MATH 3801

Logic

Department of Mathematics
UCL

Lecture Notes
2014-2015

Isidoros Strouthos

April 2015

Contents

Chapter 1

Language

In order to be able to study the structure of mathematical objects and ideas, we will first provide a framework in which mathematical objects can be defined and analysed.

1.1 First order predicate language

The *symbols* which we will use to construct our language consist of:

1) A countably infinite set of *variable symbols*: $\{x_1, x_2, x_3 \cdots\}$ or $\{w, x, y, z, w', x', y', z', \cdots\}$.

2) For each nonnegative integer n, a countably infinite set of *predicate symbols* $\{P_1, P_2, P_3 \cdots\}$ or $\{P, Q, R, P', Q', R', \cdots\}$, each of which has *arity* n. If P is a predicate symbol of arity n, then we call P an n-ary predicate symbol.

3) The symbols $\neg$, $\Rightarrow$, $\forall$.

The set of all *strings* of symbols from the above list is denoted by $\mathcal{L}_{string}$.

For example, suppose that x, y, z, x_1, x_2, x_3 are variable symbols, P is a 1-ary predicate symbol, and Q is a 2-ary predicate symbol. Then x, Px, Pxy, Qx, Qxy, Qxx, xQy, $Qx_1x_2x_3$, $\neg Px$, $\Rightarrow PxQyz$, $\forall x \forall y Qxy$, $\forall x$, $\forall x \Rightarrow \neg PxQxy$, $\forall x \Rightarrow P \neg xQxy$ are all in $\mathcal{L}_{string}$.

Let us now consider the subset of $\mathcal{L}_{string}$ that will be especially useful:

Definition 1.1. *The set of* formulae *in the general first order predicate language is denoted by $\mathcal{L}$, and is defined inductively as follows:*

(1) If P is an n-ary predicate symbol and $x_1, \cdots x_n$ are variable symbols, then $Px_1 \cdots x_n$ is a formula.

(2) If α is a formula, then so is $\neg\alpha$.

(3) If α, β are formulae, then so is $\Rightarrow \alpha\beta$.

(4) If x is a variable symbol and α is a formula, then $\forall x\alpha$ is a formula.

For example, suppose that x, y, z, x_1, x_2, x_3 are variable symbols, P is a 1-ary predicate symbol, and Q is a 2-ary predicate symbol. We consider the strings x, Px, Pxy, Qx, Qxy, Qxx, xQy, $Qx_1x_2x_3$, $\neg Px$, $\Rightarrow PxQyz$, $\forall x\forall yQxy$, $\forall x$, $\forall x \Rightarrow \neg PxQxy$, $\forall x \Rightarrow P\neg xQxy$.

Let us see how we may determine whether or not these strings are formulae. We may do so by trying to (step-by-step) construct each string using rules (1), (2), (3), (4) in Definition 1.1:

- The string x is not a formula:

  There is no rule that allows a single variable symbol to form a formula.

- The string Px is a formula:

  It is an occurrence of (1) in the definition of a formula (note that P is a 1-ary predicate).

- The string Pxy is not a formula:

  This string consists of a single 1-ary predicate symbol, namely P, followed by two variable symbols; this is not allowed. Alternatively, note that Px is a formula, but there is no rule that allows a formula (the formula Px in this case) followed by a variable symbol (the symbol y in this case) to form a formula.

- The string Qx is not a formula:

  This string consists of a 2-ary predicate symbol, namely Q, followed by (only) a single variable symbol, namely x; this is not allowed.

- The string Qxy is a formula:

  It is an occurrence of (1) in the definition of a formula (note that Q is a 2-ary predicate).

- The string Qxx is a formula:

  It is an occurrence of (1) in the definition of a formula (note that Q is a 2-ary predicate).

- The string xQy is not a formula:

  Here, a 2-ary predicate, namely Q, is followed by (only) a single variable symbol, namely y; this is not allowed.

- The string $Qx_1x_2x_3$ is not a formula:

  This string consists of a single 2-ary predicate symbol, namely Q, followed by three variable symbols; this is not allowed. Alternatively, note that Qx_1x_2 is a formula, but there is no rule that allows a formula (the formula Qx_1x_2 in this case) followed by a variable symbol (the symbol x_3 in this case) to form a formula.

- The string $\neg Px$ is a formula:

  We may note that Px is a formula using rule (1); then, $\neg Px$ is a formula using rule (2).

- The string $\Rightarrow PxQyz$ is a formula:

  The strings Px and Qyz are formulae, using rule (1). Therefore, we may use rule (3) in order to deduce that $\Rightarrow PxQyz$ is a formula.

- The string $\forall x \forall y Qxy$ is a formula:

  The string Qxy is a formula, using rule (1). As a result, $\forall y Qxy$ is a formula using rule (4); a further use of rule (4) shows that $\forall x \forall y Qxy$ is a formula.

- The string $\forall x$ is not a formula:

  There is no rule that allows us to obtain a formula consisting (only) of a '$\forall$' symbol followed by a variable symbol.

- The string $\forall x \Rightarrow \neg PxQxy$ is a formula:

  The string Qxy is a formula by rule (1). Similarly, Px is a formula by rule (1), and so $\neg Px$ is a formula by rule (2). Therefore, we can now use rule (3) to deduce that $\Rightarrow \neg PxQxy$ is a formula. Finally, we may use rule (4) in order to deduce that $\forall x \Rightarrow \neg PxQxy$ is a formula.

- The string $\forall x \Rightarrow P\neg xQxy$ is not a formula:

  Note that, if this string were a formula, then the string following the '$\forall x$', namely the string $\Rightarrow P\neg xQxy$, would be a formula, via rule (4). Then, via rule (3), the string following the '$\Rightarrow$', namely the string $P\neg xQxy$, would consist of two formulae.

Here, the string Qxy is a formula by rule (1). However, the 'remaining' string, $P\neg x$, is not a formula, since there is no rule that allows us to have a predicate symbol immediately followed by a '$\neg$' symbol (similarly, there is no rule that allows us to have a variable symbol immediately preceded by a '$\neg$' symbol).

As a result, the string $\Rightarrow P\neg xQxy$ is not a formula, and, hence, the string $\forall x \Rightarrow P\neg xQxy$ is not a formula.

For some more practice regarding the use of the definition of a formula in order to determine whether or not given strings are formulae, please Exercise Set 1 and Sample Exercises 1.

Informally, $\mathcal{L}$ consists of the set of strings of symbols to which we will be able to ascribe 'meaning'.

Note 1.2. *Formulae, particularly the 'basic' ones of the form $Px_1 \cdots x_n$, may also be referred to as propositional functions.*

Note 1.3. *In general, 1-ary and 2-ary predicates are referred to as* unary *and* binary *predicates respectively.*

Note 1.4. *The language is defined as* first order *predicate because the '$\forall$' symbol can only deal with individual variables. So, it can directly deal with statements such as*

" For each real number x, $\cdots$ "

but not with statements of the form

" For each subset of real numbers, $\cdots$ "

This is not too restrictive, as it allows us to express quite a few 'mathematical ideas', but it is a deficiency, as we shall see at the end of section 3.3, when we try to uniquely describe the natural numbers in a language derived from $\mathcal{L}$.

Definition 1.5. *If α is a formula ($\alpha \in \mathcal{L}$), then the* degree *of α, or $deg(\alpha)$, is the nonnegative integer obtained by (starting from 0 and):*

1) *adding 1 each time $\neg$ occurs in α;*
2) *adding 2 each time $\Rightarrow$ occurs in α;*
3) *adding 1 each time $\forall$ occurs in α.*

For example, if x, y are variable symbols, P is a unary predicate symbol and Q is a binary predicate symbol, then:

$$deg(Px) = 0$$

$$deg(\neg Px) = 1$$

$$deg(\forall xQxy) = 1$$

$$deg(\Rightarrow PxQxy) = 2$$

$$deg(\Rightarrow \neg Px\forall xQxy) = 4$$

Observe that, in general, if R is an n-ary predicate symbol, and $x_1, \cdots, x_n$ are variables, then $Rx_1 \cdots x_n$ has degree 0.

Informally, the degree counts the number of 'substructures' in a formula.

Note 1.6. *The degree of a formula provides an ordered structure to $\mathcal{L}$, and is particularly useful when proving certain results about 'the whole of' $\mathcal{L}$.*

Two notable absentees from the symbols on which $\mathcal{L}$ is based are the left and right brackets, '(' and ')'; their absence is especially notable because we usually include these symbols to describe the order in which 'things are done' and to remove ambiguity from a 'meaningful' statement.

For example, in our usual mathematical language, writing $\alpha \Rightarrow \beta \Rightarrow \gamma$ is ambiguous: we could be referring to $(\alpha \Rightarrow \beta) \Rightarrow \gamma$ or to $\alpha \Rightarrow (\beta \Rightarrow \gamma)$.

However, in $\mathcal{L}$, this is not a problem, because the two possibilities are written differently: $(\alpha \Rightarrow \beta)$ is written as $\Rightarrow \alpha\beta$ in $\mathcal{L}$, so that

$$(\alpha \Rightarrow \beta) \Rightarrow \gamma \quad \text{is written as} \quad \Rightarrow\Rightarrow \alpha\beta\gamma \quad \text{in } \mathcal{L}$$

whereas

$$\alpha \Rightarrow (\beta \Rightarrow \gamma) \quad \text{is written as} \quad \Rightarrow \alpha \Rightarrow \beta\gamma \quad \text{in } \mathcal{L}$$

We shall see more examples of the transformation of statements like these from one form to another in section 1.2 (e.g. see Example 1.15).

In fact there is, in general, no need for brackets in $\mathcal{L}$, as we now aim to show.

Definition 1.7. *For s a string in $\mathcal{L}_{string}$, the weight of s, written as $weight(s)$, is the integer obtained by taking the sum of:*

1) -1 *for each (occurrence of a) variable symbol in s;*
2) $n-1$ *for each (occurrence of an) n-ary predicate symbol in s;*
3) 0 *for each (occurrence of) $\neg$ in s;*
4) $+1$ *for each (occurrence of) $\Rightarrow$ in s;*
5) $+1$ *for each (occurrence of) $\forall$ in s.*

For example, if x, y are variable symbols, P is a unary predicate symbol and Q is a binary predicate symbol, then:

$$weight(Px) = 0 + (-1) = -1$$
$$weight(\neg Px) = 0 + 0 + (-1) = -1$$
$$weight(Qxy) = 1 + (-1) + (-1) = -1$$
$$weight(\Rightarrow PxQxy) = -1$$
$$weight(\forall x) = 0$$
$$weight(\neg PxQ \Rightarrow\Rightarrow \neg) = 2$$
$$weight(xyQ) = -1$$

Proposition 1.8. *If α is a* formula*, i.e. if $\alpha \in \mathcal{L}$, then $weight(\alpha) = -1$.*

Proof. Let us prove this by induction on the degree of α.

If $deg(\alpha) = 0$, then α is of the form $Px_1 \cdots x_n$ for an n-ary predicate symbol P. Then

$$weight(\alpha) = (n-1) - n = -1$$

as required.

Suppose that the result holds for all formulae of degree smaller than or equal to n, and consider a formula α such that $deg(\alpha) = n+1$. We show that $weight(\alpha) = -1$.

By the definition of $\mathcal{L}$, α must have one of the following forms:

1) α is of the form $\neg\alpha_1$ for some formula α_1.

   Then, by considering degrees: $deg(\alpha) = deg(\neg\alpha_1) = 1 + deg(\alpha_1)$, so that $deg(\alpha_1) = n$. So, we can use the inductive hypothesis to deduce that $weight(\alpha_1) = -1$. Then,

$$weight(\alpha) = weight(\neg\alpha_1) = weight(\neg) + weight(\alpha_1) = 0 + (-1) = -1$$

2) α is of the form $\Rightarrow \alpha_1\alpha_2$ for some formulae α_1 and α_2.

   Then, as above, $deg(\alpha) = 2 + deg(\alpha_1) + deg(\alpha_2)$, so that $deg(\alpha_1) + deg(\alpha_2) = n - 1$, i.e. $deg(\alpha_1) < n$ and $deg(\alpha_2) < n$. Hence, using the inductive assumption, we may deduce that $weight(\alpha_1) = -1$ and $weight(\alpha_2) = -1$, and so:

$$weight(\alpha) = weight(\Rightarrow \alpha_1\alpha_2) = weight(\Rightarrow)+weight(\alpha_1)+weight(\alpha_2) = 1+(-1)+(-1) = -1$$

3) α is of the form $\forall x\alpha_1$ for some variable symbol x and formula α_1.

   Then $deg(\alpha) = 1 + deg(\alpha_1)$, so that $deg(\alpha_1) = n$. Therefore, inductively, $weight(\alpha_1) = -1$, and so

$$weight(\alpha) = weight(\forall x\alpha_1) = weight(\forall)+weight(x)+weight(\alpha_1) = 1+(-1)+(-1) = -1$$

So, in each of the three cases, we have shown that $weight(\alpha) = -1$, as required. □

Note 1.9. *We have not proven that* any string *of symbols with weight* -1 *is a formula; this does not hold in general. For example, for a unary predicate symbol* P *and variable symbols* x, y, z*:*

$$weight(x) = -1$$

$$weight(\Rightarrow xyz\forall) = -1$$

$$weight(xPx\forall) = -1$$

but none of the considered strings is a formula.

Proposition 1.10. *Suppose that* α *is (a formula) in* $\mathcal{L}$ *and that* α *is the concatenation* $\beta\gamma$*, for (nonempty)* β, $\gamma \in \mathcal{L}_{string}$ *(so that, as a string,* α *can be obtained by writing the string* β *followed, on the right, by the string* γ*).*

Then $weight(\beta) \geq 0$.

As a result, no proper initial segment *of a formula is a formula.*

Proof. Let us prove this by induction on the degree of α.

If $deg(\alpha) = 0$, then α is of the form $Px_1 \cdots x_n$ for an n-ary predicate symbol P. Then, $\beta = Px_1 \cdots x_m$, where $m < n$, and

$$weight(\beta) = (n-1) - m = (n-m) - 1 \geq 0$$

as required (since $n - m \geq 1$, by assumption).

(We note that if P is a 0-ary predicate symbol, then the formula P, of degree 0, has no proper initial segments, so the result vacuously holds; i.e. there is 'nothing to prove' in this case.)

Suppose the result holds for all formulae of degree smaller than or equal to n, and consider a formula α such that $deg(\alpha) = n + 1$. We show that, if α is the concatenation $\beta\gamma$, then $weight(\beta) \geq 0$.

Given the definition of $\mathcal{L}$, we have the following possibilities for α:

1) α is of the form $\neg\alpha_1$ for some formula α_1.

   Then, by considering degrees: $deg(\alpha_1) = n$, so that we may use the inductive hypothesis to deduce that any initial segment of α_1 has nonnegative weight. Also, we may use Proposition 1.8 to deduce that $weight(\alpha_1) = -1$.

   The choices for the proper initial segment β are:

   (i) $\beta = \neg$: Then, $weight(\beta) = weight(\neg) = 0$

   (ii) $\beta = \neg\epsilon$, where ϵ is a proper initial segment of α_1: Then, $weight(\epsilon) \geq 0$, by the inductive assumption, and

   $$weight(\beta) = weight(\neg) + weight(\epsilon) = 0 + weight(\epsilon) \geq 0$$

2) α is of the form $\Rightarrow \alpha_1\alpha_2$ for some formulae α_1 and α_2.

   Then, as above, $deg(\alpha) = 2 + deg(\alpha_1) + deg(\alpha_2)$, so that $deg(\alpha_1) < n$ and $deg(\alpha_2) < n$. Hence, by the inductive hypothesis, any initial segment of α_1 or α_2 has nonnegative weight. Also, by Proposition 1.8: $weight(\alpha_1) = -1$, $weight(\alpha_2) = -1$.

   The choices for β are:

   (i) $\beta = \Rightarrow$: Then, $weight(\beta) = 1 \geq 0$

   (ii) $\beta = \Rightarrow \epsilon$, for ϵ a proper initial segment of α_1: Then, $weight(\epsilon) \geq 0$, so that

   $$weight(\beta) = 1 + weight(\epsilon) > 0$$

(iii) $\beta = \Rightarrow \alpha_1$: Then, we may use Proposition 1.8 to deduce that $weight(\alpha_1) = -1$, so that

$$weight(\beta) = 1 + (-1) = 0$$

(iv) $\beta = \Rightarrow \alpha_1\epsilon$, for ϵ a proper initial segment of α_2: Then, $weight(\epsilon) \geq 0$, so that

$$weight(\beta) = 1 + weight(\alpha_1) + weight(\epsilon) = 1 + (-1) + weight(\epsilon) = weight(\epsilon) \geq 0$$

3) α is of the form $\forall x\alpha_1$ for some variable symbol x and formula α_1.

Then, $deg(\alpha) = 1 + deg(\alpha_1)$, so that $deg(\alpha_1) = n$. So, inductively, we may assume that any initial segment of α_1 has nonnegative weight. Also, using Proposition 1.8, we may deduce that $weight(\alpha_1) = -1$.

The choices for β are:

(i) $\beta = \forall$: Then, $weight(\beta) = 1 \geq 0$

(ii) $\beta = \forall x$: Then, $weight(\beta) = 1 + (-1) = 0$

(iii) $\beta = \forall x\epsilon$, for ϵ a proper initial segment of α_1: Then, by the inductive assumption,

$$weight(\beta) = 1 + (-1) + weight(\epsilon) = weight(\epsilon) \geq 0$$

So, in each of the three cases, we have shown that $weight(\beta) \geq 0$, as required, i.e. we have shown that no proper initial segment of α has negative weight; we may then use Proposition 1.8 in order to deduce that, if α is a formula, then no proper initial segment of α is a formula, as required.

□

This means that a computer program, say, reading a formula in $\mathcal{L}$ as a piece of code, can use the weight function to check if and when it has reached the end of a formula. It can also apply the same principle when reading formulae present *inside* other formulae.

On that level, the presence of this internal order in $\mathcal{L}$ is a useful asset. However, possibly due to the conventions we (naturally) use in logic, it can also appear quite confusing. We tend to write $\alpha \Rightarrow \beta$ for what is written as $\Rightarrow \alpha\beta$ in $\mathcal{L}$ and we also tend to use other logical connectives and other conventions.

For example, if we define equality using a binary predicate '=', then the statement we usually write as $x = 2$ would be written as $= x2$ in $\mathcal{L}$, since, in $\mathcal{L}$, all the variables involved in a predicate relation must appear to the right of the predicate symbol used.

We would like to be free to use some binary predicates such as '=' or '$\leq$', as well as the '$\Rightarrow$' symbol, as they commonly appear in mathematics; we would also like to use other basic logical connectives which signify, for example, 'and' and 'and/or'.

Furthermore, in many mathematical 'systems', such as groups, rings, and fields, we deal with operations or functions. For example, we might wish to define a function f such that $f(x)$ is x^2.

It *is* possible to do this using predicates. For example, we could use a binary predicate F, so that Fxy is 'true' in our system if and only if $x^2 = y$. This makes it possible to define and completely encapsulate functions using only predicates, and statements that may 'become' true or false in a given mathematical system (e.g. for the example described above, $F(5, 25)$ would be true, whereas $F(5, 27)$ would be false).

However, once again, we would like to also be able to invoke the conventions we use when defining functions in order to be able to write down something like $f(x)$, while knowing that we could also achieve the same by using only predicates from $\mathcal{L}$.

So, while keeping in mind that the general first order predicate language is 'powerful' enough to support the mathematical ideas / constructions mentioned above, we now define a more 'flexible' form of this language, especially suited to the way we tend to write down mathematics.

1.2 Conventional functional first order predicate language

The *symbols* we will use to construct this more 'workable' (version of) $\mathcal{L}$ will be the following:

1) A countably infinite set of *variable symbols*: $\{x_1, x_2, x_3 \cdots\}$ or $\{w, x, y, z, w', x', y', z', \cdots\}$.

2) For each nonnegative integer *arity* n, a countably infinite set of n*-ary predicate symbols*, $\{P_1, P_2, P_3 \cdots\}$ say.

3) For each nonnegative integer *arity* n, a countably infinite set of n*-ary functional symbols*, $\{F_1, F_2, F_3 \cdots\}$ say.

4) The symbols $\neg$, $\Rightarrow$, $\forall$, $(,)$.

Definition 1.11. *The set of* formulae *in the general conventional functional first order predicate language is denoted by* $\mathcal{L}_{math}$*, and is defined inductively as follows:*

0) Each variable symbol is a variable.

1) If F *is an* n*-ary functional symbol and* $x_1, \cdots x_n$ *are variables, then* $Fx_1 \cdots x_n$ *is a variable.*

2) If P *is an* n*-ary predicate symbol and* $x_1, \cdots x_n$ *are variables, then* $Px_1 \cdots x_n$ *is a formula.*

3) If α *is a formula, then so is* $\neg\alpha$*.*

4) If α, β are formulae, then so is $\Rightarrow \alpha\beta$.

5) If x is a variable symbol and α is a formula, then $\forall x\alpha$ is a formula.

Importantly, in $\mathcal{L}_{math}$, we will allow ourselves the use of the following conventions (for a variable x and formulae α and β):

'$\alpha \Rightarrow \beta$' to denote $\Rightarrow \alpha\beta$

'$\alpha \vee \beta$' to denote $(\neg\alpha) \Rightarrow \beta$ (or $\Rightarrow \neg\alpha\beta$ in $\mathcal{L}$)

'$\alpha \wedge \beta$' to denote $\neg(\alpha \Rightarrow (\neg\beta))$ (or $\neg \Rightarrow \alpha\neg\beta$ in $\mathcal{L}$)

'$\alpha \Leftrightarrow \beta$' to denote $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

'$\exists x\alpha$' (or '$(\exists x)\alpha$') to denote $\neg\forall x\neg\alpha$.

Furthermore, in $\mathcal{L}_{math}$, we will often write down basic binary, predicate or functional, expressions in a more 'natural' form, so that 'xAy' will denote Axy for a binary predicate or functional A, e.g. if we use the predicate '$=$', then we will be able to write $= xy$ as '$x = y$'. In the same spirit of 'naturalness', for an n-ary, predicate or functional, symbol A, we will allow the use of '$A(x_1, \cdots, x_n)$' to denote $Ax_1 \cdots x_n$.

Note 1.12. *In specific examples of mathematical 'theories' or 'systems', we will use Π to denote the set of predicates we will be using and Ω to denote the set of functionals we will be using.*

Note 1.13. *When we assign meaning to the symbols in our language, in the cases of specific mathematical systems, a predicate of arity n may be thought of as a statement on n variables which can 'be' true or false depending on which values we input, whereas a functional of arity n may be thought of as a function on n variables which returns a single variable answer (it is itself a variable in this sense).*

For example, for a variable x, a unary predicate symbol P and a unary functional symbol F, Px is a statement that can be either true or false, e.g. 'x is even', whereas Fx is a 'value' or 'object' in the 'mathematical world we are living in' e.g. Fx could 'return the value of' x^2.

These ideas will become more precise (and hopefully include fewer apostrophes) in chapter 3 (see section 3.1 in particular).

Note 1.14. *In the case of a functional of arity 0, we have a function that has no arguments, but which returns 'an answer', i.e. it is always the same, with no dependence on variables: it is constant. So, in practice, functionals of arity 0 are especially useful if and when we wish to introduce distinguished constants (such as the identity element in a group).*

Let us now see an example that will hopefully help us better understand, and appreciate the contrast between, $\mathcal{L}$ and $\mathcal{L}_{math}$.

Example 1.15. *A* poset*, or partially ordered set, is a set X, together with two relations, $=$ and $\leq$, satisfying:*

1) For all x in X: $x \leq x$.

2) For all x and y in X: if $x \leq y$ and $y \leq x$, then $x = y$.

3) For all x, y, z in X: if $x \leq y$ and $y \leq z$, then $x \leq z$.

Examples of posets are the sets $\mathbb{N}$, $\mathbb{Z}$ with 'less than or equal to' 'being' the relation '$\leq$' above, or the set of all subsets of the natural numbers, $\mathbb{P}(\mathbb{N})$, with 'inclusion of subsets', $\subseteq$, 'being' the operation '$\leq$' above.

Let us now write down the three defining statements of posets in $\mathcal{L}_{math}$. Even though $\mathcal{L}_{math}$ includes quite a few helpful conventions, we are not allowed to include words of explanation in our statements, as we did above, or use commas (e.g. we cannot write '$\forall x, y$'), so we must take some care. Furthermore, we do not need to give the set of elements a name (as in X above). We are only, for now, trying to express the three statements as formulae in $\mathcal{L}_{math}$, without each formula necessarily 'meaning' something or 'holding' in some set.

The definition of posets includes no operations; the set of functionals we will use is empty, i.e.

$$\Omega = \emptyset$$

whereas we will be using two binary predicates, '$=$' and '$\leq$', so that the set of predicates is

$$\Pi = \{=, \leq\}$$

Then, the three statements defining posets become:

1) $(\forall x)(x \leq x)$

2) $(\forall x)(\forall y)(((x \leq y) \wedge (y \leq x)) \Rightarrow (x = y))$

3) $(\forall x)(\forall y)(\forall z)(((x \leq y) \wedge (y \leq z)) \Rightarrow (x \leq z))$

So, writing down the statements in $\mathcal{L}_{math}$ seems to keep them 'reasonably readable'.

