is 2 - 1,fac(1,Y3),Y2 is 2 * Y3
fac(1,Y3) :- fac(1,1)
```

at which point the series of recursive references is terminated. The first arithmetic subgoal is satisfied and produces a decremented integer that is shared with an argument in the second subgoal, a recursive call to the factorial procedure. Fresh variables are introduced in each step but they cannot be assigned values until the final terminating stage is reached. Even then it is only the last variable (`Y3`) that is assigned a value of 1. This allows `Y2` to be calculated, then `Y1` and finally `R` as follows:

```
fac(2,2) :- 1 is 2 - 1,fac(1,1),2 is 2 * 1
fac(3,6) :- 2 is 3 - 1,fac(2,2),6 is 3 * 2
fac(4,24) :- 3 is 4 - 1,fac(3,6), 24 is 4 * 6
```

An alternative approach to the traditional factorial computation is provided by the following accumulator procedure:

```
fac2(N,R) :- faux(N,1,R).
faux(1,Y,Y).
faux(N,A,S) :- N1 is N - 1,A1 is N * A,faux(N1,A1,S).
```

in which an auxiliary procedure `faux` has an argument that accumulates the value of the final result as it proceeds, avoiding the need to return to previously unsatisfied subgoals. Again it is instructive to follow the steps required to find factorial 4:

```
fac2(4,R) :- faux(4,1,R).
faux(4,1,R) :- 3 is 4 - 1,4 is 4 * 1,faux(3,4,R)
```

```
faux(3,4,R) :- 2 is 3 - 1,12 is 3 * 4,faux(2,12,R)
faux(2,12,R) :- 1 is 2 - 1,24 is 2 * 12,faux(1,24,R)
faux(1,24,24)
fac2(4,24)
```

Euclid's method for finding the greatest common divisor of two numbers is implemented by the following Prolog procedure and is by nature an accumulating algorithm:

```
gcd(X,X,X).
gcd(X,Y,Z) :- X < Y,Y1 is Y - X,gcd(X,Y1,Z).
gcd(X,Y,Z) :- X > Y,X1 is X - Y,gcd(X1,Y,Z).
```

If the two numbers are the same, then the largest number that divides into both without remainder is the number itself. If the numbers are different, the larger number is reduced by the value of the smaller number until the two are equal, and the greatest divisor is that number. The accumulative nature of this procedure is shown in the following trace:

```
gcd(12,15,R) :- 12 < 15, 3 is 15 - 12,gcd(12,3,R)
gcd(12,3,R) :- 12 > 3, 9 is 12 - 3, gcd(9,3,R)
gcd(9,3,R) :- 9 > 3, 6 is 9 - 3, gcd(6,3,R)
gcd(6,3,R) :- 6 > 3, 3 is 6 - 3, gcd(3,3,R)
gcd(3,3,3)
```

EXERCISES 4.4

1. Define a procedure that takes a pair of numbers and returns a number expressing the first as a percentage of the second, e.g.

   ```
   ?-percent(3,4,X)
   X = 75.00
   ```

2. Write a procedure that accepts the radii of two circles on the same centre and calculates the area of the space between the two circles.

3. Simple and compund interest are calculated from the formulas $p(1 + ry/100)$ and $p(1 + r/100)^y$ in which p is the principal (the amount of money invested), r is the percentage annual rate of interest and integer y is the number of years of the investment. Write Prolog procedures to calculate the simple and compound interest of money invested. Use both procedures to calculate the value of 500 currency units invested at 5.5% per annum for seven years. (The ISO standard symbol for the exponential in Prolog is `**`.)

4. Define procedures `div60` and `mod60` that reveal the number of times 60 will divide into a given integer and the remainder when this is done. (The standard symbols for `mod` and `div` are `mod` and `//`.) The interaction at the prompt should appear as follows:

```
?div60(150,X)
X = 2

?mod60(150,X)
X = 30
```

5. Define a procedure that converts a whole number of seconds into a triple of hours, minutes and seconds:

```
?-convert(4350,X)
X = hms(1,12,30)
```

6. Define an infix, arity-two operator symbol # that always yields the smaller of its two arguments:

```
?- val(4 # 7,X).
X = 4
```

7. A directed acyclic graph such as that described in Section 4.3 might carry cost information as follows:

```
path(a,b,20).
path(a,c,35).
path(b,d,43).
path(d,e,26).
path(c,e,44).
```

Write a Prolog procedure that finds the total cost of each possible route from one point to another in the graph. Test this routine by finding the costs of two possible routes from a to e.

4.5 LISTS

Lists are structures provided to hold sequences of data objects, though in Prolog these objects do not have to be of the same type. Prolog is not a typed language, so a single list might contain any mixture of atomic and structured objects. An empty list appears simply as a pair of square brackets `[]`; lists with increasing numbers of elements are represented in one of these forms:

`[]`	`[]`
`[a]`	`.(a,[])`
`[b,a]`	`.(b,.(a,[]))`
`[c,b,a]`	`(c,.(b,.(a,[])))`
regular form	cons form

Lists are commonly used in the regular form shown in the left column, so it is not immediately obvious that they consist of a series of functors applied to an empty

list. A list of one element is really that one element bound to an empty list by the constructor (cons) functor, shown as a dot in the prefix position. Larger lists consist of further elements each added by an application of cons so that every list expressed in regular form has an equivalent functor form. A small goal presented to the system confirms the equality of the regular and cons forms of a given list:

```
?-[a,b] = .(a,.(b,[ ])).
yes
```

This small query shows not only that Prolog accepts both forms of the list as equivalent, but also that it can test pairs of lists for syntactic equality, because the query

```
?-[a,b] = [b,a].
no
```

is equivalent to the comparison of two complex functors

```
?- .(a,.(b,[ ])). = .(b,.(a,[ ])).
```

Variable substitutions are allowed to achieve a unification

```
?-[2,Y,6] = [X,4,Z].
X = 2 Y = 4 Z = 6
yes
```

Lists are often processed by repeatedly removing the leftmost element, the head, for examination or computation, reversing the action of the cons operator. This process is aided by a special notation `[H|T]` in which the head element `H` is shown separated from a list of all the remaining elements by a vertical bar. Two relations

```
hd([H|T],H).
tl([H|T],T).
```

could be placed in the database and would allow the following queries:

```
?-hd([4,6,3],X).
X = 4

?-tl([mary,john,tex],X).
X = [john,tex]
```

Notice that the head is an element whereas the tail of a list is itself a list. The sum of all the elements in a numeric list is computed by the following recursive procedure:

```
addup([ ],0).
addup([H|T],X) :- addup(T,X1),X is X1 + H.
```

which implements two obvious truths: the sum of the elements in an empty list add up to 0; the sum of any other list is found by adding the value of the head element to the sum of its tail elements. The following trace of an invocation might help to show the recursive nature of this procedure:

```
addup([3,4,5],R) :- addup([4,5],R1),R is R1 + 3
addup([4,5],R1) :- addup([5],R2),R1 is R2 + 4
addup([5],R2) :- addup([ ],R3),R2 is R3 + 5
```

at which point the repeated calls are terminated because `addup([ ],R3)` can be unified with the first clause, setting `R3` to 0. Variables `R1`, `R2` and `R3` are fresh variables created for each recursive call, and they acquire values as the process returns to the top level:

```
addup([5],5) :- addup([ ],0),5 is 0 + 5
addup([4,5],9) :- addup([5],5),9 is 5 + 4
addup([3,4,5],12) :- addup([4,5],9),12 is 9 + 3
```

A similar but simpler procedure can be used to count the number of elements in a list:

```
len([ ],0).
len([_| T],X) :- len(T, X1),X is X1 + 1.
```

Any empty list has a length 0 whereas every non-empty list has a length one greater than the length of its tail. The second clause is applied until the list is empty, incrementing a counter every time a head element is removed, then the recursive procedure is terminated by the first clause. A goal could then be presented at the ? prompt:

```
?-len([tom,dick,34,harry],X).
X = 4
```

A specific element is a member of a list if it is the head element of that list or it occurs in the tail, leading to the following pair of rules:

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

Conversely, a list is free of a certain element if that element is not the head element and the element does not occur in the tail:

```
free(X,[ ]).
free(X,[H|T]) :- X \== H,free(X,T).
```

A procedure such as `addup` can be written in an alternative accumulator form described earlier in connection with the factorial evaluation. The new procedure includes an auxiliary accumulator relation `addacc`:

```
addup2(L,N) :- addacc(L,0,N).
addacc([ ],A,A).
addacc([H|T],A,N) :- A1 is A + H,addacc(T,A1,N).
```

Using this procedure, the trace of our addition example is

```
addacc([3,4,5],0,N) :- 3 is 0 + 3, addacc([4,5],3,N)
addacc([4,5],3,N) :- 7 is 3 + 4, addacc([5],7,N)
```

```
addacc([5],7,N) :- 12 is 7 + 5, addacc([ ],12,N)
addacc([ ],12,12)
```

Two lists can be checked for a common element by systematically checking whether elements of the first list are members of the second:

```
common([X|L],M) :- member(X,M).
common([X|L],M) :- common(L,M).
```

Prolog allows more than one element to be placed on the left of the vertical bar so that the lists `[c,b|[a]]` and `[c|[b,a]]` are equivalent. This feature is useful in a procedure to discover whether two given symbols occur side by side in a list:

```
next(X,Y,[X,Y|T]).
next(X,Y,[H|T]) :- next(X,Y,T).
```

4.5.1 List-producing procedures

Each of the above procedures produces a single number or Boolean result from the inspection of a list of items. Now we examine a number of procedures that both accept and produce lists. First of all, an operation called `append` combines (concatenates) two lists into one through the following procedure:

```
append([ ],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Given two lists, `append` concatenates them together as follows:

```
?-append([a,b,c],[p,q,r],R)
R = [a,b,c,p,q,r]
```

but it will also accept a list in the third argument position and return each of the possible sublists that could be concatenated to produce that list:

```
?-append(L1,L2, [a,b,c,d]).
L1 = [a], L2 = [b,c,d];
L1 = [a,b], L2 = [c,d]; etc.
```

Non-deterministic behaviour of this kind is sometimes seen as a major feature of the Prolog language, but in practice not many procedures behave in this way.

Operation `append` may be used as a subrule in a procedure to reverse the elements of a list:

```
rev([ ],[ ]).
rev([X|L],M) :- rev(L,N),append(N,[X],M).
```

Elements are repeatedly detached from the head of a list and appended to the right of a reversed tail list. An accumulator approach that does not require the use of `append` might have been used to achieve the same result.

Two important procedures called `map` and `filter` are used in Prolog in the same way that they are used in many other languages: `map` repeatedly applies some operation to every element of a list; `filter` selects specific sublists from an initial list. An operation called `oper` might simply multiply numbers by 3 as in

```
oper(X,Y) :- Y is X * 3.
```

then procedure `map` applies this operation to every element in a list:

```
map([],[]).
map([H1|T1],[H2|T2]) :- oper(H1,H2),map(T1,T2).
```

Similarly, a filtering condition might simply choose values over 12:

```
cond(X) :- X > 12.
```

then a sublist of elements that satisfy this condition is selected by procedure `filter`:

```
filter([],[]).
filter([H|T1],[H|T2]) :- cond(H),filter(T1,T2).
filter([H|T1],T2) :- filter(T1,T2).
```

4.5.2 Ordering and sorting list elements

List sorting has to be based on some defined element order, but it is desirable to write sorting procedures independently of the test for element order. Thus a simple relation called `less` might be defined as the simple numeric relation

```
less(X,Y) :- X < Y.
```

or perhaps as an ordering based on two date of birth functors:

```
less(dob(D1,M1,Y1),dob(D2,M2,Y2)) :- Y1 < Y2 ;
                                     Y1 = Y2,M1 < M2;
                                     Y1 = Y2,M1 = M2,D1 < D2.
```

It is possible to imagine many different procedures for ordering pairs of functors based on combinations or sums of argument values, but such procedures should be separated from list-sorting procedures themselves. As a result, the sorting procedures described below are generic, i.e. they work for elements of any defined type, provided an ordering such as the one above is available.

Elements from a numerical list can be divided into two sublists according to whether they are greater or less than a given number. Thus, numbers less than 12 and greater than 12 are separated by a procedure `split`:

```
?-split(12,[9,16,3,14,7],X,Y)
X = [9,3,7] Y = [16,14]
```

and the definition making this result possible is as follows:

```
split(N,[ ],[ ],[ ]).
split(N,[H|T],[H|A],B) :- less(H,N),split(N,T,A,B).
split(N,[H|T],A,[H|B]) :- less(N,H),split(N,T,A,B).
```

An empty list divides into two empty sublists whereas a non-empty list shares its head element with one of the two sublists; the choice of sublist depends on the relation of the head element to number N. Repeated applications of the two recursive clauses eventually produces a tail depleted to an empty list and the first clause then applies.