Let us now write down the same statements in $\mathcal{L}$. Remember that we cannot use brackets, $\vee$, or $\wedge$ in $\mathcal{L}$, while we must write $\Rightarrow \alpha\beta$ whenever we want to 'say' $\alpha \Rightarrow \beta$, and $\leq xy$ whenever we want to 'say' $x \leq y$ (similarly for '$=$': predicate symbols must appear on the left of 'their' variables in $\mathcal{L}$). If we do not use these conventions and, in addition, transform all occurrences of $\vee$ and $\wedge$ to the formulae in $\mathcal{L}$ they correspond to, we can arrive at the (probably) less readable statements:

1) $\forall x \leq xx$

2) $\forall x \forall y \Rightarrow \neg \Rightarrow \leq xy \neg \leq yx = xy$

3) $\forall x \forall y \forall z \Rightarrow \neg \Rightarrow \leq xy \neg \leq yz \leq xz$

So, hopefully, it is clear from even this relatively simple example (simple in terms of the number of predicates and the number and length of statements that we used) that, even though we could work at the level of $\mathcal{L}$, working with the $\mathcal{L}_{math}$ 'version' is quite helpful, as it mimics, to some extent, the way we usually write down mathematical statements (nowadays).

Let us see another example of the defining statements of a mathematical structure being written in $\mathcal{L}_{math}$:

Example 1.16. *A group is a set G together with an operation $\circ$ satisfying:*

1) If $x, y \in G$, then $x \circ y \in G$.

2) There exists an element $e \in G$ such that, for all $x \in G$, $e \circ x = x = x \circ e$.

3) For each $x \in G$, there exists a $y \in G$ such that $x \circ y = e = y \circ x$.

4) For all $x, y, z \in G$: $(x \circ y) \circ z = x \circ (y \circ z)$.

Let us now try to write down these defining statements in $\mathcal{L}_{math}$.

As in the case of posets, we do not need to give the set of elements a name or refer to it. In chapter 3 (e.g. see section 3.1), we will see that, in practice, we will 'impose' the set, which is due to form a group in this case, 'onto' the axioms and assume that the axioms hold precisely in that set; it will become the 'structure' we will be working in (e.g. see Definition 3.1). As such, there is no need to explicitly state a closure axiom (such as the one expressed in the first defining statement above).

Furthermore, in order to aid our understanding, we will define, apart from a binary predicate '$=$', a binary functional, M say, to represent the group operation, as well as a 0-ary functional E (i.e. a

constant) to denote or represent the group identity, 'e', thus making our statements simpler (e.g. we do not need to mention that the identity exists once we have already supplied the functional E).

So, for a group, let us take the set Π of predicates to contain only the binary predicate '$=$', and the set Ω of functionals to contain the binary 'group operation functional' M (so that '$x \circ y$' becomes Mxy or $M(x, y)$ in $\mathcal{L}_{math}$) together with the 0-ary functional E. Then, the statements defining a group become:

1) This 'closure' statement will be vacuously satisfied when we specify our 'structure' in any example (see discussion above), so we need not worry about it for now.

2) $(\forall x)((M(E, x) = x) \wedge (M(x, E) = x))$

3) $(\forall x)(\exists y)((M(x, y) = E) \wedge (M(y, x) = E))$

4) $(\forall x)(\forall y)(\forall z)((M(M(x, y), z)) = (M(x, M(y, z))))$

These are the kinds of structures we shall be dealing with more in chapter 3, in the setting of $\mathcal{L}_{math}$.

Having introduced the conventions for '$\exists$', '$\vee$', '$\wedge$', '$\Rightarrow$' and binary predicates, writing statements in $\mathcal{L}_{math}$ can be ambiguous, unlike in $\mathcal{L}$.

For example, as described earlier, the statement $\alpha \Rightarrow \beta \Rightarrow \gamma$ is ambiguous, as we could interprete it as $(\alpha \Rightarrow \beta) \Rightarrow \gamma$ or $\alpha \Rightarrow (\beta \Rightarrow \gamma)$.

In order to avoid using too many brackets, authors in logic often adopt some agreed conventions which allow them to remove some brackets from an expression and then restore them as required.

In this course, we will always try to have enough brackets in a statement so that the order we 'form the connections' is unambiguous. Therefore, there should be no need for us to use such a convention in practice. However, such conventions do sometimes appear in books and papers on logic.

Let us, as an aside, outline one such convention. Suppose, for example, that we use the following 'bracket removing' algorithm to remove (some of the) brackets from an expression (when the original form of the expression contains enough brackets to not be ambiguous):

1) Find the leftmost $\Leftrightarrow$, if any, and, if it is not nested within another connective, remove the brackets from the two formulae it applies to. Repeat for the next leftmost $\Leftrightarrow$ and so on.

2) Do the same for $\Rightarrow$.

3) Do the same for $\vee$.

4) Do the same for $\wedge$.

5) Do the same for $\neg$ (here removing the brackets from the single formula it applies to).

We could then restore brackets to obtain the original expression, by using the following 'bracket restoring' algorithm:

1) Find the rightmost $\neg$, if any, and, if there are no brackets present to signify its 'scope', insert brackets identifying the smallest formula it could apply to. Repeat for the next rightmost $\neg$ and so on.

2) Find the rightmost $\wedge$, if any, and, if there are no brackets present to signify its 'scope', insert brackets identifying the two smallest formulae it could apply to (one on either side of the $\wedge$). Repeat for the next rightmost $\wedge$ and so on.

3) Do the same for $\vee$.

4) Do the same for $\Rightarrow$.

5) Do the same for $\Leftrightarrow$.

As an example, consider the following statements:

1) $((\alpha \wedge \beta) \wedge (\gamma \wedge \delta))$

2) $(\alpha \wedge ((\beta \wedge \gamma) \wedge \delta))$

3) $(\alpha \wedge (\beta \wedge (\gamma \wedge \delta)))$

After applying the 'bracket removing' algorithm to each of the above statements, we obtain the following forms:

1) $(\alpha \wedge \beta) \wedge \gamma \wedge \delta$

2) $\alpha \wedge (\beta \wedge \gamma) \wedge \delta$

3) $\alpha \wedge \beta \wedge \gamma \wedge \delta$

Observe that the algorithm does not necessarily remove all brackets from an expression.

We should be able to retrieve the original expression in each case, by applying the 'bracket restoring' algorithm. So, in a textbook for example, one might often find the version of a statement that we obtain after removing some brackets, and the reader may need to apply the 'bracket restoring' algorithm in order to return to the original unambiguous form of the statement, e.g. the reader may have to convert $(\alpha \wedge \beta) \wedge \gamma \wedge \delta$ 'back' to $((\alpha \wedge \beta) \wedge (\gamma \wedge \delta))$.

As another example, we can use the above algorithm to remove brackets from the expression

$$(((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma))$$

obtaining

$$((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow \alpha \vee \beta \Rightarrow \gamma$$

to which we can apply the 'bracket restoring' algorithm to return to the original version of the expression.

As stated though, in this course, we will try to avoid the use of such a convention, at the cost of a few more brackets in some of our statements.

We have now defined the framework in which we will be able to study some mathematical ideas and/or objects, such as posets, groups etc. In chapter 3, we will look at particular cases of how the general first order predicate language can be related to structures such as posets, groups, the natural numbers, and others (e.g. see Examples 3.15, 3.16, 3.17 in section 3.1, and see also section 3.3).

Before we do so, however, let us introduce a reduced version of our language, which will allow us to introduce the concepts of truth and provability, and consider the interplay between them, in a setting simpler than general first order predicate logic. This interplay, between the concepts of truth and provability, has been a particularly interesting aspect of (modern) mathematical logic.

Chapter 2

Propositional Logic

In this chapter, we shall work with a reduced version of the general first order predicate language, which we obtain by ignoring the presence of variables in our language. As a result, the language will not contain predicates of arity greater than 0, functionals, or the symbols '$\forall$' and '$\exists$'.

So, all predicates in this new language will be 'atomic' or indivisible formulae, referred to as propositions and capable of being assigned the 'tag' of true or false in a given setting.

The *symbols* we will use are:

1) A countably infinite set of *primitive propositions* $\{P_1, P_2, P_3 \cdots\}$ or $\{P, Q, R, P', Q', R', \cdots\}$, denoted by $\mathcal{L}_0^P$.
2) The symbols $\neg$, $\Rightarrow$, (,).

The formulae in this language are referred to as *propositions* (e.g. compare with Definition 1.1).

Definition 2.1. *The set of propositions, denoted by $\mathcal{L}_0$, is defined inductively as follows:*

1) Every primitive proposition is a proposition, i.e. if $P \in \mathcal{L}_0^P$, then $P \in \mathcal{L}_0$.
2) If α is a proposition, then so is $(\neg\alpha)$.
3) If α, β are propositions, then so is $(\alpha \Rightarrow \beta)$.

Furthermore, we shall keep the conventions of $\mathcal{L}_{math}$ which apply to this language. In other words, we shall use brackets, where necessary, as described above (although we will often omit the outer-

most pair of brackets), and shall introduce the 'shorthand' conventions:

$$\text{`}\alpha \vee \beta\text{'} := (\neg\alpha) \Rightarrow \beta \quad , \quad \text{`}\alpha \wedge \beta\text{'} := \neg(\alpha \Rightarrow (\neg\beta)) \quad , \quad \text{`}\alpha \Leftrightarrow \beta\text{'} := (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$$

Note that the notion of *degree* from chapter 1 (see Definition 1.5) carries over to $\mathcal{L}_0$, and the setting of propositions.

2.1 Semantic aspects of propositional logic

We may now describe an assignment of truth values, as we get closer to using our language to describe mathematical systems (so far, we have simply described symbols and formulae in various languages). In our simplified language, we can assign truth (or falsehood) to primitive propositions and build up the truth of more complicated propositions (i.e. ones of degree greater than 0) using 'sensible' rules:

Definition 2.2. *A* valuation *is a function* $v : \mathcal{L}_0 \to \{0, 1\}$*, satisfying the following conditions:*

1) *For* $\alpha \in \mathcal{L}_0$, $v(\neg\alpha) = 1$ *if* $v(\alpha) = 0$*, while* $v(\neg\alpha) = 0$ *if* $v(\alpha) = 1$*, i.e.* $v(\neg\alpha) = 1 - v(\alpha)$.

2) *If* $\alpha, \beta \in \mathcal{L}_0$*, then* $v(\alpha \Rightarrow \beta) = 0$ *if* $v(\alpha) = 1$ *and* $v(\beta) = 0$*, while* $v(\alpha \Rightarrow \beta) = 1$ *otherwise.*

Note 2.3. *We shall use* '0' *to denote the falsehood of a statement, and* '1' *to denote the truth of a statement.*

The two 'restrictions' mean that we cannot arbitrarily assign the value 0 or 1 to any proposition. However, we can fix our 'potential' valuation on the primitive propositions (in any way we like) and then proceed to uniquely 'extend' this to a valuation on the whole of $\mathcal{L}_0$.

Let us now show that "a valuation is determined by its values on the primitive propositions":

Proposition 2.4. *If there are valuations* v, v' *satisfying* $v(\alpha) = v'(\alpha)$ *for all* $\alpha \in \mathcal{L}_0^P$*, then* $v = v'$*, i.e.* $v(\alpha) = v'(\alpha)$ *for all* $\alpha \in \mathcal{L}_0$.

Proof. We will use the degree of a proposition to construct an inductive proof of the above. If $deg(\alpha) = 0$, then $\alpha \in \mathcal{L}_0^P$, so the result holds by assumption, i.e. $v(\alpha) = v'(\alpha)$ if $deg(\alpha) = 0$.

Now, suppose that the result holds for all propositions of degree smaller than or equal to n. Consider a proposition, α, which has degree $n+1$. Either $\alpha = \neg\alpha_1$ for some proposition α_1, or $\alpha = \alpha_1 \Rightarrow \alpha_2$ for some propositions α_1, α_2.

In the former case, $deg(\alpha) = 1 + deg(\alpha_1)$ (since the presence of '$\neg$' adds 1 to the overall degree). So, $deg(\alpha_1) = n$, and, by the inductive hypothesis, $v(\alpha_1) = v'(\alpha_1)$, so that $v(\alpha) = 1 - v(\alpha_1) = 1 - v'(\alpha_1) = v'(\alpha)$ i.e. $v(\alpha) = v'(\alpha)$, as required.

In the latter case, we proceed similarly: if $\alpha = \alpha_1 \Rightarrow \alpha_2$, then $deg(\alpha) = 2 + deg(\alpha_1) + deg(\alpha_2)$, so that $deg(\alpha_1) < n$ and $deg(\alpha_2) < n$. Hence, by the inductive hypothesis, the valuations agree on the 'constituent' parts of α: $v(\alpha_1) = v'(\alpha_1)$ and $v(\alpha_2) = v'(\alpha_2)$. Therefore, by the definition of a valuation, $v(\alpha) = v'(\alpha)$, as required. □

The following result also holds:

Proposition 2.5. *For any function $f : \mathcal{L}_0^P \to \{0, 1\}$, there exists a valuation $v : \mathcal{L}_0 \to \{0, 1\}$ extending f, i.e. such that $v(\alpha) = f(\alpha)$ for all $\alpha \in \mathcal{L}_0^P$.*

Proof. We will use the degree of a proposition to inductively construct a suitable valuation v, 'starting from' f. We start by setting $v(\alpha) = f(\alpha)$ for all $\alpha \in \mathcal{L}_0^P$. This ensures that $v(\alpha)$ has been defined if $deg(\alpha) = 0$ i.e. if $\alpha \in \mathcal{L}_0^P$.

Now, suppose that the result holds for all propositions of degree smaller than or equal to n, i.e. suppose that we have already defined the valuation v for all propositions of degree smaller than or equal to n. Consider a proposition, α, which has degree $n+1$. Either $\alpha = \neg\alpha_1$ for some proposition α_1, or $\alpha = \alpha_1 \Rightarrow \alpha_2$ for some propositions α_1, α_2.

In the former case, $deg(\alpha) = deg(\alpha_1) + 1$ (since the presence of '$\neg$' adds 1 to the overall degree). So, $deg(\alpha_1) = n$ and $v(\alpha_1)$ has been defined, in which case we may simply define $v(\alpha)$ to be equal to $1 - v(\alpha_1)$.

In the latter case, we proceed similarly: if $\alpha = \alpha_1 \Rightarrow \alpha_2$, then $deg(\alpha_1) < n$ and $deg(\alpha_2) < n$. Hence, by the inductive hypothesis, the valuation v is (well) defined on α_1 and α_2 and we can therefore define $v(\alpha)$ to be 0 if $v(\alpha_1) = 1$ and $v(\alpha_2) = 0$, and to be 1 otherwise.

Then, the construction of the function $v : \mathcal{L}_0 \to \{0, 1\}$, described above, ensures that v is a valuation, while, by construction, $v(\alpha) = f(\alpha)$ for all $\alpha \in \mathcal{L}_0^P$, as required. □

So, "a valuation is determined by its values on the primitive propositions, and any values will do".

Note 2.6. *Observe that, since we started with a* countable *set of primitive propositions (those of degree 0), the set of propositions we construct for each subsequent degree is also countable, and, as a result, the set of all propositions, $\mathcal{L}_0$, is also countable.*

Let us now give the definitions of some concepts related to valuations:

Definition 2.7. *If, for a valuation v, $v(\alpha) = 1$ for a proposition $\alpha \in \mathcal{L}_0$, then we say that α is* true *in/for v, or that v is a* model *of α.*

If, for a set of propositions S and a valuation v, $v(\alpha) = 1$ for all $\alpha \in S$, then we say that v is a model *of S.*

Definition 2.8. *If a proposition $\alpha \in \mathcal{L}_0$ satisfies $v(\alpha) = 1$ for every possible valuation v, then we say that α is a* tautology*, and write $\models \alpha$.*

A well known construction that allows us to investigate whether a proposition is a tautology or not, or generally to check for which valuations a proposition is true, is the construction of a table which evaluates the proposition on all possible valuations that could be defined on its constituent 'primitive parts'. We call such a table a *truth table*, and (will) usually assume the following rules when constructing such tables (all these rules are derivable from the definition of a valuation):

- For $\alpha \in \mathcal{L}_0$, $v(\neg\alpha) = 1$ if $v(\alpha) = 0$ while $v(\neg\alpha) = 0$ if $v(\alpha) = 1$.
- If $\alpha, \beta \in \mathcal{L}_0$, then $v(\alpha \Rightarrow \beta) = 0$ if $v(\alpha) = 1$ and $v(\beta) = 0$ while $v(\alpha \Rightarrow \beta) = 1$ otherwise.
- If $\alpha, \beta \in \mathcal{L}_0$, then $v(\alpha \vee \beta) = 0$ if $v(\alpha) = 0$ and $v(\beta) = 0$ while $v(\alpha \vee \beta) = 1$ otherwise.
- If $\alpha, \beta \in \mathcal{L}_0$, then $v(\alpha \wedge \beta) = 1$ if $v(\alpha) = 1$ and $v(\beta) = 1$ while $v(\alpha \wedge \beta) = 0$ otherwise.

If two statements have the same truth table (column), then they are 'true together', for precisely the same valuations. We say that such statements are semantically equivalent:

Definition 2.9. *If, for two propositions $\alpha, \beta \in \mathcal{L}_0$, it is the case that $v(\alpha) = v(\beta)$ for every possible valuation v, then we say that the propositions α and β are* semantically equivalent*.*

For example, the statements $(\neg\alpha) \Rightarrow \beta$ and $(\neg\beta) \Rightarrow \alpha$ are semantically equivalent ($\alpha \vee \beta$ 'means the same as' $\beta \vee \alpha$), as we verify below.

Let us see some examples of truth tables:

- $\alpha \Rightarrow \beta$

α	β	$\alpha \Rightarrow \beta$
0	0	1
1	0	0
0	1	1
1	1	1

· $\alpha \Rightarrow \alpha$

α	α	$\alpha \Rightarrow \alpha$
0	0	1
1	1	1

Hence, $\alpha \Rightarrow \alpha$ is a tautology (its truth table column contains only 'ones').

· $\neg\neg\alpha$

α	$\neg\alpha$	$\neg\neg\alpha$
0	1	0
1	0	1

Hence, $\neg\neg\alpha$ has the same 'column' as α, i.e. $\neg\neg\alpha$ and α are semantically equivalent.

· $(\neg\alpha) \Rightarrow \beta$

α	β	$\neg\alpha$	$(\neg\alpha) \Rightarrow \beta$
0	0	1	0
1	0	0	1
0	1	1	1
1	1	0	1

· $(\neg\beta) \Rightarrow \alpha$

α	β	$\neg\beta$	$(\neg\beta) \Rightarrow \alpha$
0	0	1	0
1	0	1	1
0	1	0	1
1	1	0	1

The last two examples show that $(\neg\alpha) \Rightarrow \beta$ and $(\neg\beta) \Rightarrow \alpha$ are semantically equivalent.

As mentioned earlier, we have defined these formulae to be $\alpha \vee \beta$ and $\beta \vee \alpha$ respectively, so we have verified that $\alpha \vee \beta$ and $\beta \vee \alpha$ are semantically equivalent, as we might hope to be the case.

Similarly, it is possible to verify that $\alpha \wedge \beta$ and $\beta \wedge \alpha$ are semantically equivalent, by using truth tables to check that $\neg(\alpha \Rightarrow (\neg\beta))$ and $\neg(\beta \Rightarrow (\neg\alpha))$ are semantically equivalent.

· $(\neg\neg\alpha) \Rightarrow \alpha$

α	$\neg\alpha$	$\neg\neg\alpha$	$(\neg\neg\alpha) \Rightarrow \alpha$
0	1	0	1
1	0	1	1

Hence, $(\neg\neg\alpha) \Rightarrow \alpha$ is a tautology.

· $\alpha \Rightarrow (\beta \Rightarrow \alpha)$

α	β	$\beta \Rightarrow \alpha$	$\alpha \Rightarrow (\beta \Rightarrow \alpha)$
0	0	1	1
1	0	1	1
0	1	0	1
1	1	1	1

Hence, $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ is a tautology.

· $((\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)))$, which we (also) denote by δ below.

α	β	γ	$\beta \Rightarrow \gamma$	$\alpha \Rightarrow (\beta \Rightarrow \gamma)$	$\alpha \Rightarrow \beta$	$\alpha \Rightarrow \gamma$	$(\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)$	δ
0	0	0	1	1	1	1	1	1
1	0	0	1	1	0	0	1	1
0	1	0	0	1	1	1	1	1
1	1	0	0	0	1	0	0	1
0	0	1	1	1	1	1	1	1
1	0	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

Hence, $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$ is a tautology.

· $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$

α	β	$\alpha \Rightarrow \beta$	$\beta \Rightarrow \alpha$	$(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$
0	0	1	1	1
1	0	0	1	1
0	1	1	0	0
1	1	1	1	1

Hence, $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ is not a tautology. In particular, if a valuation v satisfies $v(\alpha) = 0$ and $v(\beta) = 1$, then $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$.

Let us see two other ways of proving that $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ is not a tautology:

Claim: The proposition $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ is not a tautology.

Proof 1: Consider a valuation such that $v(\alpha) = 0$ and $v(\beta) = 1$. Then $v(\alpha \Rightarrow \beta) = 1$ and $v(\beta \Rightarrow \alpha) = 0$, so that $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$.

Proof 2: Suppose that, for some valuation v, $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$. Then, it necessarily follows that $v(\alpha \Rightarrow \beta) = 1$ and $v(\beta \Rightarrow \alpha) = 0$.

In such a case, since $v(\beta \Rightarrow \alpha) = 0$, we may deduce that $v(\alpha) = 0$ and $v(\beta) = 1$.

We may then verify that any valuation v that satisfies $v(\alpha) = 0$ and $v(\beta) = 1$ must also satisfy $v(\alpha \Rightarrow \beta) = 1$, and, therefore, that, for such a valuation, $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$, as required.

Let us also see another way of proving that $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ is a tautology:

Claim: The proposition $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ is a tautology.

Proof: Suppose the given statement is not a tautology. Then, there exists a valuation v (on α, β) such that $v\left(\alpha \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$. In such a case, $v(\alpha) = 1$ and $v\left(\beta \Rightarrow \alpha\right) = 0$.

But, since $v\left(\beta \Rightarrow \alpha\right) = 0$, we may deduce that $v(\beta) = 1$ and $v(\alpha) = 0$.

This gives a contradiction, since we cannot have a valuation v such that $v(\alpha) = 0$ and $v(\alpha) = 1$.

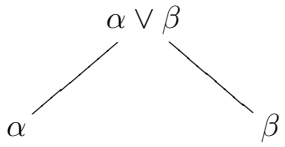
So, for every valuation v, we must have $v\left(\alpha \Rightarrow (\beta \Rightarrow \alpha)\right) = 1$, i.e. $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ is a tautology, as required.

We can make this search for 'truth' more systematic, by formalising the idea involved in the previous two proofs, where we try to find how a valuation acts on the basic building blocks of a formula by decomposing the formula into simpler and simpler components.

The *semantic tableaux method* formalises this idea, e.g. to show that a proposition α is a tautology, we may show that $v(\neg\alpha) = 0$ for all possible valuations, i.e. we may show that, for any possible attempt to describe a valuation for which α is false, there is an internal contradiction; a component of α which seems to be both true and false simultaneously.

The semantic tableaux method achieves this diagrammatically, by 'visualising the valuation'.

For example, we wish to say that $\alpha \vee \beta$ is true if α is true *or* β is true (here we use 'or' to mean 'inclusive or', or 'and/or'). Since there is an 'or' option here, there are two separate cases or 'branches' that $\alpha \vee \beta$ decomposes into. This gives rise to the following diagram:

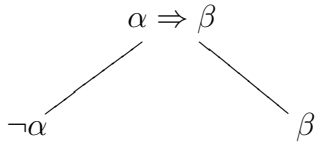


which we may read as: $\alpha \vee \beta$ is true if α is true or β is true.

Similarly, we wish to say that $\alpha \wedge \beta$ is true if α is true *and* β is true. Since there is an 'and' option here, there is only one 'branch' which $\alpha \wedge \beta$ decomposes into. This gives rise to the following diagram:

which we may read as: $\alpha \wedge \beta$ is true if α is true and β is true.

We may diagrammatically represent other formulae as well:

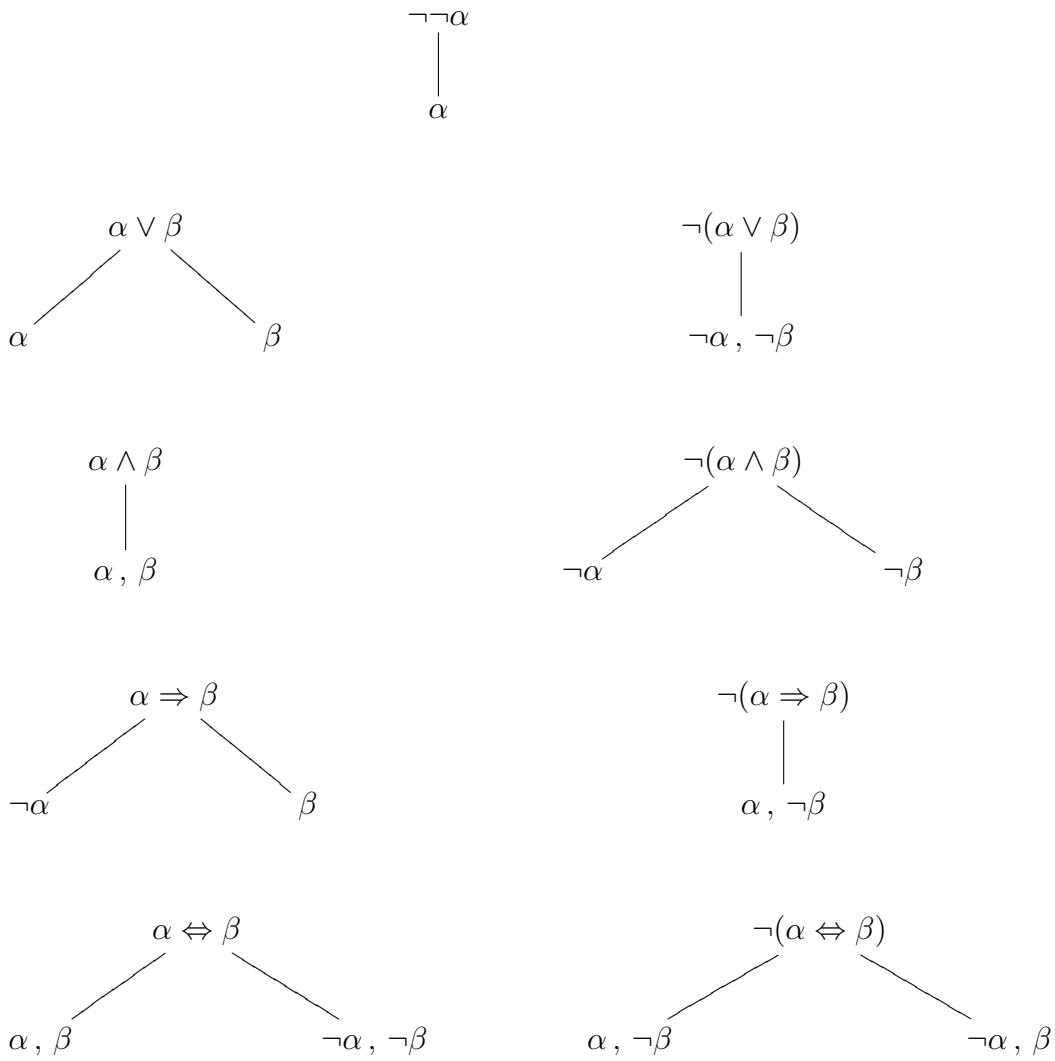


which we may read as: $\alpha \Rightarrow \beta$ is true if $\neg\alpha$ is true or if β is true (i.e. $\alpha \Rightarrow \beta$ is true whenever α is false or whenever β is true).