This split procedure clearly goes some way towards sorting a list of elements and could be used as an auxiliary definition to partially sort elements in a single list. The two sublists obtained from split could be rejoined with append to give a rule called partsort:

```
partsort([H|T],R) :- split(H,T,A,B),append(A,[H|B],R).
```

If the original head element was either the largest or smallest element in the list, then the result of partsort is the same as the original one, so nothing will have been achieved. In all other situations some degree of sorting will have occurred, though the elements of the two sublists remain in the same order as in the original. If partsort is used on a list similar to the one above, we obtain

```
?-partsort([12,9,16,3,14,7],X)
X = [9,3,7,12,16,14]
```

so the original head element separates elements smaller and larger than itself. If unsorted lists either side of the head element are themselves partsorted in this way, and the algorithm repeated for further sublists until empty lists are obtained, a fully sorted list is obtained. This technique of sorting lists is called a quicksort and is defined in Prolog as follows:

```
qsort([ ],[ ]).
qsort([H|T],R) :-split(H,T,A,B),qsort(A,A1),qsort(B,B1),
                                        append(A1,[H|B1],R).
```

A procedure called divide produces two sublists of equal or nearly equal size from a list in the first argument position:

```
?-divide([a,b,c,d,e],X,Y).
X = [a,c,e]
Y = [b,d]
```

and the fact that the resulting sublists contain alternating elements from the original list indicates how the procedure has been defined:

```
divide([ ],[ ],[ ]).
divide([X],[X],[ ]).
divide([X,Y|L],[X|M],[Y|N]) :- divide(L,M,N).
```

The first two elements of a list become the head elements of two result lists, and the rest of the initial list is divided between the two sublists. This recursive removal of pairs of elements eventually generates a call to divide with an argument list containing either one or no elements and the series of recursive calls is terminated.

Suppose we have two ordered lists that have to be merged into a single ordered list as follows:

```
?-merge([2,5,8],[3,7,9]),W).
W = [2,3,5,7,8,9]
```

so the lists in the first and second argument positions are merged to give the ordered list in the third position. Five defining clauses seems generous:

```
merge([ ],L2,L2).
merge(L1,[ ],L1).
merge([X|L1],[X|L2],[X|L]) :- merge(L1,[X|L2],L).
merge([X|L1],[Y|L2],[X|L]) :- less(X,Y),merge(L1,[Y|L2],
L).
merge([X|L1],[Y|L2],[Y|L]) :- less(Y,X),merge([X|L1],L2,
L).
```

but the first two just define the result of merging an empty list with another list. The other three clauses are related to the three ways in which head elements `X` and `Y` may be related:

a. If the head elements of the two lists to be merged are the same, one copy of the element is "consed" onto the result list and the other is passed to the second argument in a recursive call.

b. If the head element of the first list is less than the head element of the second list, it is "consed" onto the result list and its tail is merged with the whole of the second list.

c. If the head element of the second list is less than the head element of the first list, the head of the second list is taken and its tail is merged with the first list.

The ability to merge two ordered lists into a larger ordered list could be used as the basis of a list-sorting routine. Lists containing just one element are certainly in order, so merging two such lists must produce an ordered list of two elements. These lists may in turn be merged to give ordered lists of four elements, and so forth, until a list of any desired size is obtained. The procedure to sort a list with this approach is quite simple: just break down a given list into single-element lists by repeatedly using `divide`, then merge the resulting single-element lists:

```
msort([ ],[ ]).
msort([X],[X]).
msort([X,Y|L],M):-divide([X,Y|L],L1,L2),
                  msort(L1,M1),msort(L2,M2),
                  merge(M1,M2,M).
```

Two clauses specify the obvious truth that single- or zero-element lists are already sorted. A third clause contains an argument `[X,Y|M]` that can only be unified with a goal argument containing at least two elements, which explains why this style of list description is used in the head of the rule. A list presented as a goal is divided into sublists until each sublist has fewer than two elements, then `merge` is invoked.

4.5.3 Sets implemented as lists

Sets are powerful, high-level constructs but they are difficult to implement in programming languages. One solution to the difficulty is to use lists to carry a representation of the set, disregarding the order of the elements in the list. Although list `[a,b,c]` is strictly different from list `[b,c,a]`, we can write procedures that are independent of the ordering of the elements. However, sets should not contain multiple occurrences of the same element and, as a first step towards defining set operations, we need a procedure to delete every occurrence of a given element from a list:

```
delete(X,[ ],[ ]).
delete(X,[X|T],Z) :- delete(X,T,Z).
delete(X,[H|T],[H|Y]) :- X \== H,delete(X,T,Y).
```

Deleting a specified element from an empty list leaves an empty list. If the list does contain elements then either the element to be deleted occurs as the head element or it does not. In one case the list element is included in the result list and in the other it is not.

Armed with a procedure that deletes all copies of a specified element from a list, we then use it to delete all duplicate copies of the head element in a given list:

```
mkset([ ],[ ]).
mkset([H|T],[H|X]) :- delete(H,T,Y),mkset(Y,X).
```

Here the head element of a list presented as the first argument is made the head element of the second list, but all further occurrences of this element in the tail are deleted to produce a new list `Y`. The head of this list is in turn subjected to the same procedure to produce list `X`. Recursive applications terminate when every head element has been treated in this way. In use this procedure is very simple:

```
?-mkset([a,b,a,c],X).
X = [a,b,c]
```

Set union is an operation that produces a result set containing every distinctive element from two set operands, but with only one copy of any element that occurs in both operands. In the following procedure two lists in the first and second argument positions have a union given by the list in the the third position:

```
union([ ],S2,S2).
union([X|S1],S2,R) :- member(X,S2),union(S1,S2,R).
union([X|S1],S2,[X|R]) :- free(X,S2),union(S1,S2,R).
```

Each head element from the first argument list is made the head element of the third list, unless it is also contained in the second list, in which case it is left off the third list. One head element is removed from the first list with each recursive call until that list becomes empty, then all elements of the second list are transferred to the third list. An interaction at the ? prompt might now proceed as follows:

```
?-union([a,g,w,y],[w,a,d,f],X).
X = [g,y,w,a,d,f]
```

and the order of the elements in this resulting list can be related to the procedure above. Elements of the first list that do not occur in the second list will appear first, followed by all the elements from the second list.

Intersection is a set operation that extracts just the common elements from two set operands; it may be implemented as follows:

```
intersect([ ],S2,[ ]).
intersect([X|S1],S2,[X|R]) :-member(X,S2),intersect(S1,
S2,R).
intersect([X|S1],S2,R) :-free(X,S2),intersect(S1,S2,R).
```

Members of the first set are now included in the result set only if they also occur in the second set, and it is clear from the first clause that the second set itself is not copied over when the procedure terminates.

EXERCISES 4.5

1. Write a Prolog procedure that relates a list of individual character symbols and a number indicating the number of symbols in the list that are vowels. The program may give further irrelevant answers after giving an initial correct response.

2. Write a Prolog procedure that accepts a list containing both positive and negative numbers and returns a list of absolute numbers obtained from the signed numbers.

3. Define a procedure that relates a list of positive and negative integers to another list containing the same integers, but in which all positive numbers appear in the list before any negative number. Both positive and negative sublists should retain the order of the numbers in the original list.

4. Write a Prolog procedure that accepts a list of lists and returns a list of numbers indicating the lengths of each sublist in the original list.

5. Write a procedure that takes a list of integers and produces an average value for the list.

6. A functor contains the name of a child, his or her date of birth and a list of scores out of 10 for the workbooks completed. A list of such functors is contained

in a relation called `kids` and identified by the first argument as a list of details for a first group of childen, `group1`:

```
kids(group1, [f(tom,dob(14,3,89),[4,7,5]),
f(lee,dob (7,12,88),[3,6]), ... ])
```

Create a relation `kids` containing the details of six children and define procedures that produce the following lists:

a. Functors containing name and date of birth, without the grades.

b. Functors containing name and number of workbooks completed for each child name.

c. Functors containing the name and average mark for each child.

d. Functors containing name and age of each child when the current date is included in the query.

e. Names of children who are older than the average of the list.

f. Names of children in increasing order of total marks in the list.

Create a second relation that contains information for a second group

```
kids(group2,[(jill,dob(14,12,87),[8,3,5,7]), ... ])
```

g. Use the quicksort procedure to produce a list of girls in order of date of birth.

h. Define a procedure that produces a single list of both groups in order of date of birth.

i. Define a procedure that produces a list of names of children who have completed a specific workbook.

4.6 PROCEDURAL MATTERS

Prolog programs are statements in a restricted form of first-order logic; viewed in this way, they are said to have declarative semantics. A program defines the relationship of data objects to each other without committing objects to be either input or output. Ideally a logic-programming system would be presented with a relation containing both known and unknown objects and would report suitable values for the unknown objects. Prolog does indeed do this for many simple relations, but for reasons connected with practical implementation, it is not generally the case. Worse still, we find that a particular order of clauses in a file or an order of subgoals in a rule body might prevent a result being obtained at all. Prolog operates in a top-to-bottom, left-to-right direction through text files, and the outcome of a particular query is heavily dependent on this procedural behaviour. As a result, the language has a procedural semantics that depends on the layout of facts and rules in a text file. It is sometimes helpful to think of declarative semantics as the true specification and procedural semantics as the result actually obtained by a particular ordering.

To explore the difference between declarative and procedural semantics, we consider again the problem of finding routes through the directed acyclic graph of Figure 4.2. Recall that procedure `route` decides if there is a route between two points:

```
route(Start,Finish) :- path(Start,Finish).
route(Start,Finish) :- path(Start,X),route(X,Finish).
```

Now we define procedure `route2`; it is similar to `route` except that the route and path conditions in the body of the rule are interchanged:

```
route2(Start,Finish) :- path(Start,Finish).
route2(Start,Finish) :- route2(Start,X),path(X,Finish).
```

The original version first found a path from a starting-point to position `X`, then called the route procedure to find a route from that point to the finish. Now a route is found to `X` and the database is searched to find a final completing step. The existence of a route between points a and d in the graph is confirmed by the query

```
?-route2(a,d).
yes
```

but when presented with a query about a route that does not exist, such as

```
?-route2(e,a).
```

the procedure fails to produce the expected negative result or indeed any result at all. The first clause initiates a search for `path(e,a)` that results in failure, then the second clause makes an immediate recursive call to `route2(e,X)`. Since there is no fact in the database with arguments that could unify with `path(e,X)`, this clause fails and a further successful unification occurs with the head of the second clause. This cycle never terminates, so the system eventually runs out of memory.

If the order of the rules themselves is changed in addition to this reordering of the conditions, the following procedure results:

```
route3(Start,Finish) :- route3(Start,X),path(X,Finish).
route3(Start,Finish) :- path(Start,Finish).
```

This version places the recursive relation first in the body of the first clause, so the first thing the procedure does is to make a recursive call to itself. If we now pose the same query `?-route3(a,d).` that worked previously for `route2`, no result is obtained. An initial goal `route3(a,d)` results in a subgoal of `route3(a,X)`, initiating a series of non-terminating recursive calls.

These revised versions of `route` have the same declarative semantics as the original, but each one has a procedural semantics that prevents termination in some cases. It would of course be possible for two programs to have the same declarative semantics and different procedural semantics, but to produce the same result. It is the procedure for generating the results that decides the procedural semantics rather than the result obtained. Changing the order in which clauses or conditions appear

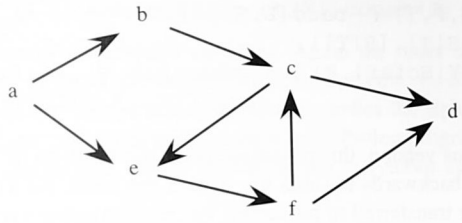

Figure 4.4 A directed cyclic graph

does not change the declarative semantics and does not therefore change any result that is obtained. What it can do is to prevent any result being obtained at all, i.e. it might prevent program termination.

Even when we accept that `route` has to be written in a particular way to get a result, the earlier working version has a number of deficiencies. This procedure applies only to directed acyclic graphs, and we might wish to find routes through directed graphs that include cycles or even through undirected graphs. Rather than extend the original procedure, we first look at a different way of holding the graph information as a list of path functors:

```
graph(one,[p(a,b),p(b,e), ... , ]).
```

This style of presentation has the advantage that it names a particular graph, so a database could contain several graphs, allowing references to be made to any required graph. A path exists between two points `X` and `Y` in a list of paths `L` if it is a member of the list

```
path(X,Y,L) :- member(p(X,Y),L)
```

and a further procedure may be defined to use the path relation

```
route4(S,F,L) :- path(S,F,L).
route4(S,F,L) :- path(S,Z,L),route4(Z,F,L).
```

then a search is related to a particular graph through the rule

```
findrt(X,Y,G) :- graph(G,L),route4(X,Y,L).
```

Procedure `route4` does much the same job as `route` and is equally restricted to directed acyclic graphs. The existence of a cycle in a graph such as Figure 4.4 causes the routine to follow a path in a circle without ever reaching the finishing-point.

One way of avoiding termination problems is to accumulate a list of nodes as they are encountered and to check this list as the route is extended. A check of this kind is easily carried out with the `free` (non-member) procedure described earlier. One advantage of accumulating a trail of used nodes for checking in this way is that the list itself may be produced, informing the user not only of the existence of a route, but also its intermediate stages. If `L` is a graph expressed in the list form above, `S` and `F` are starting and finishing nodes and `T` is a trace of the route between `S` and `F`, these objects are related as follows:

```
route5(L,S,F,T) :- racc(L,S,[F],T).
racc(L,S,[S|T],[S|T]).
racc(L,S,[Y|Sofar],R) :- member(p(X,Y),L),free(X,Sofar),
                         racc(L,S,[X,Y|Sofar],R).
```

Unlike the previous version, this procedure takes the finish as its starting-point and attempts to work backwards towards the start of the route. As a first step, the parameters above are transferred to procedure `racc`, converting variable `F` to a list at the same time. If the head of this third argument is identical to the starting-point, a route has been found and is transferred to the fourth argument. If not, a path `p(X,Y)` capable of extending the route backwards from the finish is chosen from the list of paths `L`, and node `X` is checked to see if it has already been used. If it is indeed free, the node is added to the accumulating list in the third argument position of the recursive rule. A route may then be found in any graph with the rule

```
findroute(G,S,F,T) :- graph(G,L),route5(L,S,F,T).
```

A surprisingly small modification of this program allows routes to be found between nodes in an undirected graph, since the main problem of avoiding the repeated use of a single node has already been solved.

4.6.1 Backtracking and the cut

The order of evaluation in Prolog is evident on running the following program:

```
const(true).
const(false).
pair(X,Y) :- const(X),const(Y).
```

with the query `?-pair(A,B)`. Variable `X` (shared with `A`) is instantiated first because it is the leftmost predicate in the body of the rule and value `true` is taken because this is the first matching fact, working from top to bottom. Variable `Y` is then also instantiated to `true` because attempts to match this subgoal work separately from top to bottom. Since both subgoals have succeeded, Prolog reports `A = true, B = true`. Backtracking now takes place from right to left, so the most recent instantiations are undone and new ones attempted. Another search of the database starting from a position just below the previous success reveals that `Y` can be instantiated to `false`, a second satisfaction of the body is obtained and Prolog reports `X = true, Y = false`. A further search fails to find another value for `Y` and the predicate fails, causing backtracking to the previous subgoal. `X` is instantiated to `false` and the database is searched from the top for a match with `const(Y)`, resulting in the same two instantiations as before.

The pair rule might be extended to three subgoals:

```
triple(X,Y,Z) :- const(X),const(Y),const(Z).
```

producing eight sets of instantiated variables. Again the order of the facts in the database decides the order of the triplets, so `(true,true,true)` is followed by `(true,true,false)` as backtracking first resatisfies the rightmost subgoal.

A special predicate called cut is provided to give Prolog programmers control of the backtracking mechanism and is shown by the `!` symbol as in the following example:

```
pairb(X,Y) :- const(X),!,const(Y).
```

Cut acts as a valve that allows satisfactions to proceed from left to right, but prevents any attempt to backtrack from right to left over the cut symbol. Instantiated variables on the left of a cut symbol are committed to those values when the cut symbol itself is satisfied, but those on the right can be resatisfied in the usual way. A pair rule as modified above produces only two satisfactions, both of which have `X` instantiated to `true`. Moving the cut symbol to the right as in

```
pairc(X,Y) :- const(X),const(Y),!.
```

results in just one satisfaction with both `X` and `Y` instantiated to `true`.

A suitably placed cut can make a program run much more efficiently by preventing unnecessary work. Earlier we saw that the valuation of an implication is `true` if the antecedent is `false` or the consequent `true`, but only one of these conditions needs to occur. If both conditions occur, the rule succeeds on two counts and the time spent evaluating the consequent is wasted. A suitably placed cut prevents a further attempt at satisfaction after the first attempt succeeds:

```
val(X -> Y,true) :- val(X,false),!;val(Y,true).
```

If the first subgoal is satisfied, the cut is also satisfied and no further attempt is made to satisfy the `val` relation.

Multiple reports of success might be a problem in other situations. For example, the familiar member definition finds satisfactions every time the required item occurs in the list. A cut following the first satisfaction prevents any attempt to find alternative satisfactions:

```
member(X,[X|_]) :- !.
member(X,[_|L]) :- member(X,L).
```

Programs can also be made more efficient in situations where a number of mutually exclusive possibilities occur. For example, the `max` procedure given earlier requires the use of two separate comparisons, though only one can be true:

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

A cut placed after the first comparison will be satisfied if the comparison itself is satisfied, and no attempt will be made to instantiate the second clause. But if the first comparison fails, it may be assumed that the second will succeed:

```
max(X,Y,X) :- X >= Y,!.
max(X,Y,Y).
```

If the first comparison succeeds, the cut also succeeds and the following clause is not attempted; but if the comparison fails, the second clause is the only other option.

4.6.2 Input–output predicates

Interactions with the Prolog system have so far consisted of queries at the prompt that are answered by a simple "yes" or "no" together with a possible instantiation of variables. Thus, a database of the form

```
mother(jill,bob).
mother(lucy,john).
mother(mary,carol).
```

could tell us the mother of a particular child through the following query:

```
?-mother(X,bob).
X = jill
```

An alternative interactive method of obtaining the same information is possible through `read` and `write` predicates:

```
go :- write('input child's name>>'),
      read(Child),nl,
      mother(Mother,Child),
      write('The mother of'),write(Child),
      write('is'),write(Mother).
```

This rule consists of a head without arguments and a body containing a number of relations that are, as always, satisfied from left to right. First of all, a `write` predicate is satisfied when the text contained as an argument is written to the standard output device, usually the screen. A `read` predicate is then satisfied when a term is provided by the standard input device, usually the keyboard, and this term is taken as the value of the `Child` variable. A further built-in predicate `nl` then causes a new line in the output device, so the text telling the user the mother of a child appears on a fresh line. Between reading the name of the mother and writing the output, relation `mother` is accessed to provide the required information. Notice that the `write` predicate substitutes values for variables if they are instantiated at the time, whereas text contained between quotes is printed literally. Instead of exchanging information between the head and the body of a rule, a direct exchange between the relations and input/output devices occurs. This rule is achieving its result through the side-effects of `read` and `write` relations in the same way as an imperative language.

EXERCISES 4.6

1. Cost informaton may be included in a list of paths in much the same way as in the facts of Exercise 7 in Exercises 4.4:

```
graph(five,[p(a,b,20),p(a,c,35),p(b,d,43),p(d,e,26),
p(c,e,44)]).
```

   Define a procedure that calculates the cost of each possible route from one point to another in a directed acyclic graph. Test the procedure by calculating the costs of the two routes between nodes a and e.

2. Translate the graph shown in Figure 4.4 into a labelled list of edges similar to that shown in the text for the earlier acyclic graph. Use the procedure `route5` to find every possible route from node a to node d.

3. Cost information might also be included in directed graphs containing cycles, e.g. Figure 4.4. Extend the list of edges in the previous exercise to include cost information, and define a Prolog procedure that finds the cost of every route between two possible points in the graph.

4. An undirected graph allows moves between nodes in either direction, but only one direction is included in the list of edges. As a result, the presence of an edge `p(a,b)` implies the existence of a further edge `p(b,a)`. A path between two nodes is possible as follows:

```
path(X,Y,G) :- member(p(X,Y), G);member(p(Y,X),G)
```

   Modify the route-finding procedure shown in the text so that it finds all routes through undirected graphs.

5. The `delete` procedure of Section 4.5 requires relation `X \== H` in the third clause to prevent both the second and third clauses being satisfied when the head element is the element to be deleted. Redesign the procedure using a cut rather than this relation.

6. Remove relation `free` from the union and intersection procedures as defined in the text and test the new procedure that results. Use a cut to remove the problem that now arises when the procedure is used.

7. Write a small procedure without cuts that relates times of the day before 1200, 1700 and 2400 hours to the symbols `good_morning`, `good_afternoon` and `good_evening`. An interaction at the prompt should then appear as follows:

```
?greet(14.00,X).
X = good_afternoon
```

   Show that a more efficient procedure is possible when cuts are added.

8. Write a procedure using `read` and `write` predicates that prompts a user for a starting-point and then for a finishing-point in a directed acyclic graph such as Figure 4.2. Advise the user whether a route exists between the two points.

4.7 PROGRAMS FOR PROPOSITIONS

In this section the Prolog language is used to manipulate the syntactic form of a proposition and to decide the truth of propositions in some or all valuations. As a first step in this direction, a small routine to check that a proposition is correctly formed is provided, then a further program converts an arbitrary proposition to negation normal form. An implementation of the Wang algorithm allows propositions to be tested as tautologies then a truth table program allows the evaluation of contingent propositions.

A proposition may be a constant symbol `f`, one of a number of statement symbols `p`, `q`, `r`, `s` or the application of a logical connective to other propositions. Section 4.3 explained how the connectives `~`, `&`, `v`, and `->` are defined in the Prolog operator notation, providing the usual precedence rules for propositional operators. Given that those definitions have been made, the following program checks a string to see if it is a proposition:

```
check(F) :- member(F,[p,q,r,s,f])
check(~F) :- check(F).
check(F) :- (F = X & Y;F = X v Y;F = X -> Y),check(X),
check(Y).
```

Correctly formed formulas could then be checked at the prompt:

```
?-check(~p -> ~q & r).
yes
```

An extension of the above program informs the user if a proposition is one of the three Hilbert axioms. Only a proposition having the form of the first axiom could match with the functor pattern in the head of the following rule:

```
axiom1(X -> Y -> X) :- check(X),check(Y).
```

Provided `X` and `Y` are correctly formed, a proposition that matches this pattern is an axiom. Similar rules may be written for the other two axioms.

4.7.1 Finding negation normal forms

Negation normal forms of propositions contain a restricted number of logical connectives, specifically the set {`v`, `&`, `~`} in the notation used above. More important, every occurrence of the negation symbol must stand directly before an atom, so the

scope of the operator is limited to that one atom. A conversion of NNF generally proceeds in two stages: unwanted connectives are first replaced by equivalent forms, then negation symbols are driven into the atoms by repeated applications of De Morgan's rule. Any unwanted connective may be removed in this way, but most commonly it is the implication and mutual implication connectives that have to be discarded; this is achieved by the following program:

```
impout((A <-> B),((A1 & B1) v (~A1 & ~B1))) :-
                                    impout(A,A1), impout(B,B1).
impout((A -> B),(~A1 v B1)) :- impout(A,A1),impout(B,B1).
impout((A & B),(A1 & B1)) :- impout(A,A1),impout(B,B1).
impout((A v B),(A1 v B1)) :- impout(A,A1),impout(B,B1).
impout((~A),(~A1)) :- impout(A,A1).
impout(A,A) :- member(A,[p,q,r,s,f]).
```

Unwanted logical connectives are replaced by acceptable ones, and functor arguments `A` and `B` are replaced by arguments `A1` and `B1`, free of such connectives. Once consulted, the above procedure allows the following interaction:

```
?-impout((p -> q) -> r).
~(~p v q) v r
```

returning a result free of implication symbols, but not yet in negation normal form. Negation symbols with scope greater than a single atom are then moved inwards by these mutually recursive procedures:

```
nnf((A & B),(A1 & B1)) :- nnf(A,A1),nnf(B,B1).
nnf((A v B),(A1 v B1)) :- nnf(A,A1),nnf(B,B1).
nnf(~A,A1) :- dual(A,A1).
nnf(A,A) :- member(A,[p,q,r,s,f]).

dual((A & B),(A1 v B1)) :- dual(A,A1),dual(B,B1).
dual((A v B),(A1 & B1)) :- dual(A,A1),dual(B,B1).
dual(~A,A1) :- nnf(A,A1).
dual(A,~A) :- member(A,[p,q,r,s,f]).
```

which may be combined in a single rule that converts a formula `F` to its equivalent negation normal form `R`.

```
transform(F,R) :- impout(F,X),nnf(X,R)
```

4.7.2 The Wang algorithm

Wang's algorithm is essentially an implementation of the propositional part of the Gentzen G proof system described earlier; it therefore depends on the reduction of a sequent of form

antecedent $\Rightarrow$ succedent

to axioms. Although both antecedent and consequent are sets of propositions, a Prolog program implements these sets as lists to be accessed from left to right. An axiom is a sequent with a common atom contained in both antecedent and succedent, a feature very easily recognised by the common procedure given earlier. G system rules are fairly easily translated into Prolog rules and are applied to the sequent until axioms are obtained or until no further applications are possible.