These diagrams are examples of propositional semantic tableaux, and we may combine such simple diagrams, repeatedly if required, so as to obtain larger, more complicated, tableaux; in this way, we may break down any proposition into its 'indecomposable' or 'primitive' parts.

Let us give a complete set of the simple diagrams that we will allow ourselves to use. You may find it instructive to read or recreate them as we did for the three examples above, so as to verify that the diagrams below 'make sense'.

The following diagrams are the *basic rules for propositional semantic tableaux*:



Using these rules, our goal is, when we are given a proposition $\alpha \in \mathcal{L}_0$ (expressed in terms of its primitive components):

- To determine whether or not α is a tautology.
- If α is not a tautology, to determine for which kind(s) of valuation(s) it fails to be true.

Given a proposition α, the strategy we adopt is:

1) Consider $\neg\alpha$, the negation of the given proposition.

2) Using the above rules, break down $\neg\alpha$ into smaller and smaller parts, to obtain a propositional semantic tableau for $\neg\alpha$.

3) Consider the resulting branches of the tableau obtained (a branch here means a 'whole branch', starting from one of the lower edges of the tableau and following a path up to the top):

   · If a branch contains one of the 'indecomposable', or 'primitive', parts of α as well as its negation, then we refer to that branch as *closed*. There is no valuation for which such a branch can be realised. Such a branch is thus 'self-contradictory'.

   · If a branch does not contain a 'primitive' part and its negation, then it is a branch that can be realised. We refer to such a branch as *open*.

4) If every branch of the resulting tableau, for $\neg\alpha$, is closed, then the original proposition, α, is a tautology. This is because if every branch of the tableau for $\neg\alpha$ is closed, then there is no valuation for which $\neg\alpha$ is true i.e. there is no valuation for which α is false.

   If the tableau for $\neg\alpha$ contains an open branch, then this branch corresponds to a valuation v for which $v(\neg\alpha) = 1$ i.e. for which $v(\alpha) = 0$. Therefore, α is not a tautology. We may describe the valuation corresponding to such an open branch by observing which ofδ, $\neg\delta$ shows up along the branch, for each 'primitive' part δ of the (original) proposition α.

The above strategy will hopefully be easier to understand and implement once we have seen some examples. Before we do so, we note that:

· When developing or constructing a tableau, we may break down, at each step, any of the propositions present in the tableau (using the rules given above). If we do so, we must write down the decomposition of such a proposition on each branch which leads up to it.

· It is often convenient to first break down propositions which will lead to a single branch, rather than two branches (e.g. if we have propositions of the form $\kappa \Rightarrow \lambda$ and $\neg(\mu \Rightarrow \nu)$ present in the same tableau, it is often useful to apply the rule for $\neg(\mu \Rightarrow \nu)$ before applying the rule for $\kappa \Rightarrow \lambda$.)

Let us now use the semantic tableaux method in order to determine whether or not some given propositions are tautologies.

Example 2.10. *Consider* $\alpha \Rightarrow \alpha$*:*

We form a semantic tableau starting from the negation of this proposition, $\neg(\alpha \Rightarrow \alpha)$.

$$\begin{array}{c} \neg(\alpha \Rightarrow \alpha) \\ | \\ \alpha\,,\ \neg\alpha \end{array}$$

closed

There is only one branch, which contains both α *and* $\neg\alpha$*, so every branch is closed.*

Hence, $\alpha \Rightarrow \alpha$ *is a tautology.*

Example 2.11. *Consider* $\alpha \Rightarrow \beta$*:*

We form a semantic tableau starting from the negation of this proposition, $\neg(\alpha \Rightarrow \beta)$.

$$\begin{array}{c} \neg(\alpha \Rightarrow \beta) \\ | \\ \alpha\,,\ \neg\beta \end{array}$$

open

Hence, $\alpha \Rightarrow \beta$ *is not a tautology. In fact, since the only branch present contains* α *and* $\neg\beta$*, we deduce that* $v(\alpha \Rightarrow \beta) = 0$ *for any valuation* v *for which* $v(\alpha) = 1$ *and* $v(\neg\beta) = 1$*, i.e.* $v(\alpha \Rightarrow \beta) = 0$ *for any valuation* v *for which* $v(\alpha) = 1$ *and* $v(\beta) = 0$ *(as we already know, by the definition of a valuation).*

Example 2.12. *Consider* $\alpha \Rightarrow (\beta \Rightarrow \alpha)$*:*

Let us create a semantic tableau for this, step by step. First, we may decompose $\neg (\alpha \Rightarrow (\beta \Rightarrow \alpha))$*, using the relevant rule:*

$$\begin{array}{c} \neg (\alpha \Rightarrow (\beta \Rightarrow \alpha)) \\ \vert \\ \alpha \, , \, \neg(\beta \Rightarrow \alpha) \end{array}$$

We may now decompose $\neg(\beta \Rightarrow \alpha)$*:*

$$\begin{array}{c} \neg (\alpha \Rightarrow (\beta \Rightarrow \alpha)) \\ \vert \\ \alpha \, , \, \neg(\beta \Rightarrow \alpha) \\ \vert \\ \beta \, , \, \neg\alpha \end{array}$$

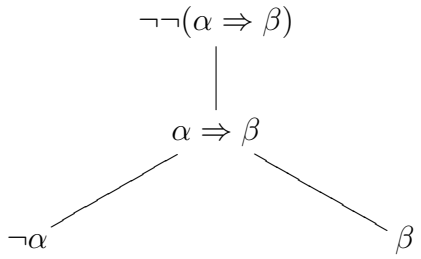
closed

There is only one branch overall, which is closed (since this branch contains both α *and* $\neg\alpha$*).*

Hence, $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ *is a tautology.*

Example 2.13. *Consider* $\neg(\alpha \Rightarrow \beta)$*:*

We form a semantic tableau starting from the negation of this proposition, $\neg\neg\,(\alpha \Rightarrow \beta)$.



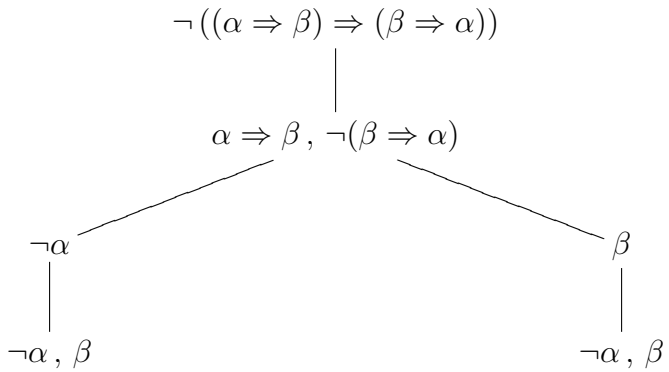
open open

Both branches are open, so $\neg(\alpha \Rightarrow \beta)$ *is not a tautology.*

In fact, since one of the branches contains a single 'indecomposable' $\neg\alpha$ *and the other branch contains a single 'indecomposable'* β*, we deduce that* $v\,(\neg(\alpha \Rightarrow \beta)) = 0$ *for any valuation* v *for which* $v(\neg\alpha) = 1$ *(i.e. for which* $v(\alpha) = 0$*), or for any valuation* v *for which* $v(\beta) = 1$.

Example 2.14. *Consider* $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$*:*

We form a semantic tableau starting from $\neg\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right)$.

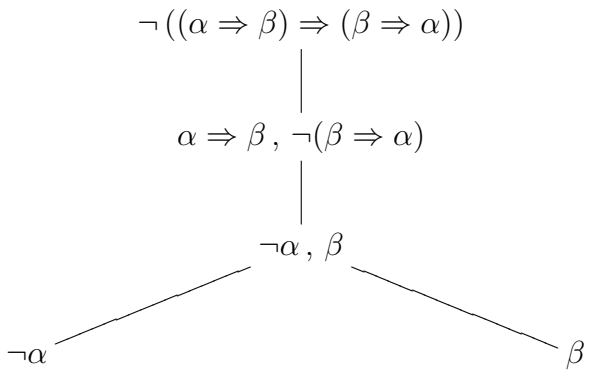


Both branches are open, and reveal that, if a valuation v *satisfies* $v(\alpha) = 0$ *and* $v(\beta) = 1$*, then* $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$.

So, the proposition $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ *is not a tautology.*

Above, we decomposed $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$*, followed by* $\alpha \Rightarrow \beta$*, and, finally,* $\neg(\beta \Rightarrow \alpha)$*. Observe that, at the last step, we used the rule for* $\neg(\beta \Rightarrow \alpha)$ *on both branches which were 'connected to'* $\neg(\beta \Rightarrow \alpha)$.

We could have decomposed $\alpha \Rightarrow \beta$ *and* $\neg(\beta \Rightarrow \alpha)$ *the 'other way around', to obtain the following valid semantic tableau:*



open open

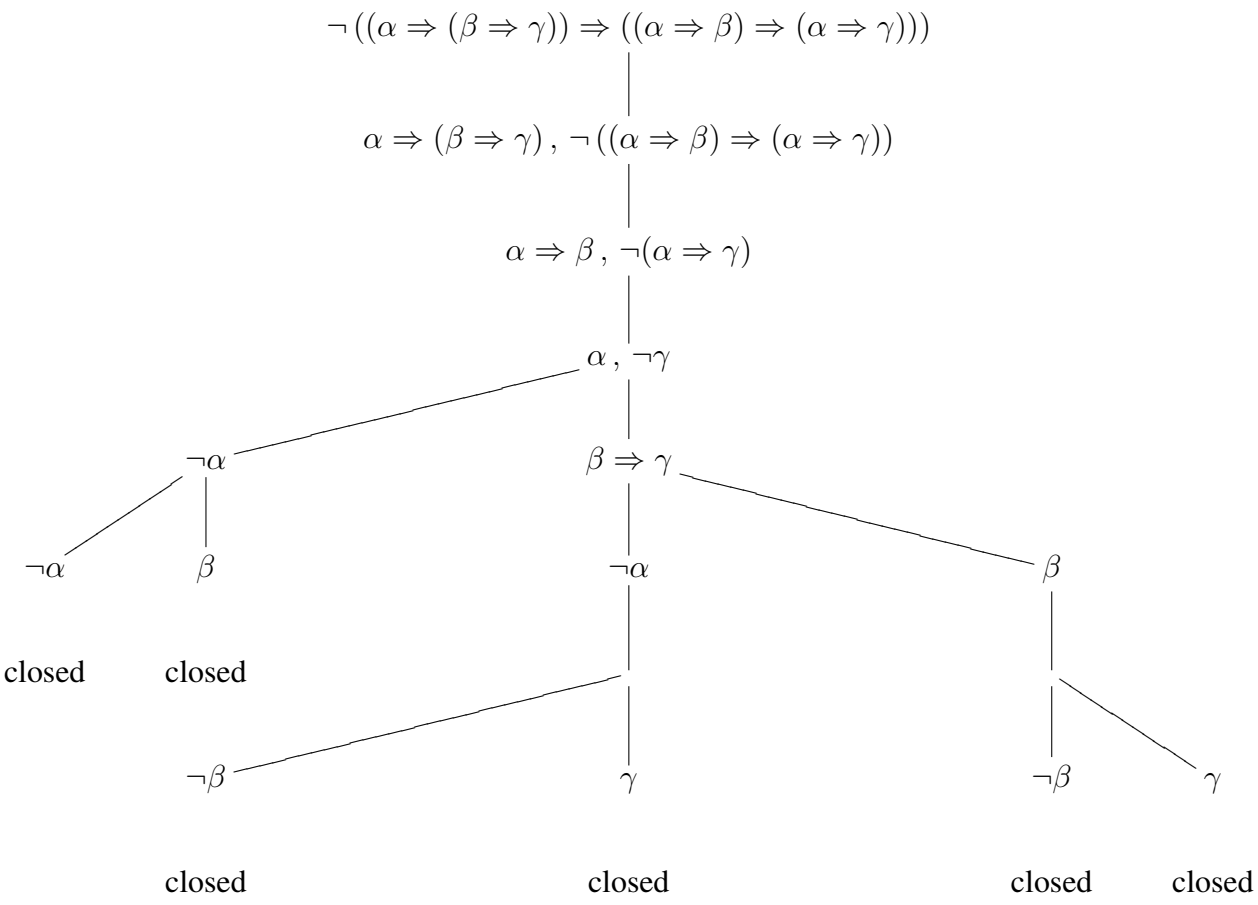
Once again, both branches are open, and we have shown that the proposition $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ *is not a tautology.*

In particular, if a valuation v *satisfies* $v(\alpha) = 0$ *and* $v(\beta) = 1$*, then* $v\left((\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)\right) = 0$.

This allows us to draw the same conclusion as earlier; the proposition $(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$ *is not a tautology.*

Example 2.15. *Consider* $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$*:*

We form a semantic tableau starting from $\neg\,((\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)))$.

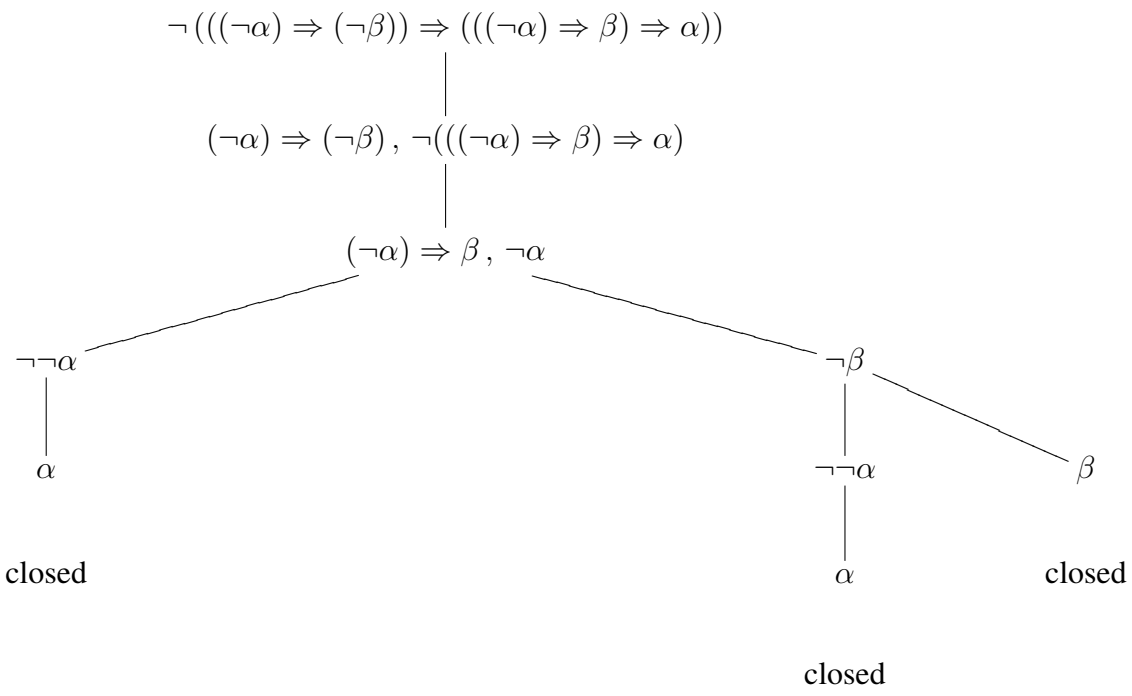


All branches are closed.

Hence, the proposition $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$ *is a tautology.*

Example 2.16. *Consider* $((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$*:*

We form a semantic tableau starting from $\neg\,(((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha))$.

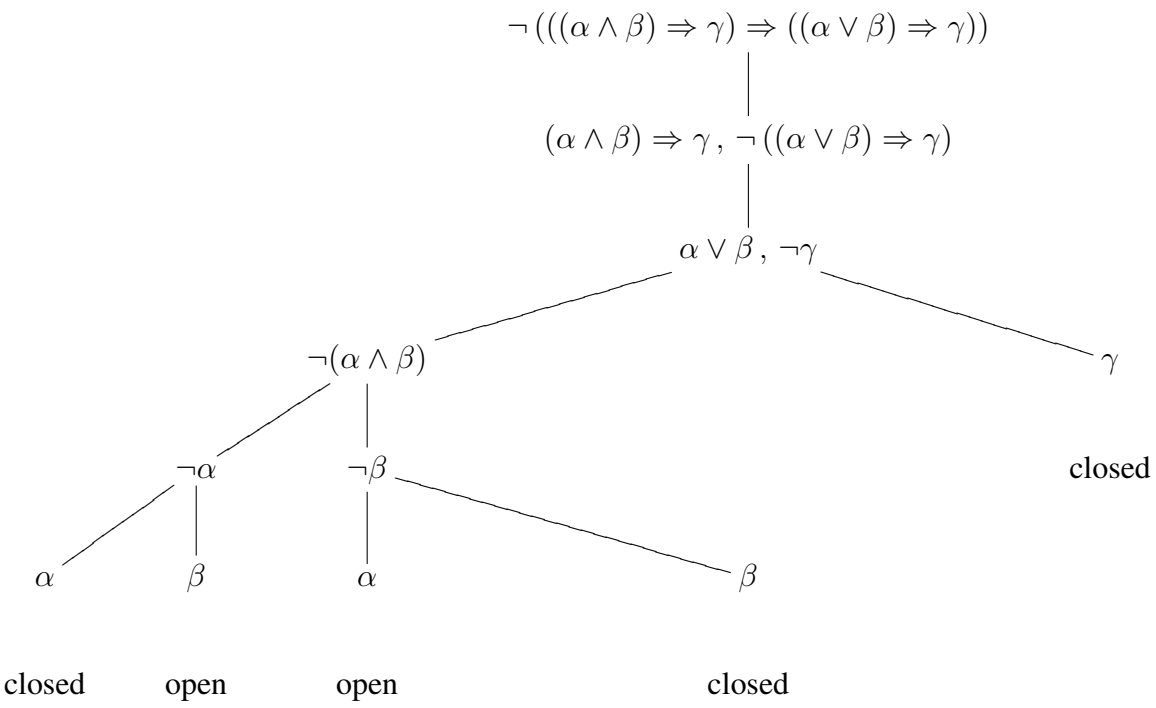


All branches are closed.

Hence, the proposition $((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$ *is a tautology.*

Example 2.17. *Consider* $((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma)$*:*

We form a semantic tableau starting from $\neg\,(((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma))$.



This is an example where there are both closed and open branches. Since there exist open branches, we deduce that $((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma)$ *is not a tautology.*

By examining the open branches, we see that there are two types of valuations for which the original proposition fails to be true; for a valuation v*:*

- *If* $v(\alpha) = 0$, $v(\beta) = 1$, $v(\gamma) = 0$, *then* $v\,(((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma)) = 0$.
- *If* $v(\alpha) = 1$, $v(\beta) = 0$, $v(\gamma) = 0$, *then* $v\,(((\alpha \wedge \beta) \Rightarrow \gamma) \Rightarrow ((\alpha \vee \beta) \Rightarrow \gamma)) = 0$.

The next idea will allow us to use valuations and, as a result, 'truth', to connect various statements. We would like to have a way of expressing the idea that one proposition may be true when another proposition is true, or when everything in a set of other propositions is true. For example, when β is true, then $\alpha \Rightarrow \beta$ is also true (whether or not α is true or false).

Definition 2.18. *Consider a subset S of the set $\mathcal{L}_0$ of all propositions, and a proposition $\alpha \in \mathcal{L}_0$. Then, we say that S* semantically implies *or* entails *α, and write*

$$S \models \alpha$$

if, for every valuation v such that $v(s) = 1$ for all $s \in S$, we also have $v(\alpha) = 1$.

Note 2.19. *Using Definition 2.7:*

- *If, for $\alpha \in \mathcal{L}_0$ and a valuation $v : \mathcal{L}_0 \to \{0, 1\}$, we have $v(\alpha) = 1$, then we say that v is a* model *of α.*
- *If, for $S \subset \mathcal{L}_0$ and a valuation $v : \mathcal{L}_0 \to \{0, 1\}$, we have $v(\alpha) = 1$ for all $\alpha \in S$, then we say that v is a* model *of S.*

So, another formulation of Definition 2.18 is: $S \models \alpha$ if every model of S is a model of α.

For example, we know, from the definition of a valuation, that $\{\neg\alpha\} \models (\alpha \Rightarrow \beta)$ and, also, that $\{\beta\} \models (\alpha \Rightarrow \beta)$, i.e. $\alpha \Rightarrow \beta$ is true 'whenever' α is false or β is true.

Furthermore, note that, if the empty set of propositions entails a proposition:

$$\emptyset \models \alpha$$

then α is always true, i.e. it is true for every possible valuation. In other words, α is a tautology, so our use of '$\models$' here agrees with the notation used for a tautology earlier: we may simply write '$\models \alpha$' instead of '$\emptyset \models \alpha$', if α is a tautology.

As an example of an entailment, let us show that if $S = \{\alpha\,,\, \alpha \Rightarrow \beta\}$, then $S \models \beta$:

Claim: $\{\alpha\,,\, \alpha \Rightarrow \beta\} \models \beta$

Proof: Suppose that α and $\alpha \Rightarrow \beta$ are true, but β is not, i.e. that for some valuation v: $v(\alpha) = 1$ and $v(\alpha \Rightarrow \beta) = 1$, but $v(\beta) = 0$. But, since $v(\alpha) = 1$ and $v(\beta) = 0$, we deduce that $v(\alpha \Rightarrow \beta) = 0$, contradicting our assumption that $v(\alpha \Rightarrow \beta) = 1$. So, for any valuation v satisfying $v(\alpha) = 1$ and $v(\alpha \Rightarrow \beta) = 1$, we must have $v(\beta) = 1$.

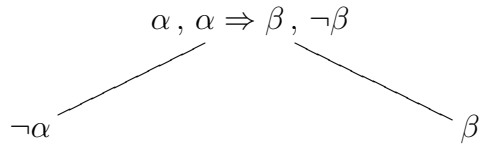
Given a set S of propositions and a proposition α, we may try to use truth tables to determine whether or not $S \models \alpha$, or find a proof as above. However, just as when investigating whether or not given propositions are tautologies, the semantic tableaux method often provides a more efficient way of studying such a problem.

If we wish to determine whether or not $S \models \alpha$, then we may simply verify that there is no valuation for which every proposition in S is true and for which α is false, as we did in the proof above. This is the same as checking that the propositions in the set $S \cup \neg\alpha$ cannot all be true simultaneously, for a given valuation, i.e. that the semantic tableau starting with the propositions in $S \cup \neg\alpha$ contains only closed branches.

For example, let us give another demonstration of $\{\alpha\,,\,\alpha \Rightarrow \beta\} \models \beta$:

Example 2.20. *Consider* $\{\alpha\,,\,\alpha \Rightarrow \beta\} \models \beta$*:*

We form a semantic tableau starting from the propositions α, $\alpha \Rightarrow \beta$, $\neg\beta$.



closed closed

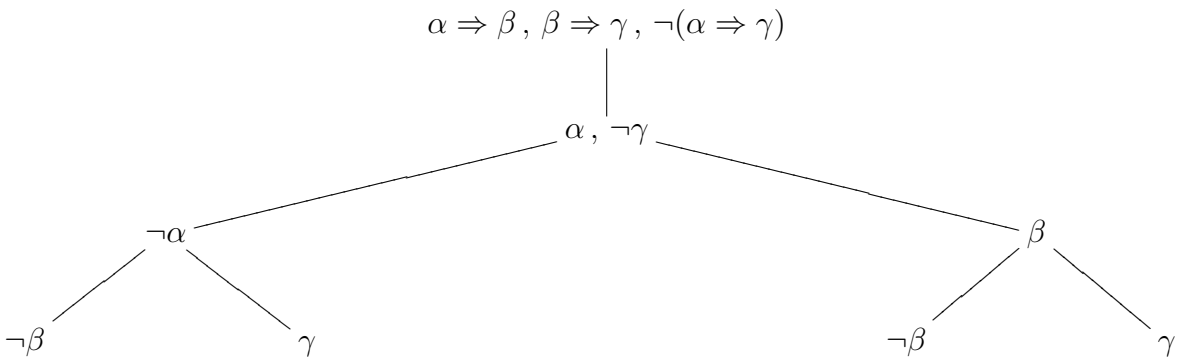
Since every branch is closed, we deduce that there is no valuation for which $\alpha \Rightarrow \beta$ *and* α *are true and* β *is false.*

In other words, the entailment $\{\alpha\,,\,\alpha \Rightarrow \beta\} \models \beta$ *holds.*

As another example, let us show that $\{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma\} \models (\alpha \Rightarrow \gamma)$:

Example 2.21. *Consider* $\{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma\} \models (\alpha \Rightarrow \gamma)$*:*

We form a semantic tableau starting from the propositions $\alpha \Rightarrow \beta$, $\beta \Rightarrow \gamma$, $\neg(\alpha \Rightarrow \gamma)$.