The first step in deciding if a formula is a tautology, i.e. is universally valid, is to make that list the succedent of a sequent and to take an empty list as the initial antecedent. These two lists are then made the arguments of a `seq` body:

```
valid(Formula) :- seq([ ],[Formula]).
```

A sequent is true if its lists contain a common element or if applications of the rules produces such sequents:

```
seq(Left,Right) :-  common(Left,Right).

seq(Left,Right) :-  member(~A,Right),
                    delete(~A,Right,Newright),
                    seq([A|Left],Newright).

seq(Left,Right) :-  member(~A,Left),
                    delete(~A,Left,Newleft),
                    seq(Newleft,[A|Right]).

seq(Left,Right) :-  member(A -> B,Right),
                    delete(A -> B,Right,Newright),
                    seq([A|Left],[B|Newright]).

seq(Left,Right) :-  member(A -> B,Left),
                    delete(A -> B,Left,Newleft),
                    seq(Newleft,[A|Right]),
                    seq([B|Newleft],Right).

seq(Left,Right) :-  member(A & B,Right),
                    delete(A & B,Right,Newright),
                    seq(Left,[A|Newright]),
                    seq(Left,[B|Newright]).

seq(Left,Right) :-  member(A & B,Left),
                    delete(A & B,Left,Newleft),
                    seq([A,B|Newleft],Right).
```

These rules just implement the G system rules of Figure 1.10 and are easily recognised from the arguments of the member subgoal. A proposition is easily tested for validity as follows:

```
?-valid((~A -> ~B) -> (B -> A)).
yes
```

and application of rules is easily followed. First an application of the right -> rule produces the sequent

```
seq([~A -> ~B],[B -> A])
```

then a further application of the same rule produces

```
seq([B,~A -> ~B],[A])
```

Obviously there was a choice of rule at this point; a left implication was used because it occurs before the right implication in the program text, not because it avoids splitting the sequent. Now the left -> rule generates two subsequents:

```
seq([B],[~A,A])
seq([B,~B],[A])
```

and applications of the right and left negation rules then produce axioms.

4.7.3 Printing truth tables

Combinations of the Boolean values `true` and `false` are generated by the pair and triple relations given earlier:

```
const(true).
const(false).
pair(X,Y) :- const(X),const(Y).
triple(X,Y,Z) :- const(X),const(Y),const(Z).
```

and these relations may be used to generate the inputs to the `val` relation described in Section 4.3. Used in this way, the `val` relation requires two terminating clauses in addition to pairs of clauses required for the evaluation of expressions with each connective. Showing only the implication relation, this gives us

```
val(true,true).
val(false,false).
val(X -> Y,true) :- val(X,false),!;val(Y,true).
val(X -> Y,false) :- val(X,true),val(Y,false).
```

A cut has been added to the implication relation to prevent `val(false -> true,` true) being satisfied twice, and similar cuts would also have to be inserted in the disjunction and conjunction relations. Once Prolog has consulted these modified definitions, a query at the prompt could tell the user the result of evaluating an expression for every input combination as follows:

```
?-triple(X,Y,Z),val(X -> Y -> Z,R).
X = true, Y = true, Y = true, R = true; and so forth
```

This works well enough, but the output style obscures any patterns that might emerge from the computation and it would be much more helpful to see the familiar Boolean table. Output of this kind is possible with `table`:

```
table :- triple(X,Y,Z),val(X -> Y -> Z,R),line(X,Y,Z,R),fail.
line(A,B,C,D) :- write(A),tab(3),write(B),tab(3),
                 write(C),tab(3),write(D),nl.
```

The first two subgoals of `table` generate values for `X`, `Y`, `Z` and `R` in just the same way as if they were used at the prompt, but in `table` they are passed to procedure `line` for printing. Input and output predicates are only satisfied once on each left-to-right pass, so a built-in predicate called `fail` is added to force backtracking. Predicate `fail` is simply a built-in predicate that always fails, causing Prolog to backtrack and thus resatisfy the triple predicate. A `tab` predicate outputs a number of spaces indicated in its argument and a newline predicate `nl` causes output to continue at the beginning of the next line, giving us the expected tabular output.

We might add headings to each of the columns with a customised header predicate as follows:

```
header :- write('X'),tab(3),write('Y'),tab(3),
write ('Z'),tab(3), write('Result'),nl.
```

The table, complete with header, might be printed through the rule

```
show :- header,table.
```

Prolog systems have a built-in predicate that allows order comparisons between terms, and this feature allows alphabetical comparisons between symbolic atoms. Thus, queries at the prompt proceed as follows:

```
?-false @< true.
yes

?-false @< false
no
```

Relational operations normally cause their two numerical arguments to be computed before a comparison is made, but these special predicates compare individual characters from left to right until a difference is obtained. Strings are tested for alphabetical order in the same way that numbers are tested by the more familar relations. Minimum and maximum relations on symbol strings may be defined analogously to those defined on numbers:

```
mins(X,Y,X) :- X @=< Y,!.
mins(X,Y,Y).

maxs(X,Y,X) :- X @>= Y,!.
maxs(X,Y,Y)
```

If these two relations are restricted to the arguments `true` and `false`, they exactly reproduce the behaviour of conjunction and disjunction operations. This allows the valuation rules for these operations given earlier to be written simply as

```
vals(A & B,V) :- vals(A,A1),vals(B,B1),mins(A1,B1,V).
vals(A v B,V) :- vals(A,A1),vals(B,B1),maxs(A1,B1,V).
```

and an implication can be written in terms of the equivalent disjunction

```
vals(A -> B,V) :- vals(~A,A1),vals(B,B1),maxs(A1,B1,V).
```

As before, it would be necessary to add terminating clauses to the relation:

```
vals(true,true).
vals(false,false).
vals(~X,true) :- vals(X,false).
vals(~X,false) :- vals(X,true).
```

Procedure `vals` produces the same results as procedure `val`, but has the advantage that it may be used for the ternary logic described in Chapter 7 without change.

EXERCISES 4.7

1. Write procedures `axiom2` and `axiom3` that test propositions to see if they match Hilbert axioms 2 and 3. The style of these procedures should follow the style of `axiom1` in this section.

2. Write a routine that converts propositions containing the symbols `~`, `v`, `&` and `->` to a proposition containing only negations and implications.

3. Enter and consult the first three sequent rules shown in the text then test that they are working with the following query:

   ```
   ?-seq([],[a,~a])
   yes
   ```

   Remember that the appropriate operator declarations have to be made. Add the two implication rules, reconsult and test with the following formula:

   ```
   (p -> q) -> (~q -> ~p)
   ```

   Write left and right `v` rules in the style of the others shown in the text, then use them to demonstrate the following tautologies:

   ```
   p v q -> q v p
   (p v (q & r)) -> ((p v q) & (p v r))
   ```

4. Show that the `vals` procedure described in this section works equally well for the ternary logic described in Chapter 7. Modify the constant facts to include a null element, then produce truth tables for the ternary expression

$(p \wedge q) \vee (\neg p \wedge r)$

5. Write rules in the Wang style for the intuitionistic form of logic explained in Chapter 9. Neglect the "thin" rule, even though this means the algorithm sometimes fails to produce a result.

6. Write a procedure based on the Wang algorithm that produces either CNF or DNF forms from a deduction tree.

CHAPTER FIVE

Logic with equality

Chapter 1 emphasised a clear distinction between the syntactic forms of propositions and the semantic functions that provide interpretations for the propositional symbols. Syntactic forms are decided by the order in which symbols appear in a string of symbols whereas the semantics of that string of symbols is decided by the interpretation, meaning or denotation given to the symbols. Many different interpretations may be given to strings defining formulas and we have to be sure that the denotations given are consistent with the syntactic form. This distinction between syntactic and semantic forms might at first seem unfamiliar, but it has been with us since our earliest days in primary school. If a child were asked whether 6×7 is equal to 42, he might reasonably reply that it is not. The symbols 6, $\times$ and 7 on the left are quite clearly different from the 4 and the 2 on the right, so the two expressions are not the same. Two expressions are syntactically the same if they consist of the same symbols in the same order. Later the child will learn that, although the expressions are syntactically different, they denote the same value and are interpreted as equivalents.

As a result of our early training, we accept a denotational or algebraic meaning of equality. We place an equality symbol between two syntactically different expressions if they denote equivalent values, and we feel free to replace arithmetic expressions with equivalent values. In our daily lives we use the equality symbol (=) to mean denotational equality. Although the Prolog language contains several versions of an equality symbol, the preceding chapters have repeatedly warned that it is a language based on logic without denotational equality. Equality in Prolog means syntactic equality, i.e. two Prolog expressions are equal when they are syntactically identical. In the following chapters we shall see that a denotational form of equality allows a different style of computation from the SLD mechanism of logic languages. From now on, we shall use the word *equality* to indicate algebraic

equality. First of all we need a clearer idea of what equality involves and how the concept extends the logic without equality described in the first four chapters.

5.1 EQUIVALENCE AND EQUALITY

Boolean expressions have the syntactic forms and semantic functions described in Chapters 1 and 2. From a very early stage in Chapter 1, we used the concept of equivalent Boolean expressions in much the same way that we use the idea of equivalent arithmetic expressions. In fact, we have already employed the denotational meaning of equality in the earlier evaluations of compound Boolean expressions. For example, if statements p and q are interpreted respectively as *true* and *false*, a compound formula $\neg p \vee q$ may be interpreted by the expression *or*(*not true*, *false*). This expression is in turn evaluated by reference to the truth tables contained in Chapter 1:

$$\begin{aligned} or(not\ true,\ true) &= or(false,true) \\ &= true \end{aligned}$$

Clearly a meaning or semantics for each of the classical Boolean connectives is contained in the truth table for that connective. Although this might seem quite a trivial example, it is worth reminding ourselves of the justifications for each step. First of all, the expression *not true* is replaced by the equivalent value *false*, then the resulting expression *or*(*false*, *true*) is replaced by the equivalent value *true*. The fragment *not true* denotes the same value as *false* and we are justified in replacing one by the other. Similarly the fragment *or*(*false*, *true*) denotes the same value as *true* and whenever the former expression occurs it may be replaced by the latter.

Two expressions denote the same value when they produce the same result in all interpretations. For example, the equivalence

$$p \rightarrow q \equiv \neg p \vee q$$

means that the two expressions denote the same value whatever the interpretations of p and q. However, denotational equality signifies more than just equivalence: it includes a justification for replacing any expression by a different but equivalent expression. Fragments within expressions are usually replaced by equivalent simpler forms until the simplest possible form is obtained. In principle there is no reason why fragments within expressions should not be replaced by increasingly large equivalent expressions, it is just less likely that this would be useful. The procedure of substituting equivalent terms is called term rewriting and leads us to a new method of computation based on term-rewriting systems (TRSs)

Boolean expressions must always reduce to one of the primitive values *true* or *false*, so every Boolean expression is equivalent to one of these constants. If we consider just the interpretations *true*, *false* and *imp* then every correctly formed expression constructed from these constants must be equivalent to either *true* or *false*. This simple observation allows us to partition all such expressions into two equivalence sets:

{*true*, *imp*(*true*,*true*), *imp*(*false*,*true*), *imp*(*false*,*false*), . . . }

{*false*, *imp*(*true*,*false*), *imp*(*imp*(*true*,*true*),*false*), . . . }

Here the first set contains all expressions equivalent to *true*, the second those equivalent to *false*. Since every element in a given equivalence set denotes the same value, any one expression may be used to replace another expression from the same set. In practice a computation consists of a gradual reduction in the size of expressions until one of the two simplest possible expressions is obtained. The Boolean expression above was evaluated by replacing subexpressions with simpler equivalents until no further simplification was possible. The two equivalence sets are characterised by these so-called canonical values, so the equivalence sets might equally well be written as

[*true*] and [*false*]

Sometimes the elements in an equivalence class are said to be congruent or to belong to a congruence class. As a result, the shorthand notations [*true*] and [*false*] are also said to represent congruence classes.

5.1.1 Numerical equivalence

The process of evaluating arithmetic expressions by rewriting terms is already familiar and needs no further explanation. This familiarity with the method perhaps prevents us from seeing it as a term-rewriting system that might be extended to more general computations outside the field of arithmetic. Looking again at the process, an expression 3 * 2 + 4 * 5 is evaluated by systematically rewriting terms until a canonical value is obtained:

```
3 * 2 + 4 * 5
  6   + 4 * 5
  6   +   20
        26
```

Terms 3 * 2 and 4 * 5 are first replaced by the values they denote, then the resulting term 6 + 20 is substituted by an equivalent value. Each subterm is replaced by a simpler equivalent value until the simplest possible value, the canonical or normal value, is obtained. Each replacement is a term rewritten within a term-rewriting system defined by arithmetic tables.

Numerical expressions containing natural numbers and arithmetic operations belong to numerical equivalence classes in the same way that the earlier expressions belong to Boolean equivalence classes. There is, however, an infinite number of numerical equivalence classes corresponding to the infinite series of natural numbers 0, 1, 2, 3, . . . and in the absence of any arithmetic operations these classes contain just the canonical values themselves

$\{0\}, \{1\}, \{2\}, \{3\}, \ldots$

If, in addition to the fundamental values of this type, we define an addition operation that makes certain expressions equivalent, each class contains more than just the canonical value. For example, the standard definition of addition makes $7 + 2$ equivalent to 9 and this expression appears in the equivalence class containing the canonical value, i.e.

$\{9, 6 + 3, 7 + 2, 8 + 1, 3 + (3 + 3), \ldots\}$

Every numerical expression in this class denotes the same value and any one could be replaced by another without changing the value of an expression in which it occurs. One of the major advances in computer science in recent years is the recognition that types such as the natural numbers are more than just the fundamental or canonical values. Operations defined on the fundamental values are equally important and have to be included in a definition of the type.

Many other equivalence classes may be defined, such as the infinite number of equivalence classes of natural numbers with multiplication:

$\{42, 6 \times 7, 3 \times 14, \ldots\}$

If both addition and multiplication are defined in a type, then the number of equivalent expressions is increased to include terms such as $3 + 4 * 3$.

Fractions form similar equivalence classes and a series of expressions is clear from the equivalents of numbers with division:

$\{1/2, 2/4, 3/6, \ldots\}$

Any one value in this equivalence class may be used to replace another in a larger expression, but usually we are interested in reducing an expression to its simplest possible value.

5.1.2 Automated reasoning

Logic languages do use an equality symbol, but its meaning in that context is quite different from denotational equivalence described above. For example, the following interaction might occur at a logic language prompt:

```
?- 6 + 3 = 9
no
```

and an unaware user might be surprised at this response. Logic languages use the equality symbol to represent syntactic equality, so the system is being asked if the syntactic string `6 + 3` is equivalent to the symbol `9`. Clearly it is not, and the language responds accordingly. If an apparently similar question is put to a language using a denotational form of equality, the interaction is

```
?6 + 3 = 9
true
```

Now the question being asked is, does the string `6 + 3` denote the same value as `9`? In other words, do these two expressions belong to the same equivalence class? The obvious answer is quickly returned. If an arithmetic expression alone is presented to a term-rewriting language, it is evaluated and its denotation is returned:

```
?6 + 3
9
```

A logic language based on logic without equality should not be capable of doing arithmetic, but in practice logic languages such as Prolog always have built-in arithmetic features as "impure" additions to the language.

In order to compare the different approaches, a simple computation will be carried out in two different ways, using the SLD logic-programming method and using a term-rewriting approach. Programs are given for the evaluation of the Boolean expression $(p \rightarrow q) \rightarrow r$ for any valuation of the atoms p, q and r. It is convenient in this example to express an individual implication in the prefix form as *imp*(*x*,*y*), so the expression above is written as *imp*(*imp*(*p*,*q*),*r*). A double implication is then evaluated by the following logic program:

C0 *Dimp*(*p*,*q*,*r*,*x*) ← *Eval*(*imp*(*imp*(*p*,*q*),*r*),*x*)
C1 *Eval*(*imp*(*a*,*c*),*false*) ← *Eval*(*a*,*true*),*Eval*(*c*,*false*)
C2 *Eval*(*imp*(*a*,*c*),*true*) ← *Eval*(*a*,*false*)
C3 *Eval*(*imp*(*a*,*c*),*true*) ← *Eval*(*c*,*true*)
C4 *Eval*(*true*,*true*)
C5 *Eval*(*false*,*false*)

An individual implication evaluates to *false* if its antecedent is *true* and its consequent *false*; otherwise it evaluates to *true*. The last two lines of this program are especially interesting because they terminate the recursive calls and effectively transfer a value from left to right.

Suppose now that we want to find the value of an expression such as *imp*(*imp*(*true*, *false*),*true*) using the logic program above. A goal ?-*Dimp*(*true*,*false*,*true*,*x*) is presented and is backward chained through the series of steps shown in the SLD tree of Figure 5.1. This process proceeds by finding every possible binding for the arguments in the goal, leading to two failed branches and two successful branches that each report an *x* value of *true*.

A logic program embeds the term to be evaluated in a relation and includes in that relation a variable term that becomes unified with the result of the evaluation. A goal containing variables might be instantiated in several different ways to produce a variety of possible answers. As a result, logic languages possess a strange ability to start with an answer and find all possible questions that might lead to that answer. For example, the goal *Dimp*(*p*,*q*,*r*,*true*) could be presented as a goal and the SLD mechanism would find all possible combinations of p, q and r that could fit this pattern. This bidirectional nature follows from the fact that logic languages are built from relations as opposed to functions and are capable of producing multiple results for a single set of input arguments. Indeed there is no clear distinction

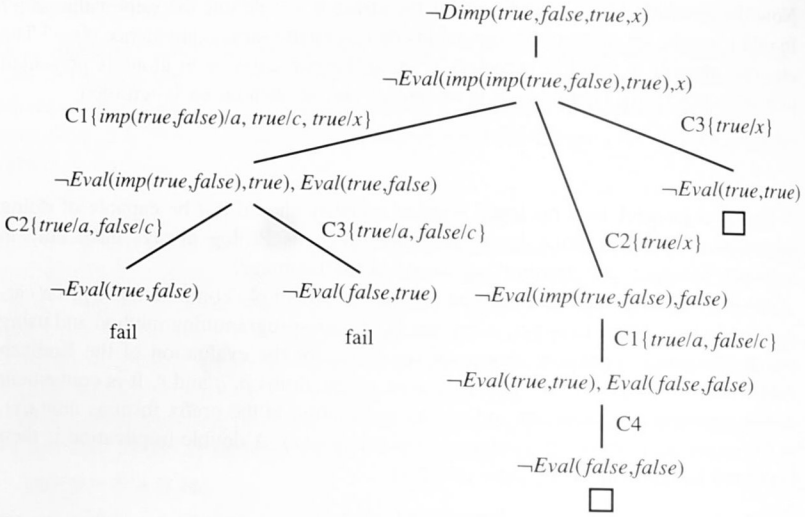

Figure 5.1 An SLD tree

between arguments as input or output, and Prolog expressions are said to be non-deterministic.

Computation by term rewriting requires a program that amounts to statements of equivalences and our earlier example requires the following equivalences:

E0 $dimp(p,q,r) \equiv imp(imp(p,q),r)$
E1 $imp(true,false) \equiv false$
E2 $imp(false,c) \equiv true$
E3 $imp(a,true) \equiv true$

To carry out the computation of $imp(imp(true,false),true)$ appropriate argument values are now placed in $dimp(p,q,r)$ and a series of terms are rewritten according to the program equivalences:

$dimp(true,false,true)$	
$imp(imp(true,false),true)$	E0
$imp(false,true)$	E1
$true$	E2

Each step in this evaluation replaces one term in the expression by another one that the program defines to be equivalent. Notice that each replacement uses the equivalences from left to right, simplifying the expression until the simplest (canonical) value is obtained. In principle these equivalences can be used in either direction, but in practical term-rewriting systems they will usually be used in the direction that ensures simplification.

A term-rewriting system uses only terms, substituting equivalent fragments within an expression according to a program until the simplest possible value is obtained. Most important, a term-rewriting system produces just one answer and there is a very clear distinction between the initial input arguments and the final value denoted by the expression, the result. If there is only one result, the computation is inherently functional and term rewriting quickly converges on that single result. On the other hand, the exhaustive searching mechanism of a logic language goes on searching for further satisfactions because nothing in the language tells it that a particular expression only has one result. A major difference between the two approaches is that a term within a logic program relation is never replaced by an equivalent value as the computation proceeds. The concept of replacing one term by another of equivalent value simply does not exist in languages such as Prolog. A functional language, on the other hand, does little else except compute equivalent values and replace subexpressions by their denotations until a canonical value is obtained.

It would be wrong to assume that term rewriting is in some way better than logic programming because the above computation is more compactly achieved as a series of term rewrites. Some queries naturally lead to multiple answers, particularly in the area of databases, and logic-programming techniques are then advantageous.

5.2 EQUALITY IN DEDUCTION TREES

The discussion above suggests that two forms of equality may be defined: the first arising as a syntactic equivalence, the second as a semantic or denotational equivalence. Logic languages such as Prolog are essentially animated forms of the logic described in the first two chapters and admit the first form of equivalence, but not the second. Since we generally accept equality to mean denotational equality, the logic of the first four chapters (including the logic of Prolog) is described as "logic without equality". From the brief discussion above it is clear that a denotational concept of equality is a very useful property, providing us with the ability to replace terms by the values they denote, hence reducing an expression to its simplest form. As a result, we now define an extension of the earlier logic that incorporates equivalence, giving us a "logic with equality".

In fact, the extension of the earlier logic to include equality is straightforward in principle, but makes constructions such as semantic tableaux very large in practice. All that is required is the addition of three equality axioms that may be added to a tableau or deduction tree at any point:

$$x = x$$

$$x_1 = y_1 \wedge x_2 = y_2 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1,x_2, \ldots ,x_n) = f(y_1,y_2, \ldots ,y_n)$$

$$x_1 = y_1 \wedge x_2 = y_2 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,x_2, \ldots ,x_n) \rightarrow P(y_1,y_2, \ldots ,y_n)$$

Thus, if a term c occurs anywhere in a tableau or G system proof we are permitted to add the formula $c = c$, asserting that a term denotes the same value as itself. This

first equality formula is sometimes called the reflexive axiom. Each of the axioms above should be read with an implicit universal quantification for each variable, so the reflexive axiom should be interpreted as the formula

$$\forall x(x = x)$$

The second and third axioms contain conjunctions of equal terms, x_i and y_i, that justify the equality of the same function with different arguments or the same predicate with different arguments. Specifically, the second equality axiom tells us that a function f applied to equivalent arguments denotes an equivalent value. Suppose we have an arity-two function *max* that denotes the maximum of two numbers; this axiom might take the form

$$2 * 8 = 16 \wedge 3 * 5 = 15 \rightarrow max(2 * 8, 3 * 5) = max(16, 15)$$

If a function is applied to syntactically different but equivalent arguments, it returns the same value. Since 16 denotes the same value as 2 * 8 and 15 denotes the same value as 3 * 5, function *max*(2 * 8, 3 * 5) denotes the same value as function *max*(16, 15).