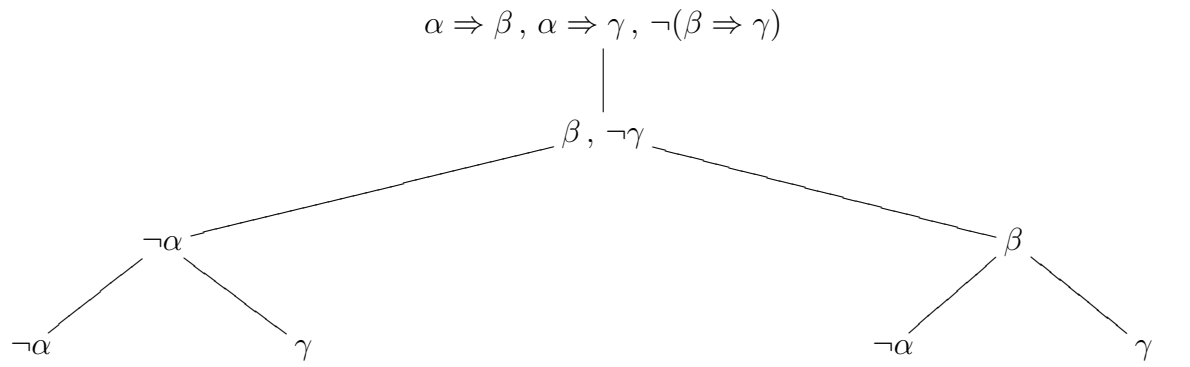
Since every branch is closed, we deduce that there is no valuation for which $\alpha \Rightarrow \beta$ *and* $\beta \Rightarrow \gamma$ *are true and* $\alpha \Rightarrow \gamma$ *is false.*

In other words, the entailment $\{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma\} \models (\alpha \Rightarrow \gamma)$ *holds.*

Finally let us determine whether or not $\{\alpha \Rightarrow \beta, \alpha \Rightarrow \gamma\}$ entails $\beta \Rightarrow \gamma$:

Example 2.22. *Consider whether or not* $\{\alpha \Rightarrow \beta, \alpha \Rightarrow \gamma\}$ *entails* $\beta \Rightarrow \gamma$*:*

We form a semantic tableau starting from the propositions $\alpha \Rightarrow \beta$*,* $\alpha \Rightarrow \gamma$*,* $\neg(\beta \Rightarrow \gamma)$*.*



open closed open closed

There are two open branches above, so we deduce that $\{\alpha \Rightarrow \beta, \alpha \Rightarrow \gamma\}$ *does not entail* $\beta \Rightarrow \gamma$*.*

By considering the two open branches, we see that they both describe the same type of valuation v*:*

- *If* $v(\alpha) = 0$*,* $v(\beta) = 1$*,* $v(\gamma) = 0$*, then* $v(\alpha \Rightarrow \beta) = 1$*,* $v(\alpha \Rightarrow \gamma) = 1$ *and* $v(\beta \Rightarrow \gamma) = 0$*.*

Under such a valuation, $\alpha \Rightarrow \beta$ *and* $\alpha \Rightarrow \gamma$ *are both true, but* $\beta \Rightarrow \gamma$ *is not; we may deduce that* $\{\alpha \Rightarrow \beta, \alpha \Rightarrow \gamma\}$ *does not entail* $\beta \Rightarrow \gamma$*.*

2.2 Syntactic aspects of propositional logic

In this section, we shall offer another viewpoint on the kinds of implications we wish to study. We have been referring to valuations, which are functions defined so that they allow us to have a well defined and 'intuitive' concept of truth in our setting. This was the *semantic* framework of propositional logic. We now wish to develop a method of *proof*. We will develop the *syntactic* framework of propositional logic.

For example, we would like to find a way to try to prove a proposition when given, as assumptions, some (other) propositions (or none at all). The semantic tableaux method seems to perform a similar task, since it provides an algorithm that allows us to break down any proposition or set of propositions.

However, we shall adopt a more 'minimalist' and 'linear' view of what a proof is. We will define a proof as a sequence of lines, such that each line is either something that we are assuming or something that can be deduced from an earlier part of the proof. Then, if we can write down a sequence of such lines ending in a proposition α, we shall say that we have proven α.

Let us now define a version of proof. We will use *axioms* and one or more *rules of deduction*.

We start by giving a list of axioms i.e. of propositions that we can always include in proofs.

As *axioms* for proofs in propositional logic, let us take the following:

Axiom 1 $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ for all $\alpha, \beta \in \mathcal{L}_0$

Axiom 2 $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$ for all $\alpha, \beta, \gamma \in \mathcal{L}_0$

Axiom 3 $((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$ for all $\alpha, \beta \in \mathcal{L}_0$

These are the basic building blocks we will use for proofs in propositional logic.

Note 2.23. *Each of the above axioms holds for all instances of propositions α, β, γ, where relevant. As such, each of these axioms is really a collection of infinitely many axioms. For this reason, you may also find each of the above axioms referred to as an* axiom scheme *in some sources.*

Note 2.24. *The above axioms are all instances of tautologies, as was verified, using the semantic tableaux method, earlier.*

In addition, we shall use the rule that says that "if we have α and we also have $\alpha \Rightarrow \beta$, then we have β", as a *rule of deduction*. This particular rule is known as *modus ponens*.

Modus ponens: From α and $\alpha \Rightarrow \beta$, we may deduce β (for all propositions α, β)

We will assume only the above axioms and will use only modus ponens as a rule of deduction, when constructing proofs (in propositional logic):

Definition 2.25. *Let S be a subset of $\mathcal{L}_0$, i.e. $S \subset \mathcal{L}_0$, and let α be a proposition, i.e. $\alpha \in \mathcal{L}_0$. Then, a* proof *of α from S is a finite, ordered sequence of propositions (or lines), t_1, t_2, $\cdots$, t_n say, such that t_n is the proposition α, and, such that, for each proposition t_i, $1 \leq i \leq n$:*

a) t_i is an (occurrence of an) axiom, or

b) t_i is a proposition in S (we call such a proposition a hypothesis*), or*

c) t_i is a proposition that can be deduced by modus ponens from two preceding propositions: t_i is some proposition β, and, for some $j, k < i$, t_j is some proposition α and t_k is the proposition $\alpha \Rightarrow \beta$.

Let us now give the notion corresponding to entailment in our current, syntactic, setting:

Definition 2.26. *Consider a subset S of the set of all propositions $\mathcal{L}_0$, and a proposition $\alpha \in \mathcal{L}_0$. Then, we say that S* syntactically implies *or* proves *α, and write*

$$S \vdash \alpha$$

if there exists a proof of α from S (S being a set of hypotheses in this case).

Just as a tautology is a proposition which is 'always true', without assuming the truth of any other propositions, a *theorem* is a proposition which we can prove without involving any hypotheses, i.e. a proposition which we can prove by taking $S = \emptyset$ above:

Definition 2.27. *If there exists a proof of a proposition $\alpha \in \mathcal{L}_0$ that uses no hypotheses, but perhaps only occurrences of the three given axioms and modus ponens, then we say that α is a* theorem*, i.e. $\alpha \in \mathcal{L}_0$ is a theorem if $\emptyset \vdash \alpha$.*

In such a case, we may also write $\vdash \alpha$.

Let us give some examples of proofs in propositional logic (at each line of a proof, we write down whether the given proposition is an axiom, or a hypothesis, or is obtained through a valid use of modus ponens).

Let us first verify that the following syntactic implication holds, for any α, β in $\mathcal{L}_0$:

$$\{\alpha\} \vdash (\beta \Rightarrow \alpha)$$

The following is a proof of $\{\alpha\} \vdash (\beta \Rightarrow \alpha)$:

1.	α	Hypothesis
2.	$\alpha \Rightarrow (\beta \Rightarrow \alpha)$	Axiom 1
3.	$\beta \Rightarrow \alpha$	Modus ponens on lines 1, 2

Next, let us verify that the following syntactic implication holds, for any α, β in $\mathcal{L}_0$:

$$\{(\neg\alpha) \Rightarrow (\neg\beta)\,,\, (\neg\alpha) \Rightarrow \beta\} \vdash \alpha$$

The following is a proof of $\{(\neg\alpha) \Rightarrow (\neg\beta)\,,\, (\neg\alpha) \Rightarrow \beta\} \vdash \alpha$:

1.	$(\neg\alpha) \Rightarrow (\neg\beta)$	Hypothesis
2.	$(\neg\alpha) \Rightarrow \beta$	Hypothesis
3.	$((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$	Axiom 3
4.	$((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha$	Modus ponens on lines 1, 3
5.	α	Modus ponens on lines 2, 4

We have chosen quite a limited set of axioms, but our machinery allows us to prove other 'obvious' propositions that we have not assumed above. For example, let us show that we can prove $\alpha \Rightarrow \alpha$ (for any proposition α), without using any hypotheses:

Proposition 2.28. *For any α in $\mathcal{L}_0$, the following is a theorem:*

$$\vdash (\alpha \Rightarrow \alpha)$$

Proof. The following is a proof of $\alpha \Rightarrow \alpha$, which uses no hypotheses:

1.	$(\alpha \Rightarrow ((\alpha \Rightarrow \alpha) \Rightarrow \alpha)) \Rightarrow ((\alpha \Rightarrow (\alpha \Rightarrow \alpha)) \Rightarrow (\alpha \Rightarrow \alpha))$	Axiom 2
2.	$\alpha \Rightarrow ((\alpha \Rightarrow \alpha) \Rightarrow \alpha)$	Axiom 1
3.	$(\alpha \Rightarrow (\alpha \Rightarrow \alpha)) \Rightarrow (\alpha \Rightarrow \alpha)$	Modus ponens on lines 1, 2
4.	$\alpha \Rightarrow (\alpha \Rightarrow \alpha)$	Axiom 1
5.	$\alpha \Rightarrow \alpha$	Modus ponens on lines 3, 4

□

Earlier, we showed that $\{\alpha \Rightarrow \beta\,,\, \beta \Rightarrow \gamma\} \models (\alpha \Rightarrow \gamma)$ (see Example 2.21).

Let us now give the related proof:

Proposition 2.29. *For any α in $\mathcal{L}_0$, the following syntactic implication holds:*

$$\{\alpha \Rightarrow \beta\,,\ \beta \Rightarrow \gamma\} \vdash (\alpha \Rightarrow \gamma)$$

Proof. The following is a proof of $\alpha \Rightarrow \gamma$ from $\{\alpha \Rightarrow \beta\,,\ \beta \Rightarrow \gamma\}$:

1.	$\alpha \Rightarrow \beta$	Hypothesis
2.	$\beta \Rightarrow \gamma$	Hypothesis
3.	$(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$	Axiom 2
4.	$(\beta \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow (\beta \Rightarrow \gamma))$	Axiom 1
5.	$\alpha \Rightarrow (\beta \Rightarrow \gamma)$	Modus ponens on lines 2, 4
6.	$(\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)$	Modus ponens on lines 3, 5
7.	$\alpha \Rightarrow \gamma$	Modus ponens on lines 1, 6

□

As may be seen from the examples given so far, the construction of proofs may appear quite convoluted, even in cases involving seemingly relatively simple propositions, such as the ones given above.

It is often helpful to use the following result (where applicable) when 'proving' in this setting:

Theorem 2.30 (Deduction Theorem for propositional logic)**.** *Let $S \subset \mathcal{L}_0$ and $\alpha, \beta \in \mathcal{L}_0$. Then*

$$S \vdash (\alpha \Rightarrow \beta) \qquad \textit{if and only if} \qquad S \cup \{\alpha\} \vdash \beta$$

Proof. We will prove the two 'directions' of this theorem separately.

First, let us suppose that $S \vdash (\alpha \Rightarrow \beta)$, i.e. that there exists a sequence of propositions $t_1, \cdots, t_n$, such that each t_i $(1 \leq i \leq n)$ is an instance of an axiom, or is an element of S, or is obtained by modus ponens on two previous lines, and such that the final proposition, t_n, is $\alpha \Rightarrow \beta$.

Then, if we simply add $t_{n+1} : \alpha$, which is a hypothesis in $S \cup \{\alpha\}$, followed by $t_{n+2} : \beta$, which is obtained by modus ponens on the propositions t_n, t_{n+1}, we obtain a proof of β from $S \cup \{\alpha\}$, i.e. $S \cup \{\alpha\} \vdash \beta$, as required.

Now, suppose there exists a proof of β from $S \cup \{\alpha\}$, say a sequence of propositions $t_1, \cdots, t_m$, with $t_m : \beta$. We will modify this proof to obtain a proof of $\alpha \Rightarrow \beta$ from S, by trying to include in our proof the proposition $\alpha \Rightarrow t_i$, for each proposition t_i above.

For each t_i, $1 \leq i \leq m$, we have the following cases:

A) t_i is an (occurrence of an) axiom.

Then, since t_i is an axiom, we may still include t_i in our new proof. We may then use the following three propositions in order to obtain $\alpha \Rightarrow t_i$:

1.	t_i	Axiom
2.	$t_i \Rightarrow (\alpha \Rightarrow t_i)$	Axiom 1
3.	$\alpha \Rightarrow t_i$	Modus ponens on lines 1, 2

B) t_i is a proposition in $S \cup \{\alpha\}$, i.e. $t_i \in S$ or t_i is α.

If $t_i \in S$, then it remains a hypothesis (since we are trying to construct a proof of $\alpha \Rightarrow \beta$ from S), so we may still include t_i in our new proof. Hence, we may proceed similarly to case A in order to obtain $\alpha \Rightarrow t_i$:

1.	t_i	Hypothesis
2.	$t_i \Rightarrow (\alpha \Rightarrow t_i)$	Axiom 1
3.	$\alpha \Rightarrow t_i$	Modus ponens on lines 1, 2

If t_i is α, we cannot use the same sequence of lines, because α is no longer a hypothesis, in the setting of $S \vdash (\alpha \Rightarrow \beta)$. However, there is another way of obtaining $\alpha \Rightarrow t_i$, i.e. $\alpha \Rightarrow \alpha$, without needing to include the proposition α (by itself): we may simply write down a proof of the theorem $\alpha \Rightarrow \alpha$, which was proven earlier (see Proposition 2.28).

C) Finally, suppose that t_i is a proposition that can be deduced by modus ponens from two preceding propositions.

So, suppose that, for some $j < i$, $k < i$ and $\gamma, \delta \in \mathcal{L}_0$, the proposition t_j is γ, the proposition t_k is $\gamma \Rightarrow \delta$, and t_i is the proposition δ, obtained by modus ponens on lines t_j, t_k.

Then, since t_i follows t_j and t_k, we may, inductively, assume that we have already successfully replaced the lines t_j and t_k by $\alpha \Rightarrow t_j$ and $\alpha \Rightarrow t_k$ respectively.

So, we may now use:

- $\alpha \Rightarrow t_j$: $\alpha \Rightarrow \gamma$
- $\alpha \Rightarrow t_k$: $\alpha \Rightarrow (\gamma \Rightarrow \delta)$

Then, we can obtain the line $\alpha \Rightarrow t_i$: $\alpha \Rightarrow \delta$, by including the following sequence of lines:

1.	$\alpha \Rightarrow \gamma$	
2.	$\alpha \Rightarrow (\gamma \Rightarrow \delta)$	
3.	$(\alpha \Rightarrow (\gamma \Rightarrow \delta)) \Rightarrow ((\alpha \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \delta))$	Axiom 2
4.	$(\alpha \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \delta)$	Modus ponens on lines 2, 3
5.	$\alpha \Rightarrow \delta$	Modus ponens on lines 1, 4

Hence we have shown that, given a proof of β from $S \cup \{\alpha\}$, it is possible to 'replace' each proposition t_i in this proof by the proposition $\alpha \Rightarrow t_i$, without using α as a hypothesis. Therefore, it is possible to construct a proof of $\alpha \Rightarrow \beta$ from S, as required: the final proposition $t_m : \beta$, of the proof $S \cup \{\alpha\} \vdash \beta$ may be replaced by the proposition $\alpha \Rightarrow t_m : \alpha \Rightarrow \beta$, to give a proof of $S \vdash (\alpha \Rightarrow \beta)$.

□

Let us see how the Deduction Theorem may be used in order to help provide a 'quicker' demonstration of $\{\alpha \Rightarrow \beta\,,\, \beta \Rightarrow \gamma\} \vdash (\alpha \Rightarrow \gamma)$.

By the Deduction Theorem, $\{\alpha \Rightarrow \beta\,,\, \beta \Rightarrow \gamma\} \vdash (\alpha \Rightarrow \gamma)$ if and only if

$$\{\alpha \Rightarrow \beta\,,\, \beta \Rightarrow \gamma\,,\, \alpha\} \vdash \gamma$$

So, it suffices to give a proof of the latter statement; we do so below:

1.	α	Hypothesis
2.	$\alpha \Rightarrow \beta$	Hypothesis
3.	$\beta \Rightarrow \gamma$	Hypothesis
4.	β	Modus ponens on lines 1, 2
5.	γ	Modus ponens on lines 3, 4

Let us now give another result, which is often useful, in the setting of proofs:

Lemma 2.31. *Let $S \subset \mathcal{L}_0$ and $\alpha \in \mathcal{L}_0$, and suppose that $S \vdash \alpha$. Then, for any $\beta \in \mathcal{L}_0$:*

$$S \vdash \beta \qquad \textit{if and only if} \qquad S \cup \{\alpha\} \vdash \beta$$

Proof. Suppose that $S \vdash \beta$. Then, any proof of β from S also 'works' as a proof of β from $S \cup \{\alpha\}$, so that $S \cup \{\alpha\} \vdash \beta$.

Now, suppose that $S \cup \{\alpha\} \vdash \beta$. Since we assume that $S \vdash \alpha$ holds, we may consider a proof of β from $S \cup \{\alpha\}$, and replace any occurrence of α, as a 'line' in this proof, by a proof of α from S. This results in a proof of β from S only (a proof that does not use α as a hypothesis), i.e. it shows that $S \vdash \beta$.

So, as required, we have shown that, if $S \vdash \alpha$, then, for any $\beta \in \mathcal{L}_0$, $S \vdash \beta$ if and only if $S \cup \{\alpha\} \vdash \beta$. □

2.3 Completeness theorem for propositional logic

We begin this section by noting that a result similar to the Deduction Theorem (Theorem 2.30) also holds for semantic implication:

Lemma 2.32. *Let $S \subset \mathcal{L}_0$ and $\alpha, \beta \in \mathcal{L}_0$. Then:*

$$S \models (\alpha \Rightarrow \beta) \qquad \textit{if and only if} \qquad S \cup \{\alpha\} \models \beta$$

Proof. Suppose that $S \models (\alpha \Rightarrow \beta)$. This means that we may assume that for every valuation v such that $v(s) = 1$ for all $s \in S$, we must have $v(\alpha \Rightarrow \beta) = 1$.

If we therefore restrict attention to those valuations v satisfying $v(s) = 1$ for all $s \in S$ and $v(\alpha) = 1$, we necessarily have $v(\alpha \Rightarrow \beta) = 1$ and $v(\alpha) = 1$, so that we must have $v(\beta) = 1$ (by the definition of a valuation).

So, for every valuation for which all propositions in S are true and α is true, β is also true, i.e. $S \cup \{\alpha\} \models \beta$.

Similarly, suppose that $S \cup \{\alpha\} \models \beta$. Then, for every valuation v satisfying $v(s) = 1$ for all s in S, and $v(\alpha) = 1$, it follows that $v(\beta) = 1$.

Let us choose a valuation v such that $v(s) = 1$ for all $s \in S$. Then, either $v(\alpha) = 1$, in which case $v(\beta) = 1$, by assumption, and so $v(\alpha \Rightarrow \beta) = 1$, or $v(\alpha) = 0$, in which case $v(\alpha \Rightarrow \beta) = 1$, irrespective of the value of $v(\beta)$, by definition of a valuation.

In either case, $v(\alpha \Rightarrow \beta) = 1$, so we have shown that $S \models (\alpha \Rightarrow \beta)$.

Therefore, as required, we have shown that $S \models (\alpha \Rightarrow \beta)$ if and only if $S \cup \{\alpha\} \models \beta$. □

This result, which mirrors the Deduction Theorem, is indicative of a deep and complete underlying connection between syntactic and semantic implication, between proof and entailment. Our main aim in this section (that is, in the remainder of this chapter) is to show that these two settings completely coincide. In other words, we aim to show that, even though semantic and syntactic implications are defined in different ways, for any $S \subset \mathcal{L}_0$ and any $\alpha \in \mathcal{L}_0$:

$$S \models \alpha \qquad \text{if and only if} \qquad S \vdash \alpha$$

This result is the *Completeness Theorem for propositional logic*.

Before we proceed to prove (the two directions of) the Completeness Theorem, let us give instances of a few more theorems, whose proofs will offer us the opportunity to practise the use of the Deduction Theorem in some more cases.

In the proofs that follow, we will allow ourselves to write down 'theorem' as a justification of a proposition that has already been proven to be a theorem. This is not 'technically' allowed, but it offers a helpful shorthand. In order to obtain a complete, rigorous proof of any of the results below, we would simply need to replace each instance of a proposition whose presence is justified as a 'theorem' by the proof of that theorem given earlier in the notes.

Similarly, we will allow ourselves to write down 'assumed' as a justification of a proposition that has already been shown to be provable (from a given set of hypotheses).

Proposition 2.33. *For any α in $\mathcal{L}_0$, the following is a theorem:*

$$\vdash ((\neg\neg\alpha) \Rightarrow \alpha)$$

Proof. By the Deduction Theorem, it suffices to give a proof of $\{\neg\neg\alpha\} \vdash \alpha$; we do so below:

1.	$\neg\neg\alpha$	Hypothesis
2.	$((\neg\alpha) \Rightarrow (\neg\neg\alpha)) \Rightarrow (((\neg\alpha) \Rightarrow (\neg\alpha)) \Rightarrow \alpha)$	Axiom 3
3.	$(\neg\neg\alpha) \Rightarrow ((\neg\alpha) \Rightarrow (\neg\neg\alpha))$	Axiom 1
4.	$(\neg\alpha) \Rightarrow (\neg\neg\alpha)$	Modus ponens on lines 1, 3
5.	$((\neg\alpha) \Rightarrow (\neg\alpha)) \Rightarrow \alpha$	Modus ponens on lines 2, 4
6.	$(\neg\alpha) \Rightarrow (\neg\alpha)$	'theorem'
7.	α	Modus ponens on lines 5, 6

On line 6 of the above proof, we assumed the theorem which was given in Proposition 2.28. □

Proposition 2.34. *For any α in $\mathcal{L}_0$, the following is a theorem:*

$$\vdash (\alpha \Rightarrow (\neg\neg\alpha))$$

Proof. By the Deduction Theorem, it suffices to give a proof of $\{\alpha\} \vdash (\neg\neg\alpha)$, we do so below:

1.	α	Hypothesis
2.	$((\neg\neg\neg\alpha) \Rightarrow (\neg\alpha)) \Rightarrow (((\neg\neg\neg\alpha) \Rightarrow \alpha) \Rightarrow (\neg\neg\alpha))$	Axiom 3
3.	$(\neg\neg\neg\alpha) \Rightarrow (\neg\alpha)$	'theorem'
4.	$((\neg\neg\neg\alpha) \Rightarrow \alpha) \Rightarrow (\neg\neg\alpha)$	Modus ponens on lines 2, 3
5.	$\alpha \Rightarrow ((\neg\neg\neg\alpha) \Rightarrow \alpha)$	Axiom 1
6.	$(\neg\neg\neg\alpha) \Rightarrow \alpha$	Modus ponens on lines 1, 5
7.	$\neg\neg\alpha$	Modus ponens on lines 4, 6

On line 3 of the above proof, we assumed the theorem which was given in Proposition 2.33. □

Proposition 2.35. *For any α, $\beta \in \mathcal{L}_0$, the following is a theorem:*

$$\vdash ((\neg\alpha) \Rightarrow (\alpha \Rightarrow \beta))$$

Proof. By an application of the Deduction Theorem, it suffices to give a proof of the syntactic implication $\{\neg\alpha\} \vdash (\alpha \Rightarrow \beta)$.