Axiom three states that if a certain predicate P is *true* for one set of arguments x_i, it must be *true* for an equivalent set y_i. The quantifiers for this axiom can be made explicit in specific examples such as the application to an arity-two predicate P:

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2 (x_1 = y_1 \wedge x_2 = y_2 \wedge P(x_1,x_2) \rightarrow P(y_1,y_2))$$

An interesting special case of this last formula arises when the predicate P is equality itself:

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2 (x_1 = y_1 \wedge x_2 = y_2 \wedge x_1 = x_2 \rightarrow y_1 = y_2)$$

Making x_1, x_2 and y_2 the same as x and renaming y_1 simply as y, we obtain the formula

$$\forall x \forall y (x = y \wedge x = x \wedge x = x \rightarrow y = x)$$

or removing the duplicated equality, we obtain the axiom

$$\forall x \forall y (x = y \wedge x = x \rightarrow y = y)$$

This axiom, together with the reflexive axiom, is shown by the Gentzen-style proof in Figure 5.2 to entail the statement

$$\forall x \forall y (x = y \rightarrow y = x)$$

usually called the symmetric condition of equality. This proof shows that the symmetric property of equality follows from two of the equality axioms given earlier. Remember that equality is a special case of a predicate symbol. If each equality symbol in Figure 5.2 is replaced by predicate symbol P, so that $x = y$ becomes $P(x,y)$, the proof looks much more like those given in Chapter 2.

$$\forall x \forall y(x = y \wedge x = x \rightarrow y = x), \forall x(x = x) \Rightarrow \forall x \forall y(x = y \rightarrow y = x)$$

$$\forall x \forall y(x = y \wedge x = x \rightarrow y = x), \forall x(x = x) \Rightarrow a = b \rightarrow b = a \qquad \text{2 right } \forall$$

$$a = b \wedge a = a \rightarrow b = a, \forall x(x = x) \Rightarrow a = b \rightarrow b = a \qquad \text{2 left } \forall$$

$$a = b \wedge a = a \rightarrow b = a, a = a \Rightarrow a = b \rightarrow b = a \qquad \text{left } \forall$$

$$a = b, a = b \wedge a = a \rightarrow b = a, a = a \Rightarrow b = a \qquad \text{right } \rightarrow$$

$$a = b, b = a, a = a \Rightarrow b = a \qquad \text{left } \rightarrow$$

$$\times$$

$$a = b, a = a \Rightarrow (a = b \wedge a = a), b = a$$

$$a = b, a = a \Rightarrow a = b, b = a \qquad a = b, a = a \Rightarrow a = a, b = a \qquad \text{right } \wedge$$

$$\times \qquad\qquad \times$$

Figure 5.2 Proof of the symmetric property

A third well-known characteristic of equivalence is called the transitive property. To demonstrate this, we again take the special case of the third equality axiom where predicate P is equality itself:

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2 (x_1 = y_1 \wedge x_2 = y_2 \wedge x_1 = x_2 \rightarrow y_1 = y_2)$$

We now set variables x_1 and x_2 to x, relabel y_1 as y and y_2 as z, obtaining a further axiom

$$\forall x \forall y \forall z (x = y \wedge x = z \wedge x = x \rightarrow y = z)$$

Now we want to show that, together with the reflexive axiom $\forall x(x = x)$, this formula entails the transitive axiom

$$\forall x \forall y \forall z (x = y \wedge y = z \rightarrow x = z)$$

If these three formulas are labelled as Axiom1, Axiom3 and Trans, the entailment that we wish to prove is

$$\text{Axiom1, Axiom3} \models \text{Trans}$$

meaning that Trans is true whenever Axiom1 and Axiom3 are true. A semantic tableau proves this entailment by demonstrating that the formula Axiom1 $\wedge$ Axiom3 $\wedge \neg$Trans is a contradiction; the detailed proof is shown in Figure 5.3. First of all, a right universal rule is used to generate three constants a, b and c in a self-generated (Herbrand) universe. Since the resulting formula has no quantifiers, it is then broken down into subformulas using the familiar proposition rules. After this has been done, the constants already generated are used to instantiate the left universal formulas of lines 1 and 2, resulting in further unquantified formulas that can also be decomposed by propositional rules. A trace of the branches of the tableau then shows that every branch has a pair of conjugated formula, so the tableau is closed and the original entailment is confirmed. As a consequence, the transitive condition of equality is proven.

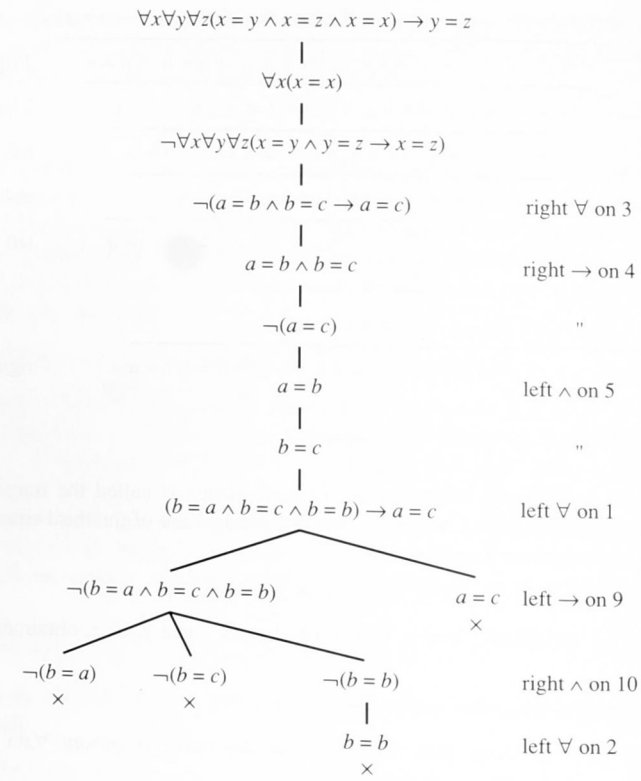

Figure 5.3 The transitive nature of equality

5.2.1 Using equality axioms in a proof

The proofs so far have only derived modified axioms from the three original equality axioms. Now we want to use the original axioms to prove formulas that include equality predicates.

Imagine that we have a relation P that associates football players with the teams in which they play. Thus $P(jones,redskins)$ tells us that *jones* is a player for the *redskins* team. At the same time, we have a function f that maps teams to their captains; if *smith* is captain of the *buffaloes*, then $f(buffaloes) = smith$. Since the captain of a team must always be a player in that team, the relation $\forall xP(f(x),x)$ must always be *true*. If now we find that the captain of team a is player b, i.e. $f(a) = b$, it follows that b plays for a. Expressing this in the form of a proof, we might write

$$\forall xP(f(x),x) \wedge f(a) = b \rightarrow P(b,a)$$

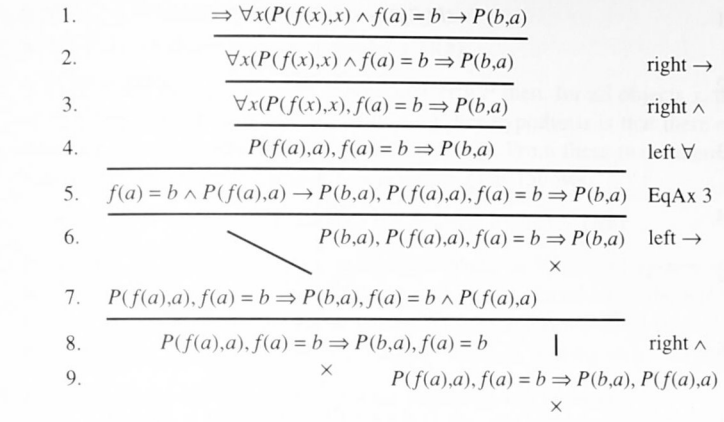

Figure 5.4 A Gentzen proof using an equality axiom

Figure 5.4 shows a Gentzen-style proof for this formula that starts with three familiar inference rules to produce the following sequent in line 4:

$$P(f(a),a), f(a) = b \Rightarrow P(b,a)$$

Intuitively this might seem quite obvious, but the proof is only terminated when identical atoms occur on both sides of the sequent. To achieve this, we have to introduce the following version of equality axiom 3 into the antecedent of the proof:

$$f(a) = \mathrm{b} \wedge P(f(a),a) \rightarrow P(b,a)$$

Two further propositional steps are then required to produce axioms and thus prove the original formula.

As a final example, we use a semantic tableau to validate the following formula involving equality:

$$P(f(y)) \rightarrow \forall x(x = f(y) \rightarrow P(x))$$

In words, if $f(y)$ satisfies predicate P and all instantiations of x denote the same value as $f(y)$, then all x values satisfy predicate P. As usual, the tableau proof follows from a demonstration that the negated formula is a contradiction, leading to a tableau in which every branch contains a clashing pair of atoms. Although the formula is fairly simple, its semantic tableau proof is large because the equality axioms introduce new formulas that require further decomposition. These axioms are introduced in the three positions indicated in Figure 5.5, each of the equality axioms being used once.

A manual approach to the proofs of Figures 5.4 and 5.5 would be guided by some insight into the form of the axioms necessary to conclude a proof. A mechanical theorem prover on the other hand would have to try every possible axiom in the relentless pursuit of a conclusion. It would be possible to add equality axioms to a

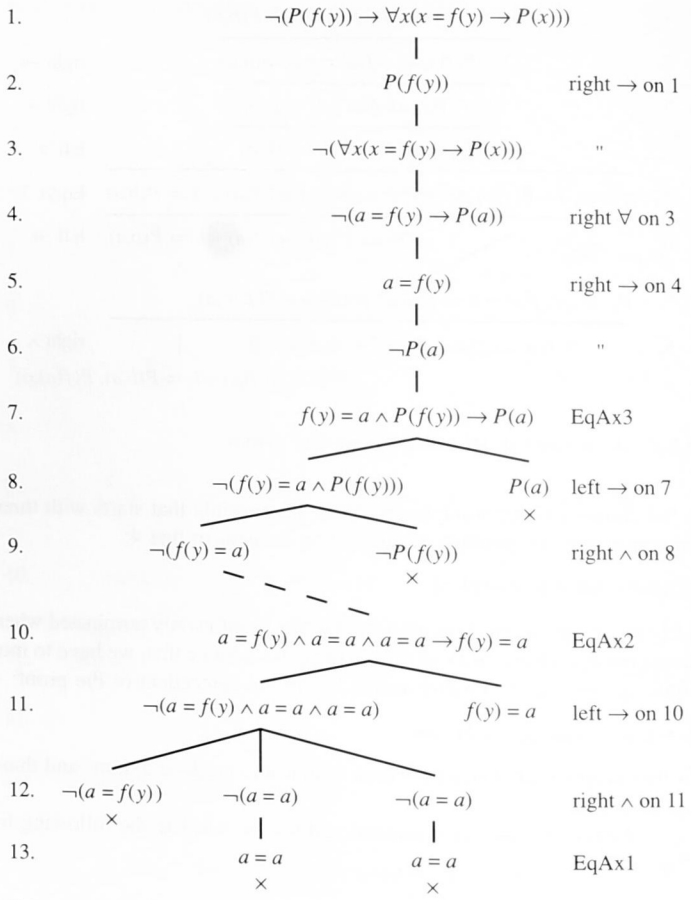

Figure 5.5 A semantic tableau involving equality

logic language like Prolog, but this would result in a massive increase in the search space and an unacceptable loss of efficiency. This is why logic languages avoid equality as though it were the plague.

EXERCISES 5.2

1. Repeat the proof of Figure 5.4 using the semantic tableau notation.
2. Repeat the proof of Figure 5.5 using Gentzen G system notation.
3. Show the following to be valid:

a. $F(a) \rightarrow \exists x(x = a \wedge F(x))$

b. $\forall x(P(x) \rightarrow Q(x)) \wedge P(a) \wedge a = b \rightarrow Q(b)$

4. If a and b are the only objects having property P then, for all objects x, the truth of $P(x)$ implies that x is either a or b. A further hypothesis is that there exists at least one object having both properties P and Q. From these two statements we reason that either object a or b has property Q as follows:

$$\forall x(P(x) \rightarrow x = a \vee x = b), \exists x(P(x) \wedge Q(x)) \vdash Q(a) \vee Q(b)$$

Prove this to be valid with a semantic tableau or with a G system diagram. Remember that a new constant (say c) must be introduced when the left existential rule is applied and this could be reused for the left universal rule. Once this has been done an equivalence axiom may be used, leading to a closed tableau.

5. Use a semantic tableau to demonstate the following theorem

$$P(a), \forall x \forall y(Q(x) \wedge P(y) \rightarrow P(f(x,y))),$$
$$\forall x(Q(x) \rightarrow f(x,a) = x) \vdash \forall x(Q(x) \rightarrow P(x))$$

5.3 ABSTRACT TYPES

We now examine a system of "abstract" types introduced by Birkhoff in 1935 to describe semantics through equivalences. In this context the semantics is also called the algebra and the whole subject area established by Birkhoff is called universal algebra. The importance of this work for computing was first recognised during the early 1970s by several disparate individuals and groups, but the defining work in the area seems to have been produced by Joseph Goguen and other members of the mysteriously named ADJ group. The approach taken by the ADJ group has become standard in computing and is adopted in the following examples. In particular, the ADJ notation has many advantages over Birkhoff's original notation, particularly for the examples of the following chapter.

Abstract types are abstract in the sense that they take away the detail of specific instances, leaving just a bare structure of the objects being described. Taking an informal analogy, we might think of a motor car as an abstract type because the concept itself conveys a great deal of information without indicating a particular vehicle. An object would be an acceptable interpretation of the abstract type motor car if it had four wheels, a metal body, windows and so forth. In short, if it looked like the abstract picture of a motor car that is contained in our minds. An abstract type defines a class of objects and any one of these objects might have features in excess of that required by the abstract type. Thus a motor car with electric windows and four-wheel steering remains an object in the same class of objects because the extras do not change its basic nature.

Abstractions are important in describing software systems because they capture the essence of what is required without the detail of how it is to be achieved.

Abstract types may be used to specify the requirements for a particular piece of software, then the specification can be checked for consistency and completeness. If all is well, the specification is then implemented in some programming language, preferably one which has built-in features that implement objects, i.e. one that is object-oriented. A language such as Miranda, which is both declarative and object-oriented, can be used to implement the abstract type directly. Machine-oriented languages such as C++ require a careful transformation of the abstract type into the machine-oriented code.

An abstract type may be "animated" in order to test that the operations specified do in fact behave as required and this is the basis of the OBJ language described in the next chapter. OBJ may be seen as a fairly inefficient declarative language capable of acting as a test bed for software specifications. In this role it is often described as a prototyping language. Computer scientists working on large-scale projects have developed the idea of abstract type definitions, prototyping and program transformations to provide powerful software production environments.

Chapters 1 and 2 showed how a meaning or semantics was given to defined syntactic forms with the aid of specific interpretations, often supplied in the form of a truth table. In what follows we shall see that this meaning is encapsulated in the syntactic form of an abstract type and that it is the concept of equality that makes this possible. It is the equations in the syntactic form of an abstract type that impose on the type the same behaviour as the truth table. However, we should avoid thinking of the equations as an interpretation or as the semantics because the equivalence statements are purely syntactic in their nature. Abstract types do not have to include equations and a small example without them provides a good introduction to the concept.

The simplest syntactic form or theory that we can describe is the abstract type describing the two constants of propositional logic. An abstract type labelled bool0 is defined as follows:

```
bool0 =
    sorts
        bool
    opns
        ⊥, ⊤ : → bool
```

Here a user-defined name introduces an abstract type bool0 by describing the sorts and the operations (opns) of the type. This particular type has one sort called bool and two zero-arity operations, $\bot$ and $\top$, both of sort bool. Within the specification, an opns line $\bot, \top : \rightarrow$ bool defines these symbols to be of zero arity by not showing a sort on the left of the arrow. This definition does no more than the syntactic definitions at the beginning of Chapter 1, it just introduces two constants. Every operation in a one-sorted abstract type must denote an object of that sort, hence Birkhoff described them as homogeneous algebras. Many-sorted types have two or more sorts defined under the appropriate heading and were described by Birkhoff as heterogeneous algebras. Operations in a many-sorted type may denote any sort of

the type, i.e. the result of applying the operation is one of the sorts defined in the type. In a one-sorted type, such as the example above, the distinction between type and sort is unecessary, but in many-sorted types it is essential. ADJ notation is used here for consistency and in the knowledge that it provides elegant specifications for many-sorted types.

Having defined the complete but very simple theory or specification bool0, we now have to find interpretations for it. These interpretations might also be called representations or, more commonly in computing, objects in the class of the abstract type. A first object, which we label obj1, can be written as follows:

$$\text{obj1} = (\{\mathit{false}, \mathit{true}\}, \bot = \mathit{false}, \top = \mathit{true})$$

This notation first shows a set of domain elements $\{\mathit{false}, \mathit{true}\}$ that carry the interpretation followed by a pairing of abstract type operations with interpretation functions. In this particularly simple case, there are two domain elements that each interpret one of the operations of the abstract type. Equality symbols may be left out of this notation, so the interpretation obj1 might sometimes appear as

$$\text{obj1} = (\{\mathit{false}, \mathit{true}\}, \mathit{false}, \mathit{true})$$

In this notation the order of functions and relations in the interpretation is taken to be the same as for operations and predicates in the abstract type. Three other interpretations may be written as follows:

$$\text{obj2} = (\{0, 1\}, \bot = 0, \top = 1)$$
$$\text{obj3} = (\{\mathit{true}\}, \bot = \mathit{true}, \top = \mathit{true})$$
$$\text{obj4} = (\{0, 1, 2\}, \bot = 0, \top = 1)$$

Object obj2 feels similar to obj1 in that the two different constants of the theory bool0 are mapped to two distinct domain elements, but interpretation obj3 maps the two distinct theory constants to a single domain element. Although obj3 is a perfectly legal interpretation, it is unlikely to reflect the intended semantics of the theory. An interpretation in which distinct constants of the abstract type are mapped to a single domain element is said to involve "confusion". A different problem arises in interpretation obj4, where there are more domain elements than constants in the theory. When both constants are mapped to distinct domain elements, a spare domain element (2) remains. This remaining element is called "junk" because it can have no purpose in implementing the intended meaning of the theory. We shall see that obj1 and obj2 are initial interpretations or, put another way, they provide an initial semantics for the theory bool0; this is connected with the fact that obj1 and obj2 are free of junk and confusion.

Every abstract type contains a signature defining the sorts and operations defined by the type, and this information also defines the Herbrand universe of the type. In addition to the signature there is usually a collection of equations defining equivalences between expressions in the universe. In a sense the equations build into a syntactic form the semantics previously provided by an interpretation, but we should avoid thinking of the equations as the semantics. Instead we should think of the

abstract type as a syntactic form that enforces its behaviour on all interpretations. To illustrate the effect of equations, we expand the specification bool0 to give the larger definition bool1:

```
bool1 =
    sorts
        bool
    opns
        ⊥, ⊤ : → bool
        ¬    : bool → bool
        ∧    : bool, bool → bool
    eqns
        x ∈ bool
        ¬⊤ = ⊥
        ¬¬x = x
        x ∧ ⊤ = x
        x ∧ ⊥ = ⊥
```

Two operations have been added. Their arities are different, one and two, but both operations are of sort bool because this is the only sort declared. By following the information given in the signature above, we can build syntactically correct expressions in the same way as we did from the formation rules in Chapter 1. Thus $\bot$ and $\top$ are elements of sort bool and any syntactically correct application of functions $\neg$ and $\wedge$ results in further expressions in the Herbrand universe. Any examples of correctly formed formulas in the universe are exactly those that might be deduced from the formation rules in Chapter 1, e.g.

$$\bot \qquad \neg\bot \qquad \neg(\neg\bot \wedge \top) \quad \text{etc.}$$

Abstract type equations not only define equivalences, but also provide authority for the replacement of one expression by another. This is a consequence of the substitutive nature of equality mentioned earlier. If the fragment $\neg\neg\top$ occurred in an expression, it could be replaced by the equivalent term $\top$. As a result, the equations of an abstract type may be used in a series of term rewrites to give a simpler canonical or normal term. For example, the expression $\neg(\bot \wedge \top) \wedge \neg\bot$ may be reduced as follows:

$\neg(\bot \wedge \top) \wedge \neg\bot$		
$\neg(\bot) \wedge \neg\bot$	eq 3	
$\neg\neg\top \wedge \neg\bot$	eq 1	(used right to left)
$\top \wedge \neg\bot$	eq 2	
$\top \wedge \neg\neg\top$	eq 1	(used right to left)
$\top \wedge \top$	eq 2	
$\top$	eq 3	

Any expression can be reduced to either $\top$ or $\bot$ with the aid of the equations, allowing us to partition all propositions in the Herbrand universe defined by bool1 into two equivalence classes:

trueclass = $\{\top, \neg\bot, \neg\neg\top, \top \wedge \top\}$

falseclass = $\{\bot, \neg\top, \neg\neg\neg\top\}$

Every element in trueclass is equivalent to $\top$ and is reduced to this canonical value using equations of the abstract type. Since every expression in the class is equivalent, only one of them has to be named in order to characterise the class. As a result, the two classes above may be represented by the congruence classes $[\top]$ and $[\bot]$, and these two elements are said to represent the quotient algebra.

Two different objects might be offered as interpretations of bool1 as follows:

obj1 = ($\{true, false\}$, $\bot = false$, $\top = true$, $\neg = not$, $\wedge = and$)
obj2 = ($\{0, 1\}$, $\bot = 0$, $\top = 1$, $\neg = flip$, $\wedge = min$)

and it would be useful to know how the operations of each representation behave in relation to the abstract type. Take as a simple first example the equation $\neg\top = \bot$ and replace each symbol in this equation with the symbol allocated in each object to give the two equations

$not\ true = false$
$flip\ 1 = 0$

Two symbols on the left of the abstract equation are replaced by their interpretations and, when reduced, produce a result that represents the right-hand symbol. Thus the behaviour of each operation in an interpretation is dictated by the equations of the abstract type. Similarly, the abstract type equation $\neg\neg\top = \top$ leads to two interpretation equations

$not\ not\ true = true$
$flip(flip(1)) = 1$

Moving on to the arity-two operation, we convert the abstract type equation $\top \wedge \bot = \bot$ into the interpretations

$and(true,false) = false$
$min(1,0) = 0$

Suppose now that we want to simplify the expression $\neg(\neg\top \wedge \neg\bot)$, according to the equations of the abstract type. The simplification proceeds as follows:

$\neg(\neg\top \wedge \neg\bot)$	
$\neg(\bot \wedge \neg\bot)$	$\neg\top = \bot$
$\neg(\bot \wedge \top)$	$\neg\bot = \top$
$\neg\bot$	$x \wedge \top = x$
$\top$	$\neg\bot = \top$

Notice that the equation $\neg\bot = \top$ does not occur explicitly in the abstract type and has to be obtained from equations $\neg(\neg\top) = \top$ and $\neg\top = \bot$. A representation of this proposition in obj1 has the form $not(not\ true\ and\ not\ false)$ and might be simplified as follows:

not(*not true and not false*)	
not(*false and not false*)	*not true* = *false*
not(*false and true*)	*not false* = *true*
not false	*x and true* = *x*
true	*not false* = *true*

At each step, equations of the abstract type have been recast in the form of the interpretation, allowing a series of reductions to the interpreted equation. The important point to be observed is that whenever an abstract expression evaluates to $\top$, an interpretation of that expression in obj1 must evaluate to *true*. Similarly, an abstract expression that evaluates to $\bot$ must have a representation in obj1 that reduces to *false*. A cynical observer might note that all we are doing here is using different symbols or names for the same things, and this would be perfectly correct. Initial representations are intended to capture an underlying structure, so the exact symbols used to represent that structure are unimportant. It does not matter whether we simplify an expression before converting it to a specific representation or after; the same answer is always obtained. A similar chain of reasoning allows us to deduce that the canonical value of a related expression in the second object *flip*(*flip* 1 *min flip* 0) is the value 1.

Both interpretations of bool1 provided above are initial interpretations because they define one domain element or domain operation for every operator in the abstract type. The words used to describe the abstract type are unimportant; it is the underlying structure defined by the operations and equations that matters. Every initial object in a class defined by an abstract type is said to be isomorphic – from the Greek meaning the same shape – because it has the same basic internal structure.

A non-initial interpretation in which both $\top$ and $\bot$ are mapped to *true* may be written as follows:

obj3 = ({*true*}, $\bot$ = *true*, $\top$ = *true*, $\neg$ = *not*, $\wedge$ = *and*)

but every expression now evaluates to *true*. This is an acceptable interpretation in the formal sense, but is unlikely to reflect the intention of the original specification. It allows a total collapse of the domain space and is an example of a final interpretation as opposed to an initial interpretation, which allows no collapse of the domain space. Initial and final semantics are the limiting forms of interpretations; anything in between is described as loose semantics. We will only be concerned with initial semantics.

5.3.1 Inheriting an existing specification

Since the intended interpretation of a Boolean theory is Boolean algebra, we would obviously like to add operations that would be interpreted as the usual Boolean connectives. One way of doing this would be to write down a new theory with the

extra operations included, but a better option is to recognise that all the previous theory is included in the new one and only the additions need to be described. An abstract type can inherit all the operations and equations of a preceeding definition simply by including the name of the previous type in the new heading. Thus, an extended Boolean type bool2 includes bool1 as follows:

bool2 = bool1 +
  opns
    $\vee$: bool, bool $\rightarrow$ bool
    $\rightarrow$: bool, bool $\rightarrow$ bool
  eqns
    $x, y \in$ bool
    $x \vee y = \neg(\neg x \wedge \neg y)$
    $x \rightarrow y = \neg x \vee y$

Theory bool2 includes all of bool1 and this is indicated in the opening line bool2 = bool1 +, then the extra operations and their equations are given in a style similar to the original. This form of addition to an existing abstract type is called an enrichment and is obviously a time-saving device. Since our intended interpretation is Boolean algebra and negation and conjunction connectives are an adequate set for the algebra, we should expect that every other connective can be expressed in terms of these two. The added equations do not have to be defined in terms of the previous ones, so we could have chosen to define operation $\vee$ directly as

$x \vee \top = \top$
$x \vee \bot = x$

In summary, an abstract type – sometimes called a theory, a presentation or a specification – has the following general form:

name =
  sorts s
  opns
    $f : s^n \rightarrow s$
    $p : s^n$
  eqns
    variable declarations
    $L = R$

First of all, a name is given to the theory so that it becomes an identifiable unit binding together a number of operations and their properties into useful modules. Large specifications might consist of collections of such theories. Keyword sorts opens the theory, listing the sorts or types of objects being defined in the abstract type. In this chapter we are concerned only with single-sorted, homogeneous abstract types and this line only occurs because it is necessary in the many-sorted, heterogeneous theories of the next chapter. Next we have keyword opns followed by one line for each of the operations or predicates being defined in the abstract

type. These lines describe the sorts of the arguments required by each function or predicate and the sort resulting from the evaluation of an operation. By definition a single-sorted theory has only one sort, so in this case the lines tell us just the arity, i.e. the number of arguments, required by the function or predicate. Constants are seen as zero-arity operations, so they appear as follows:

$c : \rightarrow s$

Each operation name has to be different from every other operation and predicate name in the abstract type. The top part of the definition, consisting of sorts and opns declarations, is called the signature of the abstract type and might be followed by a number of equations. If there are equations, they define equivalences between strings of symbols in a Herbrand universe constructed according to the signature.