A further application of the Deduction Theorem then indicates that it suffices to give a proof of $\{\neg\alpha, \alpha\} \vdash \beta$; we do so below:

1.	α	Hypothesis
2.	$\neg\alpha$	Hypothesis
3.	$((\neg\beta) \Rightarrow (\neg\alpha)) \Rightarrow (((\neg\beta) \Rightarrow \alpha) \Rightarrow \beta)$	Axiom 3
4.	$(\neg\alpha) \Rightarrow ((\neg\beta) \Rightarrow (\neg\alpha))$	Axiom 1
5.	$(\neg\beta) \Rightarrow (\neg\alpha)$	Modus ponens on lines 2, 4
6.	$((\neg\beta) \Rightarrow \alpha) \Rightarrow \beta$	Modus ponens on lines 3, 5
7.	$\alpha \Rightarrow ((\neg\beta) \Rightarrow \alpha)$	Axiom 1
8.	$(\neg\beta) \Rightarrow \alpha$	Modus ponens on lines 1, 7
9.	β	Modus ponens on lines 6, 8

□

Let us now prove one direction of the Completeness Theorem. We aim to prove that our proofs 'make sense', i.e. that the syntactic aspects of propositional logic, as described above, are 'sound':

Theorem 2.36 (Soundness Theorem for propositional logic)**.** *Let S be a set of propositions ($S \subset \mathcal{L}_0$) and α be a proposition ($\alpha \in \mathcal{L}_0$):*

$$\textit{If} \quad S \vdash \alpha \qquad \textit{then} \quad S \models \alpha$$

Proof. Consider a proof of α from S, consisting of the lines/propositions $t_1, \cdots, t_n$ say. Let us show that if v is a valuation satisfying $v(s) = 1$ for all $s \in S$, then every line of the proof is true, i.e. that $v(t_i) = 1$ for each line t_i, $1 \leq i \leq n$, in the proof.

Consider such a valuation, $v : \mathcal{L}_0 \to \{0, 1\}$. For each t_i, $1 \leq i \leq n$, we have the following cases:

A) The proposition t_i is an (occurrence of an) axiom. All our axioms are tautologies, so $v(t_i) = 1$ (for any valuation v, in fact).

B) The proposition t_i is a hypothesis, i.e. it is a proposition in S. Then, $v(t_i) = 1$, by assumption.

C) The proposition t_i is deduced by modus ponens from two preceding propositions.

So, suppose that, for some $j < i$, $k < i$ and $\beta, \gamma \in \mathcal{L}_0$, the proposition t_j is β, the proposition t_k is $\beta \Rightarrow \gamma$, and t_i is the proposition γ, obtained by modus ponens on lines t_j, t_k.

Then, since t_i follows t_j and t_k, we may, inductively, assume that the result we are trying to prove already holds for t_j and t_k, i.e. that $v(t_j) = v(\beta) = 1$ and $v(t_k) = v(\beta \Rightarrow \gamma) = 1$. This ensures that $v(\gamma) = 1$, by definition of a valuation (may be shown directly, or by contradiction: if $v(\gamma) = 0$, then either $v(\beta) = 1$, from which we obtain $v(\beta \Rightarrow \gamma) = 0$, or $v(\beta) = 0$; in either case, this contradicts our inductive assumption(s)). So, it must be the case that $v(t_i) = v(\gamma) = 1$, as required.

Hence, we have shown that, if v is a valuation satisfying $v(s) = 1$ for all s in S, then every line t_i, $1 \leq i \leq n$, in the proof of α from S, satisfies $v(t_i) = 1$. In particular, $v(t_n) = v(\alpha) = 1$, so that $S \models \alpha$, as required.

□

Proving the result in the other direction, i.e. that "if $S \models \alpha$, then $S \vdash \alpha$" will appear a bit more tricky, perhaps (partly) due to the following reasons:

1) Our proof should hold for any subset S of $\mathcal{L}_0$, whether finite or infinite, so we would not be able to readily emulate the proof of the Soundness Theorem, where it was possible to involve the lines present in a proof (a proof, by definition, consists of a finite sequence of propositions).

2) We are assuming that some, but not all, tautologies are axioms, but still want to show that any other 'true' proposition is provable from them.

To see that the above points makes a difference to the complexity of the proof to follow, let us see how relatively simple it would be to prove the desired result if S was finite, say if $S = \{s_1, \cdots, s_n\}$, and if every tautology was an axiom.

Under these two assumptions, suppose that $S \models \alpha$, i.e. that

$$\{s_1, \cdots, s_n\} \models \alpha$$

By Proposition 2.32, $\{s_1, \cdots, s_{n-1}\} \models (s_n \Rightarrow \alpha)$, and, proceeding similarly, we obtain:

$$\models (s_1 \Rightarrow (\cdots (s_{n-1} \Rightarrow (s_n \Rightarrow \alpha))))$$

Therefore, $s_1 \Rightarrow (\cdots (s_{n-1} \Rightarrow (s_n \Rightarrow \alpha)))$ is a tautology, so, by assumption, it is also an axiom, and so:

$$\vdash (s_1 \Rightarrow (\cdots (s_{n-1} \Rightarrow (s_n \Rightarrow \alpha))))$$

Then, by the Deduction Theorem, $\{s_1\} \vdash (s_2 \Rightarrow (\cdots (s_{n-1} \Rightarrow (s_n \Rightarrow \alpha))))$, and, proceeding similarly, we obtain:

$$\{s_1, \cdots, s_n\} \vdash \alpha$$

So, under the restrictions imposed above, we have shown that, if $S \models \alpha$, then $S \vdash \alpha$, as required.

Keeping these possible 'technical' issues in mind, we will proceed to give a proof of the other direction of the Completeness Theorem for propositional logic.

A key idea in our proof will be that of *consistency*, which is 'dual' to the idea of having a model, as we shall see. Let us remember that a subset S of $\mathcal{L}_0$ has a model if there exists a valuation $v : \mathcal{L}_0 \to \{0, 1\}$ such that $v(s) = 1$ for all $s \in S$. Let us now give the related idea of consistency, which is defined in the setting of proofs:

Definition 2.37. *Consider a set of propositions, S ($S \subset \mathcal{L}_0$).*

The set S is consistent *if there does not exist a proposition α such that $S \vdash \alpha$ and $S \vdash (\neg\alpha)$.*

If there exists a proposition α such that $S \vdash \alpha$ and $S \vdash (\neg\alpha)$, we say that the set S is inconsistent.

One of the key steps in the proof of the main result that we will use to prove the 'remainder' of the Completeness Theorem relies on the fact that, given a consistent set S, we can (indefinitely) extend it to a larger consistent set:

Proposition 2.38. *If S is a consistent set of propositions ($S \subset \mathcal{L}_0$), then, for any proposition α ($\alpha \in \mathcal{L}_0$), at least one of $S \cup \{\alpha\}$, $S \cup \{\neg\alpha\}$ is consistent.*

Proof. Consider $S \cup \{\neg\alpha\}$. Either this is a consistent set, in which case we are done, or it is an inconsistent set. Let us show that, in the latter case, the set $S \cup \{\alpha\}$ is consistent. We will do this by showing that $S \vdash \alpha$, so that the sets S and $S \cup \{\alpha\}$ are equivalent in terms of what can be proven from them as hypotheses. This will mean that $S \cup \{\alpha\}$ 'inherits' consistency from the set S.

So, suppose that $S \cup \{\neg\alpha\}$ is inconsistent. Then, there exists a proposition β such that $S \cup \{\neg\alpha\} \vdash \beta$ and $S \cup \{\neg\alpha\} \vdash (\neg\beta)$. Using the Deduction Theorem, we may therefore determine that the following two syntactic implications hold: $S \vdash ((\neg\alpha) \Rightarrow (\neg\beta))$ and $S \vdash ((\neg\alpha) \Rightarrow \beta)$.

So, since $S \vdash (\neg\alpha) \Rightarrow (\neg\beta)$ and $S \vdash (\neg\alpha) \Rightarrow \beta$, we may deduce that $S \vdash \alpha$ (using Axiom 3 and two applications of modus ponens):

1.	$(\neg\alpha) \Rightarrow \beta$	'assumed'
2.	$(\neg\alpha) \Rightarrow (\neg\beta)$	'assumed'
3.	$((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$	Axiom 3
4.	$((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha$	Modus ponens on lines 2, 3
5.	α	Modus ponens on lines 1, 4

Hence $S \vdash \alpha$, as desired. This is particularly useful, as described above, because, if $S \vdash \alpha$, then the consistency of $S \cup \{\alpha\}$ reduces to the consistency of S. This is because S and $S \cup \{\alpha\}$ prove precisely the same propositions: for any proposition γ, $S \cup \{\alpha\} \vdash \gamma$ if and only if $S \vdash \gamma$ (e.g. see Lemma 2.31).

Since S is assumed to be consistent, there is no proposition β such that $S \vdash \beta$ and $S \vdash (\neg\beta)$. Therefore, using the result from the previous paragraph, we may deduce that there is no proposition β such that $S \cup \{\alpha\} \vdash \beta$ and $S \cup \{\alpha\} \vdash (\neg\beta)$. Hence, $S \cup \{\alpha\}$ is (also) consistent, as required.

□

Let us now prove the following theorem, which directly relates the semantic notion of a set of propositions having a model to the syntactic notion of a set of propositions being consistent. We will use this result in order to help us prove the 'missing direction' of the Completeness Theorem:

Theorem 2.39. *Let S be a set of propositions ($S \subset \mathcal{L}_0$). If S is consistent, then S has a model.*

Proof. Starting from the assumption that S is consistent, we try to define a valuation $v : \mathcal{L}_0 \to \{0, 1\}$, such that $v(s) = 1$ for s in S. Note that, for a function we define from $\mathcal{L}_0$ to $\{0, 1\}$ to be a valuation, we must ensure that, for each proposition β, either $v(\beta) = 0$ and $v(\neg\beta) = 1$, or $v(\beta) = 1$ and $v(\neg\beta) = 0$.

Let us construct a suitable function $v : \mathcal{L}_0 \to \{0, 1\}$. First, let us set $v(s) = 1$ for all s in S.

Now $\mathcal{L}_0$ is a countable set (the set of primitive propositions is countable, so the set of propositions of degree 1 is countable, the set of propositions of degree 2 is countable, and so on, inductively; finally, the union of the set of all propositions of all degrees is also countable; see also Note 2.6), so we may list the propositions in $\mathcal{L}_0$, say:

$$\mathcal{L}_0 = \{\alpha_1, \alpha_2, \cdots, \alpha_n, \cdots\}$$

Consider S and α_1. We assume that S is consistent, so, using Proposition 2.38, we may deduce that at least one of $S \cup \{\alpha_1\}$, $S \cup \{\neg\alpha_1\}$ is consistent. If $S \cup \{\alpha_1\}$ is consistent, then set $v(\alpha_1) = 1$, $v(\neg\alpha_1) = 0$ and set $S_1 = S \cup \{\alpha_1\}$. Otherwise, set $v(\alpha_1) = 0$, $v(\neg\alpha_1) = 1$ (in this case, $S \cup \{\neg\alpha_1\}$ is consistent) and set $S_1 = S \cup \{\neg\alpha_1\}$. In either case, S_1 is consistent, and $v(s) = 1$ for each s in S_1.

Then, consider S_1 and α_2. If $S_1 \cup \{\alpha_2\}$ is consistent, then set $v(\alpha_2) = 1$, $v(\neg\alpha_2) = 0$ and set $S_2 = S_1 \cup \{\alpha_2\}$. Otherwise, set $v(\alpha_2) = 0$, $v(\neg\alpha_2) = 1$ and set $S_2 = S_1 \cup \{\neg\alpha_2\}$ (in this case, $S_1 \cup \{\neg\alpha_2\}$ is consistent). In either case, S_2 is consistent, and $v(s) = 1$ for each s in S_2.

We may proceed, inductively, to form S_3, S_4, $\cdots$, S_n, $\cdots$

As constructed, for each n, S_n is consistent, and $v(s) = 1$ for each s in S_n. In addition:

$$S \subset S_1 \subset S_2 \subset \cdots \subset S_n \subset \cdots$$

We wish to define a valuation on the whole of $\mathcal{L}_0$; we consider the following set:

$$\bar{S} = \bigcup_{n \in \mathbb{N}} S_n$$

Note that, for any proposition α, either $\alpha \in \bar{S}$ or $\neg\alpha \in \bar{S}$.

We will show that $\bar{S}$ is *consistent*, using a proof by contradiction.

Suppose that $\bar{S}$ is inconsistent. Then, for some $\beta \in \mathcal{L}_0$, there exists a proof of β from $\bar{S}$, and there exists a proof of $\neg\beta$ from $\bar{S}$. But, since a proof is a finite sequence of propositions, the inconsistency of $\bar{S}$ implies that there must exist a proof of β from some (finite subset of some) S_m, and a proof of $\neg\beta$ from some (finite subset of some) S_n, for some positive integers m, n. So, for some S_k (for example, we could choose k to be any number greater than or equal to both m and n), there exists a proof of β from S_k and a proof of $\neg\beta$ from S_k; this contradicts the fact that S_k is consistent. Hence, it must be the case that the set $\bar{S}$ is (also) consistent.

The set $\bar{S}$ is also *deductively closed*, which means that if we can prove a proposition from $\bar{S}$, the proposition is a member of $\bar{S}$ itself: if $\bar{S} \vdash \beta$, then $\beta \in \bar{S}$.

Let us prove that $\bar{S}$ is deductively closed, by contradiction. Suppose that $\bar{S}$ is not deductively closed, i.e. that, for some proposition β, $\bar{S} \vdash \beta$, but $\beta \notin \bar{S}$. By definition of $\bar{S}$, it contains either β or $\neg\beta$. So, in this case, $\bar{S}$ must contain $\neg\beta$. But then, $\bar{S} \vdash (\neg\beta)$, so that, since we also assumed that $\bar{S} \vdash \beta$, we may deduce that $\bar{S}$ is inconsistent. This contradicts the fact that $\bar{S}$ is consistent (shown above). Therefore, we deduce that $\beta \in \bar{S}$, and, hence, that $\bar{S}$ is deductively closed, as required.

We will now use the two properties of $\bar{S}$ mentioned above in order to show that $v : \mathcal{L}_0 \to \{0, 1\}$, the function we have constructed, is a valuation. Note that the process described above, which leads to the formation of S_k, for $k \in \mathbb{N}$, allows us to define v for any proposition in $\mathcal{L}_0$. It remains to show that the function v satisfies the defining properties of a valuation and that it models S.

Let us first verify that the described function $v : \mathcal{L}_0 \to \{0, 1\}$ is a valuation:

1) We must verify that, for $\alpha \in \mathcal{L}_0$, $v(\neg\alpha) = 1$ if $v(\alpha) = 0$ while $v(\neg\alpha) = 0$ if $v(\alpha) = 1$. The construction of v ensures that this holds.

2) We must verify that, if $\alpha, \beta \in \mathcal{L}_0$, then $v(\alpha \Rightarrow \beta) = 0$ if $v(\alpha) = 1$ and $v(\beta) = 0$ while $v(\alpha \Rightarrow \beta) = 1$ otherwise. Note that, for any proposition $\alpha \in \mathcal{L}_0$, $v(\alpha) = 1$ if and only if $\alpha \in \bar{S}$, by construction.

   Let us first verify that, if $v(\beta) = 1$, then $v(\alpha \Rightarrow \beta) = 1$, irrespective of the value of $v(\alpha)$. Suppose that $v(\beta) = 1$. Then $\beta \in \bar{S}$, so that $\bar{S} \vdash \beta$. Also, using Axiom 1, the syntactic implication $\bar{S} \vdash (\beta \Rightarrow (\alpha \Rightarrow \beta))$ holds. So, by modus ponens, we may deduce that $\bar{S} \vdash (\alpha \Rightarrow \beta)$. Therefore $(\alpha \Rightarrow \beta) \in \bar{S}$ (since $\bar{S}$ is deductively closed). Hence, $v(\alpha \Rightarrow \beta) = 1$, as required.

   Let us next verify that, if $v(\alpha) = 0$, then $v(\alpha \Rightarrow \beta) = 1$, irrespective of the value of $v(\beta)$. Suppose that $v(\alpha) = 0$. Then, by construction, $v(\neg\alpha) = 1$, so that $(\neg\alpha) \in \bar{S}$ and $\bar{S} \vdash \neg\alpha$. Note that the proposition $(\neg\alpha) \Rightarrow (\alpha \Rightarrow \beta)$ is a theorem (e.g. see Proposition 2.35), so that $\bar{S} \vdash ((\neg\alpha) \Rightarrow (\alpha \Rightarrow \beta))$. We may therefore use modus ponens to deduce that $\bar{S} \vdash (\alpha \Rightarrow \beta)$. Therefore $(\alpha \Rightarrow \beta) \in \bar{S}$, and so $v(\alpha \Rightarrow \beta) = 1$, as required.

   Finally, let us verify that, if $v(\alpha) = 1$ and $v(\beta) = 0$, then $v(\alpha \Rightarrow \beta) = 0$. Suppose that $v(\alpha) = 1$ and $v(\beta) = 0$. Let us show that, in this case, $v(\alpha \Rightarrow \beta) = 0$, by using a proof by contradiction. Suppose that $v(\alpha \Rightarrow \beta) = 1$. Then $(\alpha \Rightarrow \beta) \in \bar{S}$, i.e. $\bar{S} \vdash (\alpha \Rightarrow \beta)$. We also know that $\bar{S} \vdash \alpha$, since $\alpha \in \bar{S}$ (this follows from $v(\alpha) = 1$). Then, a use of modus ponens on $\bar{S} \vdash (\alpha \Rightarrow \beta)$ and $\bar{S} \vdash \alpha$ allows us to deduce that $\bar{S} \vdash \beta$, so that $\beta \in \bar{S}$ and $v(\beta) = 1$. This contradicts the assumption that $v(\beta) = 0$. Hence, $v(\alpha \Rightarrow \beta) = 0$, as required.

Finally, the valuation $v : \mathcal{L}_0 \to \{0, 1\}$ does model S, since, by construction of v, $v(s) = 1$ for all s in S. Therefore, S has a model, as required.

□

The above theorem shows that, if a set of propositions S is consistent, then S has a model. Equivalently, if S does not have a model, then S is inconsistent. It is this version of the above theorem that we will use to prove the following theorem, which shows that the version of proof described in this chapter (e.g. see Definition 2.25) is 'adequate'; it allows us to show that it is possible to prove all propositions which are true when an original set of propositions (hypotheses) is true:

Theorem 2.40 (Adequacy Theorem for propositional logic)**.** *Let S be a set of propositions ($S \subset \mathcal{L}_0$) and α be a proposition ($\alpha \in \mathcal{L}_0$):*

$$\textit{If} \quad S \models \alpha \qquad \textit{then} \quad S \vdash \alpha$$

Proof. Let us suppose that α is a proposition and that S is a set of propositions such that $S \models \alpha$. Then, if $v : \mathcal{L}_0 \to \{0, 1\}$ is any valuation such that $v(s) = 1$ for all s in S, we have $v(\alpha) = 1$. So, there is no valuation $v : \mathcal{L}_0 \to \{0, 1\}$ such that $v(s) = 1$ for all $s \in S$ and $v(\neg\alpha) = 1$, i.e. the set $S \cup \{\neg\alpha\}$ does not have a model.

By Theorem 2.39, we may therefore deduce that the set $S \cup \{\neg\alpha\}$ is inconsistent, so that there exists some proposition $\beta \in \mathcal{L}_0$ such that $S \cup \{\neg\alpha\} \vdash \beta$ and $S \cup \{\neg\alpha\} \vdash (\neg\beta)$. We now proceed similarly to the proof of Proposition 2.38.

We start by noting that, since $S \cup \{\neg\alpha\} \vdash \beta$ and $S \cup \{\neg\alpha\} \vdash (\neg\beta)$, we may use the Deduction Theorem to determine that $S \vdash ((\neg\alpha) \Rightarrow (\neg\beta))$ and $S \vdash ((\neg\alpha) \Rightarrow \beta)$.

Then, since $S \vdash (\neg\alpha) \Rightarrow (\neg\beta)$ and $S \vdash (\neg\alpha) \Rightarrow (\neg\beta)$, we may deduce that $S \vdash \alpha$ (using Axiom 3 and two applications of modus ponens):

1.	$(\neg\alpha) \Rightarrow \beta$	'assumed'
2.	$(\neg\alpha) \Rightarrow (\neg\beta)$	'assumed'
3.	$((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$	Axiom 3
4.	$((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha$	Modus ponens on lines 2, 3
5.	α	Modus ponens on lines 1, 4

The above argument shows that $S \vdash \alpha$, i.e. that there exists a proof of the proposition α from the set of propositions S, as required. □

We have now completed the proof of both directions of implication in the Completeness Theorem for propositional logic:

Theorem 2.41 (Completeness Theorem for propositional logic)**.** *Let S be a set of propositions ($S \subset \mathcal{L}_0$) and α be a proposition ($\alpha \in \mathcal{L}_0$). Then:*

$$S \models \alpha \qquad \textit{if and only if} \qquad S \vdash \alpha$$

Proof. The result follows from the Soundness and Adequacy Theorems for propositional logic (see Theorems 2.36 and 2.40 respectively). □

The Completeness Theorem essentially concludes our study of propositional logic. It might be seen to be an inherently important result, since it shows the equivalence of the semantic and syntactic settings of propositional logic, as described in this chapter, and therefore brings together the notions of 'truth' and 'provability'.

It is also useful as a bridge between these settings, in the sense that, if one wishes to prove a certain result in one of the two settings we have seen, a direct demonstration might appear quite complicated or elusive, but, for certain results at least, using the Completeness Theorem to move from one setting to the other may yield a demonstration much more readily.

We will see two such results next; they may be regarded as corollaries of the Completeness Theorem for propositional logic.

The first result is expressed in the setting of valuations and models. It essentially says that if a proposition α is true whenever an infinite set of propositions S is true, then we will also be able to show that α is true whenever some finite, 'compact', subset of the original set of propositions is true. Valuations, by construction, are defined on an infinite set, namely $\mathcal{L}_0$. However, proofs are finite sequences of propositions; moving to the setting of proofs allows us to deduce this result relatively easily:

Theorem 2.42 (Compactness Theorem for propositional logic)**.** *Consider a (possibly infinite) set of propositions S in $\mathcal{L}_0$ and a proposition α in $\mathcal{L}_0$, such that $S \models \alpha$. Then, there is a finite subset S' of S, such that $S' \models \alpha$.*

Proof. Suppose that $S \models \alpha$. Then, by the Completeness Theorem, we may deduce that $S \vdash \alpha$, i.e. that there exists a proof of α from S. Proofs are finite sequences of propositions, so any proof of α from S uses only a finite subset, S' say, of the propositions in S. Therefore, there exists a proof of α from a finite subset of S, i.e. for a finite subset S', say, of S, $S' \vdash \alpha$. Using the Completeness Theorem once again, we may deduce that $S' \models \alpha$ for a finite subset S' of S, as required. □

The second result we will see is a result concerning proofs. It might not be obvious how to determine if there is a way to 'decide' whether or not there exists a proof of a given proposition from a given (finite) set of hypotheses. However, if we translate this to the setting of valuations and models, there are many readily available ways of 'deciding' whether or not any given proposition is true, assuming that each proposition, in a given (finite) set of propositions, is true.

Theorem 2.43 (Decidability Theorem for propositional logic)**.** *Consider any finite set of propositions S in $\mathcal{L}_0$ and any proposition $\alpha \in \mathcal{L}_0$. Then, there is an algorithm which (in a finite number of steps) allows us to determine whether or not there exists a proof of α from S (i.e. whether or not $S \vdash \alpha$).*

Proof. By the Completeness Theorem, $S \vdash \alpha$ if and only if $S \models \alpha$. So, it is sufficient to show that there is a 'finite' algorithm that determines whether or not $S \models \alpha$. However, to check whether or not $v(\alpha) = 1$ for any valuation v satisfying $v(s) = 1$ for all s in S, it is sufficient to write down an appropriate truth table containing a finite number of 'cells' (such a 'finite' truth table will exist, since there is a finite number of propositions in the set composed of S and α, as well as a finite number of primitive propositions in the elements of this set). So, there does exist an algorithm which (in a finite number of steps) allows us to determine whether or not $S \models \alpha$, and, therefore, by the Completeness Theorem, whether or not $S \vdash \alpha$, as required. □

Note that we could also construct a suitable algorithm in the proof of the Decidability Theorem by using a method other than one involving truth tables, e.g. the semantic tableaux method, which, by hypothesis, will terminate after a finite amount of 'branching'.

In this chapter, we have investigated the notions of truth and provability in the setting of propositional logic. The next chapter involves such a study in the case of first order predicate logic. It includes a description of a possible extension of the definitions and results given here, which will allow us to accommodate the presence of variables, predicates, functionals and the '$\forall$' and '$\exists$' symbols, in the setting of first order predicate logic, as well as to eventually phrase and prove results similar to some of the ones that appear in this chapter, such as a Completeness Theorem for first order predicate logic.

In the setting of first order predicate logic, we will also have the opportunity to study examples of mathematical structures that can be defined within the setting of the general first order predicate language, as well as to examine some of the consequences of working in this setting.

Chapter 3

First Order Predicate Logic

In the previous chapter, we investigated the notions of truth and provability, and the interplay between these notions, in the setting of propositional logic, using $\mathcal{L}_0$. Here, we will investigate the same ideas in the setting of first order predicate logic, and $\mathcal{L}$ or $\mathcal{L}_{math}$.

3.1 Semantic aspects of first order predicate logic

In chapter 2, the notion of 'truth' was relatively simple to define and study. We could assign truth or falsehood to any primitive proposition in $\mathcal{L}_0$, and then uniquely extend this assignment to (a valuation on) the 'whole' of $\mathcal{L}_0$. In this chapter, an extra 'layer' of 'truth assignment' is needed. We may consider the point of view that our variables are meant to represent objects or elements in a set. Then, to determine whether or not a given statement is true, we require to be provided with a set or structure; we may then try to determine if the given statement is true 'there'.

Note that, in the setting of first order predicate logic, we can use variable symbols, so that our statements may include:

- the '$\forall$' symbol (and, as a result, the '$\exists$' symbol)
- any predicate symbol of any arity
- any functional symbol of any arity

As mentioned above, a formula will not necessarily be true or false by itself. It will only 'become' true or false, given a setting or structure.