EXERCISES 5.3

1. Use the abstract type bool2 to reduce the following expression to its simplest (canonical) value:

   $\neg(\neg(\top \wedge \bot) \vee \neg\top)$

2. An operation called *max* may be defined on symbols 0 and 1 as follows:

   $x\ max\ 1 = x$
   $x\ max\ 0 = 0$

   and this operation has the same relation to $\vee$ as that of *min* to $\wedge$ in bool2. Evaluate the following expression:

   $flip(flip(1\ min\ 0)\ max\ 0)$

3. Define an abstract type called booln with a signature consisting only of the constants $\top$, $\bot$, $\neg$ and $\rightarrow$. Add equations to the abstract type that force objects of the type to behave as indicated by the interpretations of *true*, *false not* and *implication* as described in Chapter 1.

4. Define an abstract type called boolm as an enrichment of booln. The enrichment should include the operation $\leftrightarrow$ with the meaning defined in the truth tables of Chapter 1. It should also include operations $\wedge$, $\vee$ with the meanings defined above.

5.4 A NATURAL THEORY

A first attempt at a theory to describe numbers begins with a fundamental abstract type called nat0 as follows:

nat0 =
  sorts
    nat
  opns
    zero : → nat
    suc : nat → nat

Any theory that consists of a signature without any equations is said to be fundamental because it generates all possible strings of symbols without defining any equivalences between the strings. In this particular case the signature contains an arity-zero operation called *zero* and an arity-one operation called *suc*. These operators generate the following infinite series of expressions:

zero, *suc*(*zero*), *suc*(*suc*(*zero*)), *suc*(*suc*(*suc*(*zero*))), . . .

in the Herbrand universe of the type. As usual, the elements of this universe are all those strings of symbols constructed in accordance with the arity rules of the abstract type. The only well-formed applications of these operators are the constant *zero* itself or successive applications of the *suc* function beginning with *zero*. The Herbrand universe is sometimes called the Herbrand domain, but in the context of abstract types it is often also called the word algebra. Since there are no equations in this theory, every element is distinct and we obtain an infinite number of one-element equivalence classes.

One very obvious interpretation for the possible elements of the abstract type nat0 is the series of denary numbers {0, 1, 2, 3, . . . }, setting *zero* equal to 0, *suc*(*zero*) equal to 1, and so forth, but it is not the only possible choice. Number systems may be developed from any base number in just the same way that the denary system is developed on the base number 10, and the binary and hexadecimal systems based on numbers 2 and 16 are widely used in computer science. The relationship of these possible interpretations with increasingly large strings of the word algebra are then shown as follows:

Word algebra	**Denary**	**Binary**	**Hexadecimal**
zero	0	0	0
suc(*zero*)	1	1	1
suc(*suc*(*zero*))	2	10	2
suc(*suc*(*suc*(*z*)))	3	11	3
suc(*suc*(*suc*(*suc*(*z*))))	4	100	4
suc(*suc*(*suc*(*suc*(*suc*(*zero*)))))	5	101	5
⋮			
$suc^{20}(zero)$	20	10100	14

Each term in the word algebra corresponds to distinct terms in each of the other three interpretations, so there is no confusion. At the same time, there are no junk elements in any of the interpretations, i.e. there are no symbols that do not correspond

to an element of the word algebra. Since there is no junk or confusion, each of the three interpretations can be considered an initial interpretation of the abstract type.

Fundamental abstract types have no equations hence no method of making strings of symbols equivalent. The following extended presentation is a first step towards a more useful theory:

nat1 =
  sorts
    nat
  opns
    zero : → nat
    suc : nat → nat
    add : nat, nat → nat
  eqns
    x, *y* ∈ nat
    add(*zero*,*x*) = *x*
    add(*suc*(*x*),*y*) = *add*(*x*,*suc*(*y*))

An extra operation called *add* has been added, increasing the word algebra to include strings such as *add*(*zero*, *suc*(*zero*)), but the equivalences relating to this operation allow it to be simplified through term rewrites. Consider a term *add*(*suc*(*suc*(*suc*(*zero*))), *suc*(*suc*(*zero*))) and a series of term rewrites following the abstract type equations above:

add(*suc*(*suc*(*suc*(*zero*))), *suc*(*suc*(*zero*)))	
add(*suc*(*suc*(*zero*)), *suc*(*suc*(*suc*(*zero*))))	eq 2
add(*suc*(*zero*), *suc*(*suc*(*suc*(*suc*(*zero*)))))	eq 2
add(*zero*, *suc*(*suc*(*suc*(*suc*(*suc*(*zero*))))))	eq 2
suc(*suc*(*suc*(*suc*(*suc*(*zero*)))))	eq 1

A series of term rewrites using equation 2 of the abstract type gradually moves *suc* operators from the left to the right argument. Eventually the left argument becomes zero, equation 1 becomes applicable and the series terminates with a canonical value. It is clear from this reduction that the original expression is equivalent to the term *suc*(*suc*(*suc*(*suc*(*suc*(*zero*)))))). Otherwise stated, the term belongs to this equivalence class. Of course, additions may be applied to any pair of expressions and the one-element equivalence classes of the fundamental type are increased to include these expressions:

{*zero*, *add*(*zero*,*zero*), *add*(*zero*,*add*(*zero*,*zero*)), . . . }
{*suc*(*zero*), *add*(*zero*,*suc*(*zero*)), *add*(*suc*(*zero*),*zero*)}
{*suc*(*suc*(*zero*)), *add*(*zero*,*suc*(*suc*(*zero*)), *add*(*suc*(*zero*),*suc*(*zero*))}
etc.

Denary, binary and hexadecimal numbers equipped with an addition operation provide initial interpretations for the specifications above. As a result, we defined the three interpretations

obj1 = ({nat} *zero* = 0, *add* = +)
obj2 = ({bin} *zero* = 0, *add* = ◊)
obj3 = ({hex} *zero* = 0, *add* = ♦)

in which {nat}, {bin} and {hex} each represent an infinite series of natural numbers expressed in the appropriate base. The usual + symbol represents denary addition, and in common practice this symbol would also be used for binary and hexadecimal addition. Since we are interested in comparing the operations, and the base will not always be obvious from the context, two new symbols (◊ and ♦) are used to represent binary and hexadecimal addition. Note carefully that operator *add* in the abstract type is defined as a prefix operation whereas we shall use each of the others in the more familar infix style.

The result of the term rewrites above confirms the following equivalence:

$$add(suc(suc(suc(zero))),suc(suc(zero))) = suc(suc(suc(suc(suc(zero)))))$$

and when the operations of this equation are mapped onto the three interpretations, the following representations are obtained:

3 + 2 = 5
11 ◊ 10 = 101
3 ♦ 2 = 5

When the appropriate addition operator is applied to the arguments of a particular representation, the result is the image of that obtained in the abstract equation. Each of the three interpretions above is initial, so each reflects exactly the operations of the abstract type. More importantly, the three interpretations are isomorphic and are in effect three different notations for a common underlying structure. If two denary numbers are added together using denary addition and the result converted to binary, the result is the same as if the numbers had first been converted to binary then subjected to binary addition. It is the isomorphism between the representations that permits this form of interconversion.

5.4.1 Homomorphisms between objects

We now wish to extend the abstract type nat1 with just one additional equation to give new type called mod3:

mod3 = nat1 +
  eqns
    $suc(suc(suc(zero))) = zero$

The effect of this extra equation is to reduce the infinite number of equivalence classes of nat1 to just three, because large strings now reduce to much simpler ones; for example

$suc(suc(suc(suc(zero)))) = suc(zero)$
$suc(suc(suc(suc(suc(zero))))) = suc(suc(zero))$

Every string of symbols in the Herbrand universe then falls into one of the three equivalence classes

$\{zero, suc(suc(suc(zero))), \ldots\}$
$\{suc(zero), suc(suc(suc(suc(zero)))), \ldots\}$
$\{suc(suc(zero)), suc(suc(suc(suc(suc(zero))))), \ldots\}$

This abstract type should be recognisable as an abstraction of the natural numbers modulo 3, more familiar in the interpretation

obj1 = ({0, 1, 2}, add = $\oplus$)

The single equation of the abstract type may be stated in the form 3 = 0 and the derived equations below as 4 = 1 and 5 = 2, but representations of numbers greater than 2 are not necessary in the interpretation. Modulo 3 additions never produce numbers larger than 2, for example

$2 \oplus 2 = 1$
$3 \oplus 5 = 2$

One way of working out a modulo 3 result is to perform the calculation in denary, then convert the result according to Figure 5.6. We just divide by three and take the remainder.

The relationship between denary and mod3 numbers revealed in Figure 5.6 is quite different from that between the three representations of nat1 given earlier. Given a number in any one of the three representations, denary, binary and hexadecimal, it is possible to find the unique equivalent number in either of the other representations. Given a denary number it is possible to find a unique equivalent modulo 3 number from the graph of Figure 5.6, but the reverse process is impossible. For example, the modulo 3 number 1 could be traced back to any infinite series

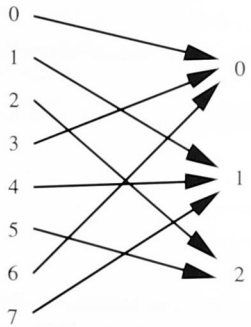

Figure 5.6 Mapping denary to mod 3

of denary numbers 1, 4, 7, . . . rather than to a unique number. This is an example of the confusion property mentioned earlier. A relationship of this kind is called a homomorphism.

A homomorphism is a structure preserving mapping from one object or algebra to another in one direction only. If we write down a small denary expression

$$5 + 6 = 11$$

this could be easily be converted to the mod3 equivalent

$$2 \oplus 0 = 2$$

However, it would not be possible to first convert the denary numbers to mod3, carry out the addition in mod3, then convert the answer back to denary. The result 2 could be mapped to any one of an infinite series of numbers. On the other hand, it is possible to carry out a denary addition first then convert the result to mod3, as opposed to first converting to mod3 then applying the mod3 addition.

Generally, if h is the homomorphism function that maps the domain elements of one interpretation onto another and if f is some operation, then

$$h(f(a_1,a_2, \ldots ,a_n)) = (f'(h(a_1),h(a_2), \ldots ,h(a_n)))$$

For example, a homomorphism mapping the calculation from denary to mod3 is

$$\begin{aligned} h(3 + 2) &= h(3) \oplus h(2) \\ &= 0 \oplus 2 \\ &= 2 \end{aligned}$$

but there is no homomorphism in the reverse direction. An isomorphism essentially consists of homomorphisms working in both directions.

A relational operation may be added to a specification in order to allow comparisons between elements of the type. For example, the specification nateq (natural with equality) allows comparisons between naturals:

nateq = nat1 + bool1 +
  opns
    eq : nat, nat $\rightarrow$ bool
  eqns
    $x, y \in$ nat
    $eq(zero,zero) = true$
    $eq(zero,suc(x)) = false$
    $eq(suc(x),zero) = false$
    $eq(suc(x),suc(y)) = eq(x,y)$

This extension simply says that the successors of two numbers are equal to each other when the numbers, x and y, are themselves equal. Thus strings with equal numbers of successor functions are equal, whereas those with differing numbers of successor functions are unequal. A similar set of equations might be used to define the less than (lt) relation as follows:

$lt(zero,suc(x)) = true$
$lt(zero,zero) = false$
$lt(suc(x),zero) = false$
$lt(suc(x),suc(y)) = lt(x,y)$

EXERCISES 5.4

1. An abstract type may be defined as follows:

integer =
sorts int
opns
$zero$: $\rightarrow$ int
suc : int $\rightarrow$ int
$pred$: int $\rightarrow$ int
eqns
$x, y \in$ int
$pred(suc(x)) = x$
$suc(pred(x)) = x$

   a. List the equivalence classes of this specification.

   b. Simplify the expression $suc(pred(pred(suc(zero))))$.

2. Children sometimes do multiplications by repeated addition, i.e. to work out 8×4 they compute $8 + (8 + (8 + 8))$. Use this technique to define multiplication in terms of an addition operation called *add*. Remember that any number multiplied by 0 is 0.