Consider the formula that, informally, says:

"every element has an inverse, under some binary functional"

If F is a binary functional, x, y are variables, and E is the 'identity' functional, we may express the informal version of the formula above as $(\forall x)(\exists y)(Fxy = E)$. Then:

- If we are in $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, with F denoting addition and E denoting 0, the formula is 'false'.
- If we are in $\mathbb{Z}$, with F denoting addition and E denoting 0, the formula is 'true'.
- If we are in $\mathbb{Z}$, with F denoting multiplication and E denoting 1, the formula is 'false'.

Let us formalise the connection between a formula, which is simply a string of symbols in $\mathcal{L}$ or $\mathcal{L}_{math}$, and a structure that the formula is able to 'say something about'.

Suppose that we are given a set of predicates, Π, and a set of functionals, Ω, of assigned arities, in the setting of $\mathcal{L}_{math}$. Let us refer to any formula α that we can obtain using Π and Ω (together with the other non-predicate and non-functional symbols of $\mathcal{L}$) as an element of (the language) $\mathcal{L}(\Pi, \Omega)$, and write $\alpha \in \mathcal{L}(\Pi, \Omega)$.

Definition 3.1. *Given a set of predicate symbols Π and a set of functional symbols Ω, with assigned arities, an $\mathcal{L}(\Pi, \Omega)$-*structure *consists of the following:*

1) A non-empty set U.

2) For each predicate symbol P of arity n in Π, an n-ary predicate, or relation, P_U on U.

3) For each functional symbol F of arity n in Ω, an n-ary functional, or function, F_U on U, i.e. a functional $F_U : U^n \to U$ (if $n = 0$, F_U corresponds to a specific, constant, element of U).

Note 3.2. *For convenience, we often regard the non-empty set U underlying an $\mathcal{L}(\Pi, \Omega)$-structure as 'being' the $\mathcal{L}(\Pi, \Omega)$-structure itself, and thus refer to 'an $\mathcal{L}(\Pi, \Omega)$-structure U'.*

Note 3.3. *In order to agree well with the conventions adopted when defining the first order predicate language (e.g. see beginning of chapter 1), we will assume, for the most part, that the structures we are working with in this chapter have a countable underlying set U (so that U is finite or countably infinite). This will be a necessary assumption for some results.*

For example, Definition 3.9 requires the introduction of one constant, or 0-ary functional, for each element of the set U underlying a given structure. So, since we have assumed the existence of (only)

a countable infinity of functional symbols, we are only equipped to deal with structures for which the underlying set U is finite or countably infinite.

However, many of the results we shall see also hold in the case of 'larger' structures and languages (e.g. see Theorem 3.35)

Let us now give an example of the kind of structure that it might be possible to describe using a specified $\mathcal{L}(\Pi, \Omega)$:

Example 3.4. *Suppose that:*

$\Pi = \{=\}$, *where:*

the predicate '$=$' has arity 2

$\Omega = \{F_1, F_2, E_1, E_2\}$, *where:*

the functionals 'F_1', 'F_2' each have arity 2

the functionals 'E_1', 'E_2' each have arity 0.

Then, the set of integers, $\mathbb{Z}$, (or, indeed, any ring) may be viewed as an $\mathcal{L}(\Pi, \Omega)$-structure; for example, set F_1 to be addition, E_1 to be the '0' in $\mathbb{Z}$ (or the ring in general), F_2 to be multiplication, E_2 to be the '1' in $\mathbb{Z}$ (or the ring in general), and let '$=$' denote the usual equality predicate.

Similarly, the set of real numbers, $\mathbb{R}$, (or, indeed, any field) may be viewed as an $\mathcal{L}(\Pi, \Omega)$-structure, and so can the set $\mathbb{N}_0 = \{0, 1, 2, \cdots\}$, with addition and multiplication 'interpreted' as above.

Note that we have not given any rules that an $\mathcal{L}(\Pi, \Omega)$-structure must satisfy, yet.

Furthermore, within the setting of $\mathcal{L}(\Pi, \Omega)$-structures, we seem to be treating equality as 'just another' predicate. However, in order to preserve its special meaning in 'mathematical systems', we will, later, introduce a specific set of rules that any structure with an equality predicate must satisfy.

Once we have a structure, we can, in general, try to define a notion of meaning, and perhaps search for 'defining rules', which the given structure, as well as related structures, must satisfy.

Before we do that, let us mention some technical considerations related to variables and the use of the '$\forall$' (and, as a result, the '$\exists$') symbol in $\mathcal{L}_{math}$ (or $\mathcal{L}$).

We would like to be able to substitute one variable symbol for another, e.g we might want to say that '$(\forall x)Px$' has the same meaning as '$(\forall y)Py$'.

In some cases, some problems may arise when using such substitutions.

For example, consider the formula '$(\forall x)(Pyx)$'. If we happen to replace every variable symbol x in this formula by the variable symbol y, we obtain the formula '$(\forall y)(Pyy)$', which may have a different meaning from the original formula.

In order to see a specific instance of this, suppose that P is the relation '$\leq$', and that we are given a structure, consisting of the natural numbers, $\mathbb{N} = \{1, 2, \cdots\}$, and having only the predicate P. Then '$(\forall x)(Pyx)$' says that 'for all x, $y \leq x$' which is true depending on which element of $\mathbb{N}$ is assigned to the variable y (it is true if y takes the value $1 \in \mathbb{N}$, and false otherwise), while '$(\forall y)(Pyy)$' says that 'for all y, $y \leq y$', which is a different statement, that always holds (whichever value of y we choose), in the setting of $\mathbb{N}$.

We (could) agree to only ever substitute by variable symbols that are not already present in a formula (or, at least, symbols that will not affect the 'meaning').

Another possibly confusing type of formula is, for example, '$(\forall x Px) \wedge (Qxy)$' (for a unary predicate P and a binary predicate Q).

The first (leftmost) two occurrences of the variable symbol x are 'bound together', but the third (rightmost) one is 'free' (i.e. not within a scope of a '$\forall x$'). In such cases, we might consider using a different variable symbol for the occurrences of x that are 'connected' above, so as to emphasise that (only) two uses of the variable symbol x are 'bound together'. We could, write, for example, '$(\forall t Pt) \wedge (Qxy)$', without changing the meaning of the statement.

The uses of the variable symbol x in the above example are similar to the various uses of x in

$$\int_0^x f(x)\, dx$$

Reading from left to right, the second and third uses of the symbol x are linked, while the first one denotes the upper limit of integration, which is free to be assigned any particular number in the final answer. If we substitute t for x in the second and third cases, we obtain an expression which has the same meaning as the original one, and leads to the same final answer, but which is less ambiguous, in the sense described above:

$$\int_0^x f(t)\, dt$$

Let us distinguish between the possibly different occurrences of a variable symbol in an expression:

Definition 3.5. *An occurrence of a variable symbol, x say, in a formula $\alpha \in \mathcal{L}_{math}$, is said to be* free *if it is not within the scope of a '$\forall x$', i.e. if x is not present in a formula to which a '$\forall x$' applies.*

Otherwise, an occurrence of x within the scope of a '$\forall x$' is said to be bound *(the x present within a '$\forall x$' string is also said to be* bound*).*

For example (below, x, y, z are variable symbols, P is a unary predicate symbol and Q is a binary predicate symbol):

- In the formula $(\forall x)(Px)$, the two occurrences of the variable x are bound.
- In the formula $(\forall x)(Qxy)$, the two occurrences of the variable x are bound, whereas the occurrence of the variable y is free.
- In the formula $(\forall x)(Qxx)$, the three occurrences of the variable x are bound.
- In the formula $(\forall x)(Qyz)$, the occurrence of the variable x is bound, whereas the occurrences of the variables y and z are free.
- In the formula $((\forall x)(Px)) \vee (Qxy)$, the first (leftmost) two occurrences of the variable x are bound, whereas the final (rightmost) occurrence of the variable x (as a variable associated to the predicate Q) is free. The occurrence of the variable y is free.

As the last example shows, as well as the two examples preceding Definition 3.5, it might be the case that a given formula contains both free and bound occurrences of the same variable (symbol).

We could always change the bound occurrences of such a variable symbol, by substituting by a variable symbol not already present in our formula, as we did in the two examples preceding Definition 3.5, in order to make sure that no variable (symbol) appears both bound and free in the same formula.

We will assume that we can overcome these 'technical difficulties', so that the formulae we use are *clean*: they do not contain both bound and free occurrences of the same variable symbol.

So, we will assume that any formula is always 'logically equivalent' to a clean formula. This 'logical equivalence' will be made more precise soon, when we consider interpretations of formulae in $\mathcal{L}(\Pi, \Omega)$-structures (see Definition 3.9).

It might also be worthwhile to realise that general statements, of the type we may wish to use as 'defining rules' for mathematical 'objects', do not contain any occurrences of free variables, e.g. please consider the defining statements of posets and groups, given in Examples 1.15 and 1.16 respectively; such statements are sentences:

Definition 3.6. *A* sentence *is a formula containing no free (occurrences of) variables.*

We now return to the notion of 'meaning' within a structure. Unlike in the setting of propositional logic, described in chapter 2, a 'true/false' valuation will not be sufficient to ascribe 'meaning', since there are some valid formulae to which we might not be able to generally assign truth.

For example, for an n-ary *predicate* P, the formula $Px_1 \cdots x_n$ might not be true or false; however, it should be possible to interprete it as true or false, in a given structure, if, instead of general variables $x_1, \cdots, x_n$, we consider elements in the set corresponding to the structure we are considering.

Before we explain how we will assign meaning to formulae in an $\mathcal{L}(\Pi, \Omega)$-structure, let us describe the types of formulae that we will be considering for much of the remainder of this chapter.

Definition 3.7. *The set of* terms *in a first order predicate language is defined, inductively, as follows:*

(i) Each variable symbol is a term.

(ii) Each 0*-ary functional symbol is a term.*

(iii) If F *is an* n*-ary functional symbol, and* $t_1, \cdots, t_n$ *are terms, then* $Ft_1 \cdots t_n$ *is a term.*

For example, if M is a functional symbol of arity 2, E is a functional symbol of arity 0, and x, y are variable symbols, then the following are terms: E, Mxy, MxE, MEE.

For a more concrete example, consider the set of integers modulo 5, $\mathbb{Z}_5$, while using the binary functional '$+$' to denote addition and the 0-ary functionals $\bar{0}$, $\bar{1}$, $\bar{2}$, $\bar{3}$, $\bar{4}$ to denote the elements of $\mathbb{Z}_5$, and take x to be a variable.

Then, the following are examples of terms: $\bar{1}$, $x + \bar{0}$, $x + \bar{1}$, $\bar{0} + \bar{1}$, $\bar{1} + \bar{1}$, $\bar{2} + \bar{3}$.

Note that predicate symbols are excluded from the set of terms.

Let us now identify terms that, when considered within a structure, will correspond to specific elements of the set under consideration:

Definition 3.8. *A* closed term *is a term containing no variables.*

For example, if M is a functional of arity 2 and E is a functional of arity 0, then the following are closed terms: $E, MEE, M(E, MEE)$.

Similarly, for the example of $\mathbb{Z}_5$ described above, the following are closed terms: $\bar{1}$, $\bar{0} + \bar{1}$, $\bar{1} + \bar{1}$, $\bar{2} + \bar{3}$; the terms $x + \bar{0}$ and $x + \bar{1}$ are not closed.

Closed terms are the basic objects that, when 'placed inside' a structure, correspond to specific elements of the set we are considering.

It might be useful to contrast a term containing no variables (a closed term), such as $\bar{1} + \bar{1}$ for the example of $\mathbb{Z}_5$, with a general formula containing no variables, such as $\bar{1} = \bar{1}$ for the example of

$\mathbb{Z}_5$. The former may be seen to correspond to a specific element of $\mathbb{Z}_5$ (the element $\bar{2}$ in this case), while the latter is stating a 'relation' in $\mathbb{Z}_5$, which may or may not hold (in the case of $\bar{1} = \bar{1}$, we have a relation that holds, or is true, in $\mathbb{Z}_5$).

Having defined terms and closed terms, we are now able to generalise the concept of a valuation from chapter 2, so that we can also consider the effect of 'interpreting' (closed) terms within a given structure. This allows us to have an extended, workable version of 'meaning' within a general first order predicate language, assuming that we are (also) given an $\mathcal{L}(\Pi, \Omega)$-structure.

Before we do so, let us introduce some notation. Given a formula α, a variable x and a term t, we will use $\alpha[t/x]$ to denote the formula obtained by replacing each (free) occurrence of x in α by t.

For instance, if α is the formula $(\forall x)(Pyx)$, for a binary predicate symbol P and variable symbols x, y, then, for a term t, $\alpha[t/y]$ is the formula $(\forall x)(Ptx)$.

For example, suppose that P is the relation '$\leq$', and that we are given a structure, consisting of the set of natural numbers, $\mathbb{N} = \{1, 2, \cdots\}$, and including the predicate P as well as a 0-ary functional $\bar{1}$ (which corresponds to the number 1 in $\mathbb{N}$).

Then, if $\alpha = (\forall x)(Pyx)$, α corresponds to 'for all x, $y \leq x$', while $\alpha[z/y]$ corresponds to 'for all x, $z \leq x$', and $\alpha[\bar{1}/y]$ corresponds to 'for all x, $\bar{1} \leq x$'.

Having explained the above 'variable substitution notation', let us give the definition of what will allow us to 'interprete' (certain) formulae in the setting of first order predicate logic:

Definition 3.9. *An* interpretation *in a given $\mathcal{L}(\Pi, \Omega)$-structure U is defined as follows:*

1) *If F is a functional of arity n and $t_1, \cdots, t_n$ are closed terms (i.e. if $Ft_1 \cdots t_n$ is a closed term), then the interpretation of $Ft_1 \cdots t_n$ is in the set U: $(Ft_1 \cdots t_n)_U = F_U(t_1)_U \cdots (t_n)_U \in U$.*

2) *If P is a predicate of arity n, and $t_1, \cdots, t_n$ are closed terms, then the interpretation of $Pt_1 \cdots t_n$ is $(Pt_1 \cdots t_n)_U$, where $(Pt_1 \cdots t_n)_U = 1$ if $P_U(t_1)_U \cdots (t_n)_U$ holds in U, and $(Pt_1 \cdots t_n)_U = 0$ otherwise.*

3) *If $(\forall x)\alpha$ is a sentence, then the interpretation of $(\forall x)\alpha$ is $((\forall x)\alpha)_U$, which we obtain as follows: for each u in U, we introduce $\bar{u}$ as a new 0-ary functional (to $\mathcal{L}(\Pi, \Omega)$), which takes the value of u when interpreted in U. Then, we set $((\forall x)\alpha)_U = 1$, if, for each u in U, $(\alpha[\bar{u}/x])_U = 1$ (i.e. if 'α holds for each u in U'), while we set $((\forall x)\alpha)_U = 0$ if it is not true that $(\alpha[\bar{u}/x])_U = 1$ for each u in U.*

4) *For a sentence α in $\mathcal{L}(\Pi, \Omega)$, with interpretation α_U: $(\neg\alpha)_U = 1$ if $\alpha_U = 0$ and $(\neg\alpha)_U = 0$ if $\alpha_U = 1$, i.e. $(\neg\alpha)_U = 1 - \alpha_U$.*

5) For sentences α, β in $\mathcal{L}(\Pi, \Omega)$, with interpretations α_U and β_U: $(\alpha \Rightarrow \beta)_U = 0$ if $\alpha_U = 1$ and $\beta_U = 0$, and $(\alpha \Rightarrow \beta)_U = 1$ otherwise.

Note 3.10. *Not all formulae expressed in some specific language $\mathcal{L}(\Pi, \Omega)$ can be interpreted in an $\mathcal{L}(\Pi, \Omega)$-structure. For example, if x is a variable symbol and P is a unary predicate symbol, then Px has no interpretation (in the sense described above).*

Note 3.11. *As suggested in Note 3.3, (the third part of) Definition 3.9 might require the extension of $\mathcal{L}(\Pi, \Omega)$ by infinitely many constants, if the set underlying our structure consists of infinitely many elements. This is because we may need to introduce a 0-ary functional symbol $\bar{u}$ for each element u of U.*

So, if U contains infinitely many elements, we will need to introduce infinitely many 0-ary functionals into $\mathcal{L}(\Pi, \Omega)$, and this is possible in the first order predicate language we are using precisely when U is countably infinite.

Let us now give the definitions of some concepts related to interpretations (please compare with Definition 2.7):

Definition 3.12. *If, for a structure U, the sentence α satisfies $\alpha_U = 1$, then we say that α is* true *in U, or that α holds in/for U. Equivalently, we say that U* models *α, or that U is a* model *of α, and we write $U \models \alpha$.*

If there is a set T of sentences such that $U \models s$ for each $s \in T$, then we write $U \models T$, and say that U models *T, or that U is a* model *of T.*

We are now ready to describe the ways in which we can start with given predicates and functionals and describe axioms that will specify the types of mathematical 'theories' we might like to study. As mentioned earlier, the general statements that we use as axioms, which our mathematical 'theories' must satisfy, contain no free occurrences of variables, i.e. they are sentences.

Definition 3.13. *A* theory *in some (specified) language $\mathcal{L}(\Pi, \Omega)$ is a set of sentences in $\mathcal{L}(\Pi, \Omega)$.*

The sentences will be the 'defining rules' that must hold for every structure satisfying the theory. As above, if every sentence of a given theory is true for a structure U, then we will refer to the structure U as a *model* of the given theory.

Below, we construct theories for some mathematical structures.

Before we do so, let us mention some remarks related to the use of the equality predicate in what follows:

Note 3.14. *Equality, usually denoted by '=', is a predicate that we must define for every theory that we require it for, as we did in chapter* 1.

Furthermore, we will use sentences that capture the main properties of the equality predicate, and which we often assume without mentioning; these will account for the extra sentences we will include in most of the examples below, and which will be of the following types:

- *Reflexivity:*

$$(\forall x)\,(x = x)$$

- *Symmetry:*

$$(\forall x)(\forall y)\,((x = y) \Rightarrow (y = x))$$

- *Transitivity:*

$$(\forall x)(\forall y)(\forall z)\,(((x = y) \wedge (y = z)) \Rightarrow (x = z))$$

- *Substitutivity: We will include certain 'substitutivity' sentences for each predicate or functional (of arity greater than or equal to* 1*) present (in given sets* Π *and* Ω*).*

  In general, if P *is an* $(n+1)$*-ary predicate (for* $n \geq 0$*), the substitutivity sentences are:*

$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (P(z_1, \cdots, z_n, x) \Rightarrow P(z_1, \cdots, z_n, y)))$$
$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (P(z_1, \cdots, z_{n-1}, x, z_n) \Rightarrow P(z_1, \cdots, z_{n-1}, y, z_n)))$$
$$\vdots$$
$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (P(x, z_1, \cdots, z_n) \Rightarrow P(y, z_1, \cdots, z_n)))$$

  Also, in general, if F *is an* $(n+1)$*-ary functional (for* $n \geq 0$*), the substitutivity sentences are:*

$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (F(z_1, \cdots, z_n, x) = F(z_1, \cdots, z_n, y)))$$
$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (F(z_1, \cdots, z_{n-1}, x, z_n) = F(z_1, \cdots, z_{n-1}, y, z_n)))$$
$$\vdots$$
$$(\forall x)(\forall y)(\forall z_1)\cdots(\forall z_n)\,((x = y) \Rightarrow (F(x, z_1, \cdots, z_n) = F(y, z_1, \cdots, z_n)))$$

In particular, if we consider, for example, a binary predicate, denoted by '$\leq$', and a binary functional, denoted by '$\cdot$', we obtain the following four substitutivity sentences:

$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((z \leq x) \Rightarrow (z \leq y)))$$
$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((x \leq z) \Rightarrow (y \leq z)))$$
$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((z \cdot x) = (z \cdot y)))$$
$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((x \cdot z) = (y \cdot z)))$$

In this chapter, the $\mathcal{L}(\Pi, \Omega)$-structures that become models of a theory will have these additional restrictions, i.e. when the set Π of predicates contains an equality predicate '$=$', we will always include (relevant instances of) the above sentences.

Furthermore, we will assume that equality behaves as we expect, in the sense that in the set underlying a given $\mathcal{L}(\Pi, \Omega)$-structure U, it is only true that $a = b$, for closed terms a and b, precisely when a_U is the same element as b_U. So, we will assume that an equality predicate cannot identify distinct elements of an $\mathcal{L}(\Pi, \Omega)$-structure.

Models that satisfy this assumption are sometimes known as normal models*, so that a model of a theory is a normal model of that theory precisely when distinct elements of the underlying set cannot be identified by an equality predicate present in Π. In this course, we will assume that all of the models we will be using are normal models (where relevant).*

With these considerations in mind, let us now see some examples of theories:

Example 3.15. Theory of groups:

Let us first define an appropriate $\mathcal{L}(\Pi, \Omega)$*, i.e. sets of predicates and functionals we will use (please see also Example 1.16):*

$\Pi = \{=\}$*, where:*

the (equality) predicate '$=$*' has arity* 2

$\Omega = \{\cdot, E\}$*, where:*

the functional '$\cdot$*' has arity* 2

the functional 'E*'has arity* 0*.*

Then, we may use the theory consisting of the following sentences:

$$(\forall x)\left((E \cdot x = x) \wedge (x \cdot E = x)\right)$$

$$(\forall x)(\exists y)\left((x \cdot y = E) \wedge (y \cdot x = E)\right)$$

$$(\forall x)(\forall y)(\forall z)\left(((x \cdot y) \cdot z) = (x \cdot (y \cdot z))\right)$$

$$(\forall x)\left(x = x\right)$$

$$(\forall x)(\forall y)\left((x = y) \Rightarrow (y = x)\right)$$

$$(\forall x)(\forall y)(\forall z)\left(((x = y) \wedge (y = z)) \Rightarrow (x = z)\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((z \cdot x) = (z \cdot y))\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((x \cdot z) = (y \cdot z))\right)$$

Example 3.16. Theory of posets:

Let us first define an appropriate $\mathcal{L}(\Pi, \Omega)$*, i.e. sets of predicates and functionals we will use (please see also Example 1.15):*

$\Pi = \{=, \leq\}$*, where:*

the (equality) predicate '$=$*' has arity* 2

the predicate '$\leq$*' has arity* 2

$\Omega = \emptyset$.

Then, we may use the theory consisting of the following sentences (please see also Example 1.15):

$$(\forall x)(x \leq x)$$

$$(\forall x)(\forall y)(((x \leq y) \wedge (y \leq x)) \Rightarrow (x = y))$$

$$(\forall x)(\forall y)(\forall z)(((x \leq y) \wedge (y \leq z)) \Rightarrow (x \leq z))$$

$$(\forall x)(x = x)$$

$$(\forall x)(\forall y)\left((x = y) \Rightarrow (y = x)\right)$$

$$(\forall x)(\forall y)(\forall z)\left(((x = y) \wedge (y = z)) \Rightarrow (x = z)\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((z \leq x) \Rightarrow (z \leq y))\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((x \leq z) \Rightarrow (y \leq z))\right)$$

Example 3.17. Theory of graphs:

Let us now describe a theory for structures that might appear harder to define or that may not be commonly defined (in a mathematics course / degree, in the setting of first order predicate logic).

Here, we wish to define graphs. A graph consists of a set of vertices, as well as a (possibly empty) set of edges. An edge connects two distinct vertices of a graph, but it is not assumed that every two distinct vertices of a graph must have an edge connecting them.

We will include sentences that specify that there is no edge between a vertex and itself, and that an edge leads to a certain 'commutativity': if an edge connects a vertex x to a vertex y, then it also connects the vertex y to the vertex x (an alternative way of saying this is that edges are not 'directed').

Then, if we treat the vertices as the elements of the set underlying a given structure, we may describe the theory of graphs by using a single binary predicate to identify edges between vertices (and without requiring the use of any functionals).

With this in mind, let us define an appropriate $\mathcal{L}(\Pi, \Omega)$, i.e. sets of predicates and functionals we will use:

$\Pi = \{\sim\}$, *where:*

the predicate '$\sim$' has arity 2

$\Omega = \emptyset$.

Then, we may use the theory consisting of the following two sentences:

$(\forall x)(\neg(x \sim x))$

$(\forall x)(\forall y)((x \sim y) \Rightarrow (y \sim x))$

Note that there is no equality predicate in the general theory of graphs given above, so the theory does not include sentences related to the reflexivity, symmetry, transitivity and, perhaps, substitutivity properties of an equality predicate, which were present in the earlier theories of groups and posets.

We may define many other mathematical theories within the setting of first order predicate logic, and we may also add sentences to theories like the ones described above, in order to restrict the models we allow. For example, suppose that, to the theory of groups described in Example 3.15, we include a sentence specifying commutativity of the operation:

$$(\forall x)(\forall y)((x \cdot y) = (y \cdot x))$$

Then, the expanded theory will have as (normal) models only commutative groups.

We may also be able to modify our theories so that they model structures of only certain sizes, as indicated next:

Example 3.18. *Suppose that we wish to have a theory that models only groups of order* 3 *i.e. groups containing exactly* 3 *elements.*

Then, we may start with the theory of groups and introduce two extra 0*-ary functionals, which will represent the two non-identity elements of such groups. Then, we may add sentences to the earlier theory which will specify that the identity and the two new 'constants' we have added are all distinct, and that each element (in a group of order 3) must be either the identity, or one of the two new 'constants' (this will make sure that our theory models only groups of order* 3*, rather than groups of order at least* 3*).*

Let us give a complete description of such a theory, by first defining sets of predicates and functionals we will use:

$\Pi = \{=\}$*, where:*

the (equality) predicate '$=$*' has arity* 2

$\Omega = \{\cdot, E, A_1, A_2\}$*, where:*

the functional '$\cdot$*' has arity* 2

the functionals 'E*', '*A_1*', '*A_2*' each have arity* 0*.*

Then, we may use the theory consisting of the following sentences:

$$(\forall x)\left((E \cdot x = x) \wedge (x \cdot E = x)\right)$$

$$(\forall x)(\exists y)\left((x \cdot y = E) \wedge (y \cdot x = E)\right)$$

$$(\forall x)(\forall y)(\forall z)\left(((x \cdot y) \cdot z) = (x \cdot (y \cdot z))\right)$$

$$(\forall x)\left(x = x\right)$$

$$(\forall x)(\forall y)\left((x = y) \Rightarrow (y = x)\right)$$

$$(\forall x)(\forall y)(\forall z)\left(((x = y) \wedge (y = z)) \Rightarrow (x = z)\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((z \cdot x) = (z \cdot y))\right)$$

$$(\forall x)(\forall y)(\forall z)\left((x = y) \Rightarrow ((x \cdot z) = (y \cdot z))\right)$$

$$(\neg(E = A_1))$$

$$(\neg(E = A_2))$$

$$(\neg(A_1 = A_2))$$

$$(\forall x)\left((x = E) \vee (x = A_1) \vee (x = A_2)\right)$$

Note that, if the three sentences:

$$(\neg(E = A_1))$$

$$(\neg(E = A_2))$$

$$(\neg(A_1 = A_2))$$

were not present in the theory described above, we would, essentially, not be specifying that the 'constants' that appear are all distinct; this would correspond to a theory that models groups containing 1 or 2 or 3 elements, rather than groups containing exactly 3 elements only.

This gives a way of providing a theory that models only groups of order up to (and including) 3.

We may, similarly, write down a theory that models only groups of order at most n (for any, given, natural number n), e.g. a theory that has as models all groups of order at most 10, or a theory that has as models all groups of order at most 1000, or a theory that has as models all groups of order at most 1000000.

As we shall see later in this chapter, describing a (single) theory that has, as models, *all* finite groups (but that does not model any infinite groups), is a bit more problematic (e.g. see Theorem 3.33, and the discussion following it).

3.2 Syntactic aspects of first order predicate logic

Having now defined a notion of meaning for first order predicate logic, and having seen how we can form theories that have as models specific structures, such as groups, rings, fields, posets or graphs, we move on to describe the syntactic setting of first order predicate logic, which will allow us to be able to define and write proofs in the setting of first order predicate logic, as we did for propositional logic in chapter 2.

So, as in chapter 2, and section 2.2 in particular, let us start by defining a version of *proof* for first order predicate logic. We will do so by giving an extended list of axioms and rules of deduction, starting from the ones given in the setting of propositional logic, and including some new ones, which will allow us to deal with the presence of variables in our current setting.

As *axioms* for proofs in first order predicate logic, let us take the following:

Axiom 1 $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ for all formulae α, β

Axiom 2 $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$ for all formulae α, β, γ

Axiom 3 $((\neg\alpha) \Rightarrow (\neg\beta)) \Rightarrow (((\neg\alpha) \Rightarrow \beta) \Rightarrow \alpha)$ for all formulae α, β

Axiom 4 $((\forall x)\alpha) \Rightarrow \alpha[t/x]$

for any formula α, variable symbol x, and term t such that no free variable of t occurs bound in α

Axiom 5 $((\forall x)(\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow ((\forall x)\beta))$

for any variable symbol x and formula α such that α contains no free occurrences of x

Note 3.19. *As in chapter 2 (see Note 2.24), the above axioms are all instances of tautologies, in cases where the formulae involved are capable of being interpreted as 'true' or 'false' in a given, suitable, $\mathcal{L}(\Pi, \Omega)$-structure (e.g. when the formulae involved are suitably defined sentences).*

Let us interprete the given formulae in some suitable structure, U say:

For Axiom 4, suppose that $((\forall x)\alpha)_U = 1$. Then, for all u in U, $(\alpha[\bar{u}/x])_U = 1$, by the definition of an interpretation. It therefore follows that, for any closed term t, which is interpreted as an element of U (i.e. such that $t_U \in U$), $(\alpha[t/x])_U = (\alpha[\bar{t}_U/x])_U = 1$. Therefore $(((\forall x)\alpha) \Rightarrow \alpha[t/x])_U = 1$.

Note that, if $((\forall x)\alpha)_U = 0$, then, irrespective of $(\alpha[t/x])_U$, $(((\forall x)\alpha) \Rightarrow \alpha[t/x])_U = 1$ follows, by the definition of an interpretation.

As for Axiom 5, suppose that $((\forall x)(\alpha \Rightarrow \beta))_U = 1$. Then, for all u in U,

$$((\alpha \Rightarrow \beta)[\bar{u}/x])_U = ((\alpha[\bar{u}/x] \Rightarrow \beta[\bar{u}/x]))_U = 1$$

Now, by assumption, α contains no free occurrences of x, so that $\alpha[\bar{u}/x] = \alpha$ and therefore

$$((\alpha[\bar{u}/x] \Rightarrow \beta[\bar{u}/x]))_U = ((\alpha \Rightarrow \beta[\bar{u}/x]))_U$$

Therefore, $(\alpha \Rightarrow ((\forall x)\beta))_U = 1$, and $(((\forall x)(\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow ((\forall x)\beta)))_U = 1$.

Once again, note that, if $((\forall x)(\alpha \Rightarrow \beta))_U = 0$, then, irrespective of $(\alpha \Rightarrow ((\forall x)\beta))_U$, it follows that

$$(((\forall x)(\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow ((\forall x)\beta)))_U = 1$$

In addition, we shall use the following rules of deduction:

Modus ponens — From α and $\alpha \Rightarrow \beta$, we may deduce β (for all formulae α, β)

Generalisation — From α, we may deduce $(\forall x)\alpha$ (for any formula α and variable symbol x), if no free occurrence of x appears in the premises used to prove α

Note 3.20. *As mentioned earlier, the axioms and rule of deduction that are mentioned above, but which were not present in the case of propositional logic, (Axioms 4 and 5, and generalisation), are the main tools that will allow us to deal with sentences involving variables and the '$\forall$' symbol, in proofs given in the setting of first order predicate logic.*

As in section 2.2 (see Definition 2.25), we will now express a notion of proof, for first order predicate logic, using the above axioms and rules of deduction:

Definition 3.21. *Let S be a subset of formulae in a first order predicate language $\mathcal{L}(\Pi, \Omega)$ and α a formula in $\mathcal{L}(\Pi, \Omega)$. Then, a proof of α from S is a finite, ordered sequence of formulae (or lines), $t_1, t_2, \cdots, t_n$ say, such that t_n is the formula α, and, such that, for each formula t_i, $1 \leq i \leq n$:*

a) t_i *is an (occurrence of an) axiom, or*

b) t_i *is a formula in S (we call such a formula a* hypothesis*), or*

c) t_i *can be deduced by one of the two rules of deduction (modus ponens or generalisation) from earlier lines, so that either:*

(i) t_i *is a formula that can be deduced by modus ponens from two preceding formulae: t_i is some formula β, and, for some $j, k < i$, t_j is some formula α and t_k is the formula $\alpha \Rightarrow \beta$, or*

(ii) t_i *is a formula that can be deduced by generalisation: t_i is some formula $(\forall x)\alpha$, and, for some $j < i$, t_j is the formula α, such that, for $k < j$, no free occurrence of x appears in any formula t_k used to obtain the formula t_j.*

Let us now give the notion corresponding to entailment in our current, syntactic, setting:

Definition 3.22. *Consider a subset S of the set of formulae in a first order predicate language $\mathcal{L}(\Pi, \Omega)$, and a formula α in $\mathcal{L}(\Pi, \Omega)$. Then, we say that S* syntactically implies *or* proves *α, and write*

$$S \vdash \alpha$$

if there exists a proof of α from S (S being a set of hypotheses in this case).

Note 3.23. *The 'machinery' of proofs described above allows us to take as hypotheses general formulae and not only sentences (and thus also to give proofs of general formulae). However, when wishing to give proofs of statements in various mathematical theories, our hypotheses will be sentences, and, restricting to this case, as we do later in this chapter (and in particular in section 3.3), will allow us to interprete formulae present in proofs in terms of structures which model the original theories, and obtain a Completeness Theorem for first order predicate logic.*

As in the case of propositional logic, a theorem is a formula which we can prove without involving any hypotheses i.e. a formula which we can prove by taking $S = \emptyset$ above:

Definition 3.24. *If, within a first order predicate language, $\mathcal{L}(\Pi, \Omega)$ say, there exists a proof of a formula α that uses only occurrences of the five given axioms, modus ponens and generalisation, then we say that α is a* theorem*, i.e. $\alpha \in \mathcal{L}(\Pi, \Omega)$ is a theorem if $\emptyset \vdash \alpha$. In such a case, we may also write $\vdash \alpha$.*

In the remainder of this section, and this chapter in general, we will try to investigate the way in which many of the results proven in the setting of propositional logic transfer to the setting of first order predicate logic (results such as the Deduction Theorem and Completeness Theorem), and we will also try to illustrate some of the differences between the two settings.

Before we do so, let us give two relatively simple examples of proofs in first order predicate logic. We will be working within the 'theory of groups', the sentences of which we will take as hypotheses. However, the two proofs we give are of statements for which similar results hold more generally, as they essentially only involve sentences that describe properties of the equality predicate:

Below, we let T be the theory of groups (i.e. the set of sentences given in Example 3.15).

Let us first give a (simple) proof of $y = y$ from T (for a variable symbol y):

1.	$(\forall x)(x = x)$	Hypothesis (from T)
2.	$((\forall x)(x = x)) \Rightarrow (y = y)$	Axiom 4
3.	$y = y$	Modus ponens on lines 1, 2

Having written the lines of the above proof, we could have also applied generalisation to obtain (or 'go back to') the statement $(\forall y)(y = y)$. This would have been a valid use of generalisation, since no free instance of y is used (in any hypothesis used) to prove the 'non-generalised' statement $y = y$ above (e.g. we did not assign a specific closed term to y in order to prove $y = y$).

Our second example will hopefully be more instructive in understanding the restriction imposed when applying generalisation. Let us show that the implication $T \cup \{z = E\} \vdash E = z$ holds, i.e let us show that, for a variable symbol z, there is a proof of $E = z$, when assuming the sentences present in the theory of groups and the formula $z = E$:

1.	$(\forall x)(\forall y)\left((x = y) \Rightarrow (y = x)\right)$	Hypothesis (from T)
2.	$z = E$	Hypothesis
3.	$((\forall x)(\forall y)\left((x = y) \Rightarrow (y = x)\right)) \Rightarrow ((\forall y)\left((z = y) \Rightarrow (y = z)\right))$	Axiom 4
4.	$(\forall y)\left((z = y) \Rightarrow (y = z)\right)$	Modus ponens on lines 1, 3
5.	$((\forall y)\left((z = y) \Rightarrow (y = z)\right)) \Rightarrow ((z = E) \Rightarrow (E = z))$	Axiom 4
6.	$(z = E) \Rightarrow (E = z)$	Modus ponens on lines 4, 5
7.	$E = z$	Modus ponens on lines 2, 6

Having written down the lines of the proof, it seems that we ought not be able to apply generalisation in order to deduce that $(\forall z)(z = E)$. This is because, it seems that we have proven a statement about a 'specific' z, which we assumed (in our hypotheses) satisfies $z = E$, so that we cannot generalise the final result.

This is a kind of situation that the restriction in the use of generalisation, given earlier, is designed to avoid. It would be invalid to use generalisation to deduce $(\forall z)(z = E)$ in this case, because there is a free occurrence of the variable symbol z in one of the premises used to prove $E = z$; in particular, there is a free occurrence of z in line 2, and the formula $z = E$, which is then used to prove $E = z$ (by modus ponens) on the final line of the above proof.

Le us now show that a version of the Deduction Theorem for propositional logic (see Theorem 2.30) also holds for first order predicate logic (in fact, the proof given below is largely identical to the proof of Theorem 2.30):

Theorem 3.25 (Deduction Theorem)**.** *Let S be a set of formulae ($S \subset \mathcal{L}$), and α, β be formulae ($\alpha, \beta \in \mathcal{L}$). Then*

$$S \vdash (\alpha \Rightarrow \beta) \qquad \textit{if and only if} \qquad S \cup \{\alpha\} \vdash \beta$$

Proof. We will prove the two 'directions' of this theorem separately.

First, let us suppose that $S \vdash (\alpha \Rightarrow \beta)$, i.e. that there exists a sequence of formulae $t_1, \cdots, t_n$, such that each t_i is an instance of an axiom, or an element of S, or obtained by modus ponens on two previous lines, or by generalisation on a previous line, and such that the final formula, t_n, is $\alpha \Rightarrow \beta$.

Then, if we simply add $t_{n+1} : \alpha$, which is a hypothesis in $S \cup \{\alpha\}$, followed by $t_{n+2} : \beta$, which is obtained by modus ponens on the formulae t_n, t_{n+1}, we obtain a proof of β from $S \cup \{\alpha\}$, i.e. $S \cup \{\alpha\} \vdash \beta$, as required.

Now, suppose there exists a proof of β from $S \cup \{\alpha\}$, say a sequence of formulae $t_1, \cdots, t_m$, with $t_m : \beta$. We will modify this proof to obtain a proof of $\alpha \Rightarrow \beta$ from S, by trying to include in our proof the formula $\alpha \Rightarrow t_i$, for each formula t_i above.

For each t_i, $1 \leq i \leq m$, we have the following cases:

A) t_i is an (occurrence of an) axiom.

Then, since t_i is an axiom, we may still include t_i in our new proof. We may then use the following three formulae in order to obtain $\alpha \Rightarrow t_i$:

1.	t_i	Axiom
2.	$t_i \Rightarrow (\alpha \Rightarrow t_i)$	Axiom 1
3.	$\alpha \Rightarrow t_i$	Modus ponens on lines 1, 2

B) t_i is a formula in $S \cup \{\alpha\}$, i.e. $t_i \in S$ or t_i is α.

If $t_i \in S$, then it remains a hypothesis (since we are trying to construct a proof of $\alpha \Rightarrow \beta$ from S), so we may still include t_i in our new proof. Hence, we may proceed as in case A in order to obtain $\alpha \Rightarrow t_i$:

1.	t_i	Hypothesis
2.	$t_i \Rightarrow (\alpha \Rightarrow t_i)$	Axiom 1
3.	$\alpha \Rightarrow t_i$	Modus ponens on lines 1, 2

If t_i is α, we cannot use the same sequence of lines, because α is no longer a hypothesis, in the setting of $S \vdash (\alpha \Rightarrow \beta)$. However, there is another way of obtaining $\alpha \Rightarrow t_i$, i.e. $\alpha \Rightarrow \alpha$, without needing to include the formula α (by itself): we may simply write down a proof of the theorem $\alpha \Rightarrow \alpha$, which was proven earlier (see chapter 2, and Proposition 2.28; the same proof transfers directly to the setting of first order predicate logic).

C) Suppose that t_i is a formula that can be deduced by modus ponens from two preceding formulae.

So, suppose that, for some $j < i$, $k < i$ and $\gamma, \delta \in \mathcal{L}$, the formula t_j is γ, the formula t_k is $\gamma \Rightarrow \delta$, and t_i is the formula δ, obtained by modus ponens on lines t_j, t_k.

Then, since t_i follows t_j and t_k, we may, inductively, assume that we have already successfully replaced the lines t_j and t_k by $\alpha \Rightarrow t_j$ and $\alpha \Rightarrow t_k$ respectively.

So, we may now use:

- $\alpha \Rightarrow t_j$: $\alpha \Rightarrow \gamma$
- $\alpha \Rightarrow t_k$: $\alpha \Rightarrow (\gamma \Rightarrow \delta)$

Then, we may obtain the line $\alpha \Rightarrow t_i$: $\alpha \Rightarrow \delta$, by including the following sequence:

1.	$\alpha \Rightarrow \gamma$	
2.	$\alpha \Rightarrow (\gamma \Rightarrow \delta)$	
3.	$(\alpha \Rightarrow (\gamma \Rightarrow \delta)) \Rightarrow ((\alpha \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \delta))$	Axiom 2
4.	$(\alpha \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \delta)$	Modus ponens on lines 2, 3
5.	$\alpha \Rightarrow \delta$	Modus ponens on lines 1, 4

D) Suppose that t_i is a formula that can be deduced by generalisation from a preceding formula.

So, suppose that, for some $j < i$, variable symbol x and $\gamma \in \mathcal{L}$, the formula t_j is γ, and t_i is the formula $(\forall x)\gamma$ (where we are assuming that no free occurrence of x appears in any of the premises used to prove γ).

Then, since t_i follows t_j, we may, inductively, assume that we have already successfully replaced t_j by $\alpha \Rightarrow t_j$.

There are now two separate cases to consider, depending on whether or not α contains any free occurrences of x.

If α contains no free occurrences of x, then neither does $\alpha \Rightarrow \gamma$, or any of $\alpha \Rightarrow t_k$, where t_k is one of the earlier lines used to prove t_j (by assumption, our premises do not contain free occurrences of x). So, we may now use:

- $\alpha \Rightarrow t_j \ : \quad \alpha \Rightarrow \gamma$
- $\alpha \Rightarrow t_k$ (for any t_k used to prove t_j)

Then, we may obtain the line $\alpha \Rightarrow t_i \ : \ \alpha \Rightarrow ((\forall x)\gamma)$, by including the following sequence of lines:

1.	$\alpha \Rightarrow \gamma$	
2.	$(\forall x)(\alpha \Rightarrow \gamma)$	Generalisation
3.	$((\forall x)(\alpha \Rightarrow \gamma)) \Rightarrow (\alpha \Rightarrow ((\forall x)\gamma))$	Axiom 5
4.	$\alpha \Rightarrow ((\forall x)\gamma)$	Modus ponens on lines 2, 3

The generalisation used to give the second line above is valid by the assumption that no free occurrences of x occur in $\alpha \Rightarrow \gamma$ or $\alpha \Rightarrow t_k$, for any t_k used to prove $t_j \ : \ \gamma$, while the use of Axiom 5 used to give the third line is, similarly, valid because α is assumed to contain no free occurrences of the variable x.

Now, suppose that α contains a free occurrence of the variable symbol x. Then, since we are assuming that no free occurrence of x appears in any of the premises used to prove γ in the original proof (with set of hypotheses $S \cup \{\alpha\}$), we deduce that α cannot appear in the premises used to prove γ (in the original proof). Hence, there exists a proof of $t_j \ : \ \gamma$ and of $t_i \ : \ (\forall x)\gamma$, using just hypotheses from S: $S \vdash (\forall x)\gamma$.

Then, we may obtain the line $\alpha \Rightarrow t_i \ : \ \alpha \Rightarrow ((\forall x)\gamma)$ from S, by including, to a proof of $(\forall x)\gamma$ from S, the following sequence of lines:

1.	$(\forall x)\gamma$	
2.	$((\forall x)\gamma) \Rightarrow (\alpha \Rightarrow ((\forall x)\gamma))$	Axiom 1
3.	$\alpha \Rightarrow ((\forall x)\gamma)$	Modus ponens on lines 1, 2

So, we can write down a proof of $\alpha \Rightarrow ((\forall x)\gamma)$ from S.

Hence we have shown that, given a proof of β from $S \cup \{\alpha\}$, it is possible to 'replace' each formula t_i in this proof by the formula $\alpha \Rightarrow t_i$, without using α as a hypothesis. Therefore, it is possible to construct a proof of $\alpha \Rightarrow \beta$ from S, as required: the final formula, $t_m \ : \ \beta$, of the proof of $S \cup \{\alpha\} \vdash \beta$ may be replaced by the formula $\alpha \Rightarrow t_m : \ \alpha \Rightarrow \beta$, to give a proof of $S \vdash (\alpha \Rightarrow \beta)$.

□

3.3 Completeness theorem for first order predicate logic

We now aim to show that the settings of semantic and syntactic implication coincide. When trying to show that these concepts coincide, we will deal with 'objects' that can be interpreted as true or false in a given setting, something which holds for all propositions in $\mathcal{L}_0$ (e.g. please see chapter 2) but not for all formulae in $\mathcal{L}$. So, we will restrict attention to sentences and describe how, for any set of sentences, S, in $\mathcal{L}$, and any sentence, α, in $\mathcal{L}$:

$$S \models \alpha \qquad \text{if and only if} \qquad S \vdash \alpha$$

This result is the *Completeness Theorem for first order predicate logic*.

In this section, we will give a description of the proof of (the two directions of) the Completeness Theorem.

We start with one direction, and the Soundness Theorem for first order predicate logic:

Theorem 3.26 (Soundness Theorem)**.** *Let S be a set of sentences in $\mathcal{L}$ and α be a sentence in $\mathcal{L}$:*

$$\textit{If} \quad S \vdash \alpha \qquad \textit{then} \quad S \models \alpha$$

Proof. Consider a proof of α from S, consisting of the formulae $t_1, \cdots, t_n$ say. Let us show that, if we are given a model of S, i.e. a suitably defined structure U such that each sentence in S is interpreted as true in U (i.e. such that $s_U = 1$ for all s in S), then the sentence t_i is also true in U (i.e. $(t_i)_U = 1$), for each line t_i in the proof (i.e. for $1 \leq i \leq n$).

To this end, consider a model U of S. For each t_i, $1 \leq i \leq n$, we have the following possibilities:

A) The sentence t_i is an (occurrence of an) axiom. All our axioms are tautologies, so $(t_i)_U = 1$ (for any model U, in fact).

B) The sentence t_i is a hypothesis, i.e. it is a sentence in S. Then, $(t_i)_U = 1$, by assumption.

C) The sentence t_i is deduced by modus ponens from two preceding sentences.

So, suppose that, for some $j < i$, $k < i$ and sentences β, γ in $\mathcal{L}$, the sentence t_j is β, the sentence t_k is $\beta \Rightarrow \gamma$, and t_i is the sentence γ, obtained by modus ponens on lines t_j, t_k.

Then, since t_i follows t_j and t_k, we may, inductively, assume that the result we are trying to prove already holds for t_j and t_k, i.e. that $(t_j)_U = \beta_U = 1$ and $(t_k)_U = (\beta \Rightarrow \gamma)_U = 1$. This ensures that $\gamma_U = 1$, by definition of an interpretation (may be shown directly, or by contradiction: if $\gamma_U = 0$, then $(\beta \Rightarrow \gamma)_U = 0$ or $\beta_U = 0$, contradicting our inductive assumption). So, it must be the case that $(t_i)_U = \gamma_U = 1$, as required.

D) The sentence t_i is deduced by generalisation from a preceding sentence.

So, suppose that, for some $j < i$ and a sentence β in $\mathcal{L}$, the sentence t_j is β, and t_i is the sentence $(\forall x)\beta$, obtained by generalisation on t_j. Inductively, we may assume that $(t_j)_U = 1$, so that $\beta_U = 1$. Then, let us show that $((\forall x)\beta)_U = 1$ (remember that, by definition of generalisation, we may also assume that no free occurrences of x appear in any of the premises used to prove β).

Recall that $((\forall x)\beta)_U = 1$, if, for all u in U, $(\beta[\bar{u}/x])_U = 1$, where $\beta[\bar{u}/x]$ is the formula obtained by replacing each free occurrence of x in β by $\bar{u}$.

If (a free occurrence of) the variable x does not appear in β, then such 'substitutions' have no effect on β, i.e. $\beta[\bar{u}/x] = \beta$, so that the interpretation of $(\forall x)\beta$ is the same as the interpretation of β, i.e. $((\forall x)\beta)_U = (\beta)_U = 1$.

On the other hand, if (a free occurrence of) the variable x does appear in β, then, since no free occurrences of x appear in any of the premises used to prove β, the sentence β would continue to satisfy $(\beta)_U = 1$ even if we replaced each occurrence of x in the premises used to prove β by any closed term, e.g. by any constant functional $\bar{u}$. So, $(\beta[\bar{u}/x])_U = 1$ for each u in U, and, hence, $((\forall x)\beta)_U = 1$, as required.

Hence, we have shown that, if U is a model of S, i.e. if $s_U = 1$ for all s in S, then every line t_i, $1 \leq i \leq n$, in the proof of α from S, satisfies $(t_i)_U = 1$. In particular, $(t_n)_U = (\alpha)_U = 1$, so that $S \models \alpha$, as required.

□

Let us now turn our attention to the other direction of the Completeness Theorem: let us try to show how one might prove an Adequacy Theorem for first order predicate logic. As in chapter 2, this direction will be the one that seems to require most of the work, at least in the setting we have described here. We will not give a complete proof of the Adequacy Theorem, but will give a description of how it may be proven, along the lines of the proof of the Adequacy Theorem for propositional logic given in chapter 2.

Let us start by mentioning that the notion of consistency transfers readily to the case of first order predicate logic:

Definition 3.27. *Consider a set of sentences, S, in a first order predicate language $\mathcal{L}(\Pi, \Omega)$.*

The set S is consistent *if there exists no sentence α in $\mathcal{L}(\Pi, \Omega)$ such that $S \vdash \alpha$ and $S \vdash (\neg\alpha)$.*

If there exists a sentence α such that $S \vdash \alpha$ and $S \vdash (\neg\alpha)$, we say that the set S is inconsistent.

Then, as in chapter 2, the main result which leads to the Completeness Theorem is:

Theorem 3.28. *Let S be a set of sentences in a first order predicate language. Then, if S is consistent, S has a model, i.e. if S is consistent, there exists a structure U such that each sentence from S is interpreted as true in U ($s_U = 1$ for each s in S).*

Sketch of proof:

Our aim is to construct a suitable structure U, which will serve as a model for the set of sentences (or theory) S. A possible way of constructing such a structure is outlined in the steps given below:

1) We may put all closed terms together and create a model for them.

   For example, if, as in the case of groups, we have a 0-ary functional 'E', and a 2-ary functional '$\cdot$', we may form closed terms such as E, $E \cdot E$, $(E \cdot E) \cdot E$, etc.

   If, as in the case of rings, we have 0-ary functionals '0' and '1', and 2-ary functionals '$+$' and '$\cdot$', we may form closed terms such as 0, $0 + 0$, $0 \cdot 0$, $0 + 1$, $1 + 1$, $1 \cdot 1$, $(1 + 1) \cdot 0$.

2) We may 'apply' the sentences present in S to the closed terms.

   For example, consider a theory (e.g. the theory of groups), which contains a 2-ary equality predicate '$=$' and the sentence $(\forall x)\,((E \cdot x = x) \wedge (x \cdot E = x))$. Then, this allows us to prove sentences such as $E \cdot E = E$, which 'identifies' two of the closed terms from earlier (the closed terms $E \cdot E$ and E, in this case).

3) After having constructed the basic 'spine' of the structure, we may extend the interpretation of sentences in U to a complete, consistent set, by including α or $\neg\alpha$, for each closed term α (as in chapter 2, and the proof of Theorem 2.39).

4) The consistent set formed may require the introduction of what are referred to as 'witnesses'. For example, the sentence $(\exists x)\,((x \cdot (1 + 1) = 1) \wedge (x \cdot (1 + 1) = 1))$ will hold in the theory of fields, of characteristic not equal to 2 say; this sentence will therefore be present in the complete consistent theory constructed in step 3.

   Note that the closed term for x (the inverse of $1 + 1 = 2$) cannot be uniquely identified. For example, in the field with 5 elements, $\mathbb{F}_5$, we would 'choose' x to be $3 = (1+(1+1))$ in order to make the above sentence hold, while, in the field with 7 elements, $\mathbb{F}_7$, we would 'choose' x to be $4 = (1 + (1 + (1 + 1)))$ in order to make the above true.

   In any case, if we are to construct a model U for the theory expressed by S, then we would have to introduce an element in U which 'acts as the inverse of 2'. We may achieve this by extending our set of functionals to include a new 0-ary functional, c say (a witness), and extend our theory by adding a sentence of the form $(c \cdot (1 + 1) = 1) \wedge ((1 + 1) \cdot c = 1)$.

   We may similarly deal with other sentences present in our complete, consistent, theory, by introducing witnesses.

5) The theory formed at the end of step 4 may no longer be complete, so we may need to loop back to step 3 (and repeat).

The above theorem, which may be viewed as a form of the Completeness Theorem itself, leads to a proof of an Adequacy Theorem for first order predicate logic:

Theorem 3.29 (Adequacy Theorem)**.** *Let S be a set of sentences in a first order predicate language $\mathcal{L}(\Pi, \Omega)$ and let α be a sentence in $\mathcal{L}(\Pi, \Omega)$:*

$$\textit{If} \quad S \models \alpha \qquad \textit{then} \quad S \vdash \alpha$$

Then, by combining the Soundness and Adequacy Theorems for predicate first order logic, we may deduce the Completeness Theorem for first order predicate logic, a result analogous to the Completeness Theorem for propositional logic (Theorem 2.41):

Theorem 3.30 (Completeness Theorem)**.** *Let S be a set of sentences in a first order predicate language $\mathcal{L}(\Pi, \Omega)$ and let α be a sentence in $\mathcal{L}(\Pi, \Omega)$. Then:*

$$S \models \alpha \qquad \textit{if and only if} \qquad S \vdash \alpha$$

The Completeness Theorem essentially concludes our study of first order predicate logic. Like the corresponding result in chapter 2 (Theorem 2.41), it might be seen to be an inherently important result, since it shows the equivalence of the semantic and syntactic settings of first order predicate logic, as described in this chapter; it brings together the notions of 'truth' and 'provability'.

It is also useful as a bridge between these settings, in the sense that if one wishes to prove a certain result in one of the two settings we have seen, a direct demonstration might appear quite complicated or elusive, but, for certain results at least, using the Completeness Theorem to move from one setting to the other may yield a demonstration much more readily.

Let us see now use the Completeness Theorem to prove one form of the Compactness Theorem for first order predicate logic, before studying some hopefully interesting consequences of this type of result, when we apply it to mathematical theories, and in particular, when we apply it to a theory that tries to emulate the arithmetic of the natural numbers in the setting of first order predicate logic.

In proving Theorem 3.32, it will be useful to use another version of the Completeness Theorem for first order predicate logic given above, which may be viewed as a corollary of Theorem 3.30, or as an equivalent version of the theorem, stated in the 'language' of Theorem 3.28:

Theorem 3.31. *Let S be a set of sentences in a first order predicate language. Then, S has a model if and only if S is consistent, i.e. there exists a structure U such that each sentence from S is interpreted as true in U ($s_U = 1$ for each s in S) if and only if S is consistent.*

The theorem given below allows us to 'move' from the finite to the infinite setting, and is indicative of the kinds of questions we will be considering for the remainder of this chapter. It shows that, if there exists a model for each finite subset of a set of sentences, then there also exists a model for the theory consisting of the original set of sentences (which might be an infinite set):

Theorem 3.32 (Compactness Theorem)**.** *Let S be a (possibly infinite) set of sentences in a first order predicate language. If every finite subset of S has a model, then so does S.*

Proof. We may prove this by contradiction. Suppose that every finite subset of S has a model, but that S does not have a model.

Then, by the form of the Completeness Theorem given in Theorem 3.31, every finite subset of S is consistent, but S is not consistent, i.e. there exists a sentence α such that $S \vdash \alpha$ and $S \vdash (\neg\alpha)$.

However, since proofs are finite, each of the proofs of α and $\neg\alpha$ from S uses a finite number of hypotheses. So, it is the case that $S' \vdash \alpha$ and $S'' \vdash (\neg\alpha)$, for finite subsets S', S'' of S.

Hence, we may deduce that $S' \cup S'' \vdash \alpha$ and $S' \cup S'' \vdash (\neg\alpha)$, i.e. that the finite subset $S' \cup S''$ of S is inconsistent.

Therefore, by the form of the Completeness Theorem given in Theorem 3.31, the finite subset $S' \cup S''$ of S does not have a model, which contradicts the hypothesis that every finite subset of S has a model.

So, as required, S is consistent, and therefore has a model, by the Completeness Theorem for first order predicate logic (e.g. by Theorem 3.31). □

The following corollary is an important result in what follows. It shows that, if there is a first order predicate theory which has models of any finite size, then it must necessarily have an infinite model (here, a finite or infinite model is a model of finite or infinite size, respectively, i.e. a structure whose corresponding/constituent set contains finitely or infinitely many elements, respectively).

Theorem 3.33 (Upward Löwenheim-Skolem Theorem)**.** *Suppose that a first order predicate theory (set of sentences) T has arbitrarily large finite models (i.e. suppose that, for any $n \in \mathbb{N}$, there is a structure U of size greater than or equal to n, which is a model of T). Then, the theory T also has an infinite model.*

Proof. In order to prove this result, we will extend our language and theory, by including (countably) infinitely many functionals and sentences. This will force the extended theory to have an infinite model, and such a model will necessarily be a model of the original theory too.

Suppose that the original theory T, which has arbitrarily large finite models, is expressed in $\mathcal{L}(\Pi, \Omega)$, i.e. using predicates from the set Π and functionals from the set Ω.

Then, let us extend Ω by introducing infinitely many 0-ary functionals, c_1, c_2, $\cdots$ and set

$$\Omega' = \Omega \cup \{c_1, c_2, \cdots\}$$

i.e. $\Omega' = \Omega \cup \{c_i : i \in \mathbb{N}\}$

We may now treat the theory T as a theory expressed in $\mathcal{L}(\Pi, \Omega')$, and this will allow us to extend T to a new theory, where we ensure that the 0-ary functionals correspond to distinct elements of any structure we wish to consider. To this end, let us define

$$T' = T \cup \{\neg(c_1 = c_2), \neg(c_1 = c_3), \neg(c_1 = c_4), \cdots, \neg(c_2 = c_3), \neg(c_2 = c_4), \cdots\}$$

i.e. $T' = T \cup \{\neg(c_i = c_j) : i, j \in \mathbb{N}; i \neq j\}$

(Note that the equality predicate '$=$' is used in the extended theory T'; to this end, if such a predicate is not present in Π, we may extend Π by introducing such a binary predicate, and, if necessary, may extend the theory T' by including any relevant sentences related to the equality predicate.)

We may now use the Compactness Theorem to show that T' has a model. Consider any finite subset of T' , S say. Then, since S is finite, the sentences in S will contain only finitely many constants (0-ary functionals), so these sentences will be true for any model of the original theory T that contains at least as many elements (note that the only types of sentences that could be in S, but not in T, are those of the form '$\neg(c_i = c_j)$'). But, by assumption, T has some model of finite size greater than or equal to any given natural number, so we may deduce that S has a model.

Hence, every finite subset S of T' has a model. Therefore, by the Compactness Theorem (for first order predicate logic), the infinite theory T' (also) has a model.

It remains to show that this model will also be a model of the original theory, T. Since T is a subset of T', any model of T' is also a model of T (i.e. any structure in which every sentence of T' is true is necessarily also a structure in which every sentence of T is true, since the sentences in T are all sentences in T' too). So, the infinite model we have found for T' is also an infinite model for T. Therefore, T has an infinite model, as required. □

This result is relevant to some of the examples of theories given in this chapter.

In particular, by employing a 'finite version' of one of the arguments mentioned in the proof above, we were (in section 3.1) able to see how we may describe theories having as (normal) models groups of certain 'sizes', e.g groups of order 3 (see Example 3.18).

The Upward Löwenheim-Skolem Theorem, mentioned above, indicates that we cannot achieve a theory that will have as (normal) models all groups containing a finite number of elements, or even a theory having as (normal) models groups containing an arbitrarily large (finite) number of elements, without such a theory also having an infinite (normal) model.

Any (first order predicate) theory which has *arbitrarily large finite models* must also have an *infinite model*, so, any theory which, for example, has as models all finite groups, must necessarily have as a model a group of infinite size.

Note 3.34. *In this course, we have restricted ourselves to the consideration of (first order predicate) languages containing, at most,* countably *infinitely many predicates, functionals etc. Much of what we have proven may be translated directly to a wider setting, where our language is allowed to incorporate uncountably many predicates, functionals etc.*

In such a setting, using a very similar proof, we may deduce an 'uncountable version' of the Löwenheim-Skolem Theorem, which states the following:

Theorem 3.35. *Suppose that a first order predicate theory (set of sentences) T has a countably infinite model. Then, the theory T (also) has an uncountably infinite model.*

We will use this result it in the final part of this section, which follows.

We conclude this section (and chapter), by trying to express the natural numbers, $\mathbb{N}$, using the first order predicate language. We observe that an important property of the natural numbers that we must try to include in our language is that of *induction*, which we may phrase in the following way:

Suppose that there is a property which is true for a subset of the natural numbers. If the property is 'true' for the first natural number, and, if, whenever the property is 'true' for one number, it is 'true' for the next number, then the property must be 'true' for all natural numbers.

In a sense, the attempt to suitably express this sentence in the framework of the first order predicate language will lead us to a theory which has the natural numbers as a model.

For example, note that we must find a way of suitably expressing the notion of the number following a number, or of a 'next' number. We may achieve this by using a 1-ary functional, which, in the setting of the natural numbers, may be thought of as a function mapping $n \in \mathbb{N}$ to $n + 1 \in \mathbb{N}$. This is often referred to as the *successor function*, and will be denoted by the functional symbol s.

Also, we will need to use substitution in order to be able to check that a given sentence holds for any given natural number, by being able to substitute any 'number' into the formula. This is permitted in the framework of first order predicate logic, e.g. by the definition of an interpretation (see Definition 3.9) in the semantic setting and by the use of Axiom 4 in the syntactic setting.

We will also attempt to suitably define addition and multiplication in our setting, but we start by giving a basic theory, which expresses some of the underlying features of the natural numbers, $\mathbb{N} = \{1, 2, 3, \cdots\}$.

Let us first define an appropriate $\mathcal{L}(\Pi, \Omega)$, i.e. sets of predicates and functionals we will use:

$\Pi = \{=\}$, where:

the (equality) predicate '$=$' has arity 2

$\Omega = \{1, s\}$, where:

the functional '1' has arity 0
the functional 's' has arity 1.

Then, let us consider the theory consisting of the following sentences:

PA1. $(\forall x)\,(x = x)$

PA2. $(\forall x)(\forall y)\,((x = y) \Rightarrow (y = x))$

PA3. $(\forall x)(\forall y)(\forall z)\,(((x = y) \wedge (y = z)) \Rightarrow (x = z))$

PA4. $(\forall x)(\forall y)\,((x = y) \Rightarrow (s(x) = s(y)))$

PA5. $(\forall x)(\forall y)\,((s(x) = s(y)) \Rightarrow (x = y))$

PA6. $(\forall x)\,(\neg\,(s(x) = 1))$

PA7. $(\forall y_1) \cdots (\forall y_n)\,((p[1/x] \wedge ((\forall x)\,(p \Rightarrow p[s(x)/x]))) \Rightarrow (\forall x)p)$
where p is a formula containing free occurrences of the variable symbol x as well as free occurrences of the variable symbols $y_1, \cdots, y_n$

The above set of sentences describes one form of the theory of *(weak) Peano arithmetic*.

We can obtain a 'stronger' form of this theory, by trying to suitably define the operations of addition and multiplication in our setting.

In order to do so, it will be convenient to include 0 in our theory, so let us give a version of Peano arithmetic which models the set

$$\mathbb{N}_0 = \mathbb{N} \cup \{0\} = \{0, 1, 2, \cdots\}$$

We will also extend the set of functionals we will use.

So, let us define an appropriate $\mathcal{L}(\Pi, \Omega)$:

$\Pi = \{=\}$, where:

the (equality) predicate '$=$' has arity 2

$\Omega = \{+, \cdot, 0, s\}$, where:

the functionals '$+$', '$\cdot$' each have arity 2

the functional '0' has arity 0

the functional 's' has arity 1.

Then, let us consider the theory consisting of the following sentences:

PA1. $(\forall x)\,(x = x)$

PA2. $(\forall x)(\forall y)\,((x = y) \Rightarrow (y = x))$

PA3. $(\forall x)(\forall y)(\forall z)\,(((x = y) \wedge (y = z)) \Rightarrow (x = z))$

PA4. $(\forall x)(\forall y)\,((x = y) \Rightarrow (s(x) = s(y)))$

PA5. $(\forall x)(\forall y)\,((s(x) = s(y)) \Rightarrow (x = y))$

PA6. $(\forall x)\,(\neg\,(s(x) = 0))$

PA7. $(\forall y_1) \cdots (\forall y_n)\,((p[0/x] \wedge ((\forall x)\,(p \Rightarrow p[s(x)/x]))) \Rightarrow (\forall x)p)$

where p is a formula containing free occurrences of the variable symbol x as well as free occurrences of the variable symbols $y_1, \cdots, y_n$

PA8. $(\forall x)\,(x + 0 = x)$

PA9. $(\forall x)(\forall y)\,(x + s(y) = s(x + y))$

PA10. $(\forall x)\,(x \cdot 0 = 0)$

PA11. $(\forall x)(\forall y)\,(x \cdot s(y) = (x \cdot y) + x))$

The theory described by this set of sentences is often referred to as *strong Peano arithmetic*.

Note 3.36. *The sentence PA7 is intended to describe induction. It says that to show that a formula, p say, holds for any natural number, i.e. that $(\forall x)p$ is true, we may show that it holds for $x = 0$, i.e. that $p[0/x]$ is true, and that, if the formula holds for any x, then it also holds for the successor of x, $x + 1$, i.e. that $(\forall x)\,(p \Rightarrow p[s(x)/x])$ is true.*

The presence of the other variable symbols, $y_1, \cdots, y_n$, in PA7 allows us to prove general results by induction, concerning sentences which have more than one bound variable. For example, it allows us to 'work with' the sentence:

$$(\forall a)(\forall b)\,(a + b = b + a)$$

We may arrive at a proof of the above sentence by using the induction outlined above, in PA7, twice. For example, we may first 'fix' a and use induction on b to prove the result for all b, before using induction on a to prove the result for all a as well.

In particular, if, in PA7, we set p to be $a + b = b + a$, y_1 to be a and x to be b, we may use this particular instance of PA7 to deduce $(\forall b)\,((a + b) = (b + a))$ (by induction on b), before setting p to be $(\forall b)\,((a + b) = (b + a))$ and x to be a, in PA7, and using this instance of PA7 to deduce $(\forall a)(\forall b)\,((a + b) = (b + a))$ (by induction on a), as required.

Note 3.37. *There are many different ways of describing Peano arithmetic. The forms of Peano arithmetic given above could be expanded to include more of the sentences we would expect to be satisfied for the natural numbers. For example, we have not included sentences which describe the commutativity of addition and multiplication, or the substitutivity of the equality predicate 'with respect to' addition and multiplication:*

$$(\forall x)(\forall y)\,(x + y = y + x)$$

$$(\forall x)(\forall y)\,(x \cdot y = y \cdot x)$$

$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((z + x) = (z + y)))$$

$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((x + z) = (y + z)))$$

$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((z \cdot x) = (z \cdot y)))$$

$$(\forall x)(\forall y)(\forall z)\,((x = y) \Rightarrow ((x \cdot z) = (y \cdot z)))$$

The above sentences, as well as many other sentences expressing properties of $\mathbb{N}$ or $\mathbb{N}_0$, can be proven from the theory of (strong) Peano arithmetic given earlier.

The theory of Peano arithmetic was historically an important theory, which 'encapsulated' the concept of the natural numbers, and is defined with the view that it models precisely the natural numbers. However, our notion of the natural numbers does not provide the only (normal) model of this

theory, as can be seen by using the more general, 'uncountable', version of the Upward Löwenheim-Skolem Theorem given earlier (see Theorem 3.35).

If the theory, of weak Peano arithmetic say, described above has an infinite (normal) model, e.g. $\mathbb{N}$, then it must also have an uncountably infinite (normal) model, a model which is therefore not isomorphic to $\mathbb{N}$ (since $\mathbb{N}$ is of countably infinite size). This might, at first, appear strange, since the theory was produced in the hope of modelling precisely the natural numbers.

One possible deficiency of the theory, in this regard, is that it cannot completely 'capture' the full strength of induction. We are working in the setting of first order predicate logic, with a countable number of variables, predicates, functionals etc. Therefore, if we consider a countably infinite (normal) model of the theory above (a model such as $\mathbb{N}$), there are (only) countably infinitely many formulae that PA7 may apply to (there are only countably infinitely many formulae that we may substitute for 'p' in the form of PA7 given above).

The fact that our theory, by virtue of the setting we are in, can only cope with countably many formulae means that we can only apply the form of induction defined by PA7 to countably many formulae, i.e. the 'induction' defined by PA7 can only apply to countably many properties of $\mathbb{N}$.

In 'real induction', we should be able to start with any property and check whether or not it holds for all the numbers in $\mathbb{N}$. Each subset of $\mathbb{N}$ corresponds (not uniquely perhaps) to a property of the natural numbers that we might like to 'test' using induction.

This means that there are uncountably many properties that we may wish to try 'applying induction to', since there are uncountably many subsets of $\mathbb{N}$.

So, the fact that our first order predicate theory can only deal with a countable infinity of formulae means that we cannot express, and therefore test by induction, every possible property (or subset) of $\mathbb{N}$, within this 'countable framework'. Therefore, the sentence PA7 does not completely describe the 'real induction' of $\mathbb{N}$. In this sense, it might not be a surprise that the first order theory of weak Peano arithmetic, described above, may have (normal) models that are not isomorphic to $\mathbb{N}$.

Even though the theory of weak Peano arithmetic, which was described above, does not uniquely identify $\mathbb{N}$, it does have $\mathbb{N}$ as a (normal) model, so it still seems to provide a useful way to express $\mathbb{N}$ in a first order predicate language.

The setting of first order predicate logic has some other 'deficiencies'. One such deficiency may be related to what is known as a 'halting problem', which we shall study in chapter 4 (see section 4.3); this may be 'encoded' into $\mathbb{N}$ or $\mathbb{N}_0$, and may be used to demonstrate that there is no general decidability theorem for first order predicate logic, unlike in the case of propositional logic.

Chapter 4

Computability

In this chapter, we will try to describe the main notions associated to computable and recursive (partial) functions. This will allow us to define the halting problem, which is a basic component of some (version of) results concerning decidability.

4.1 Computable (partial) functions

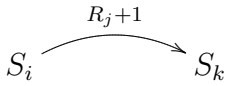
Let us start by describing an idealised machine that will provide the 'language' of this section, and this chapter.

Definition 4.1. *A* register machine *consists of the following:*

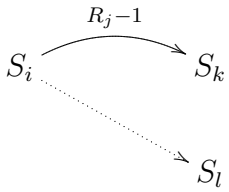
- *A sequence of registers $R_1, R_2, \cdots$, each capable of being assigned a nonnegative integer.*
- *A program, which consists of a specified finite number of states $S_0, S_1, \cdots, S_n$ such that:*
  - *Each of the states $S_1, \cdots, S_n$ is associated to an instruction, to be performed in that state.*
  - *S_1 is the initial state: its instruction is to be performed first.*
  - *S_0 is the terminal state: on reaching it, the program ends.*

An instruction in state S_i, $1 \leq i \leq n$, is of one of the following two forms:

- *An instruction which adds* 1 *to* R_j*, and then moves to state* S_k*.*

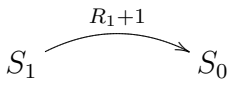


- *An instruction which:*
  - *If* $R_j > 0$*, subtracts* 1 *from* R_j*, and then moves to state* S_k*.*
  - *If* $R_j = 0$*, moves to state* S_l*.*

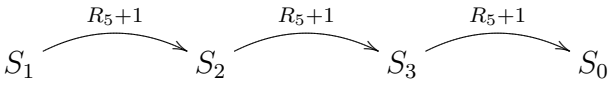


Let us use the diagrammatic notation introduced in the above definition to give some examples of register machines:

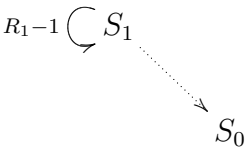
1) Below is a diagrammatic description of a register machine that adds 1 to the register R_1:



2) Below is a diagrammatic description of a register machine that adds 3 to the register R_5:



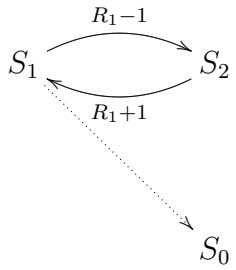
3) Below is a diagrammatic description of a register machine that 'clears' the register R_1, i.e that makes the register R_1 hold the value 0:



In general, we will refer to (suitably defined) functions that can be obtained using register machines as *computable functions* (below, we let $\mathbb{N}_0 = \{0, 1, 2, \cdots\}$ denote the set of nonnegative integers, as in chapter 3):

Definition 4.2. *A function* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *is a* computable function *if there exists a register machine that, started with* n_1 *in* R_1, $\cdots$, n_k *in* R_k, *and zero values in all remaining registers, ends (i.e. reaches state* S_0*) with* $f(n_1, \cdots, n_k)$ *in* R_1.

When studying the types of processes that we can understand or analyse using register machines, we stumble upon the possibility of defining a register machine that does not terminate. For example, the register machine with the following diagrammatic description does not terminate, if it is started with a positive integer in R_1 (a 'loop' is formed, where 1 is subtracted from R_1 and, then, 1 is added to R_1); if started with 0, the register machine terminates immediately:



If such processes are defined within a register machine, then the register machine might correspond to a map which is not defined for any or some of the 'input values'. For example, we might say that the register machine described above corresponds to a map from $\mathbb{N}_0$ to $\mathbb{N}_0$, which maps 0 to 0, and which is undefined for any positive integer.

So, register machines can also cope with such maps, which may be undefined at some points of their 'domain' (and which are therefore only functions if we restrict their 'domain'). Such maps are also known as *partial functions*:

Definition 4.3. *A* partial function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *is a map whose restriction to some subset,* A *say, of its domain,* $\mathbb{N}_0^k$*, is a function, i.e.* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *is a partial function if* $f|_A : A \to \mathbb{N}_0$ *is a function, for some* $A \subset \mathbb{N}_0^k$.

For example, the following are partial functions:

1) The map $f : \mathbb{N}_0 \to \mathbb{N}_0$, $f(n) = n/2$ is a partial function, which is only defined for (non-negative) even numbers (so that, in the notation of Definition 4.3, $A = \{0, 2, 4, \cdots\}$, in this case).

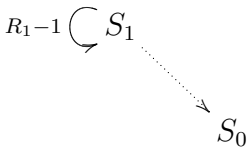
2) The map $f : \mathbb{N}_0 \to \mathbb{N}_0$, $f(n) = \sqrt{n}$ is a partial function.

Then, as mentioned above, it is possible for partial functions to be computable:

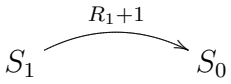
Definition 4.4. *A partial function* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *is a* computable partial function *if there exists a register machine that, started with* n_1 *in* R_1*,* $\cdots$*,* n_k *in* R_k*, and zero values in all remaining registers, ends (i.e. reaches state* S_0*) with* $f(n_1, \cdots, n_k)$ *in* R_1 *if* $f(n_1, \cdots, n_k)$ *is defined, and does not terminate if* $f(n_1, \cdots, n_k)$ *is undefined.*

Let us now return to the setting of functions, defined on the whole of their 'domain', and give some examples of computable functions:

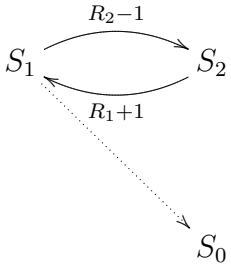
1) The (clearing) function $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $f(n) = 0$ is computable (as shown earlier):



2) The (successor) function $f : \mathbb{N}_0 \to \mathbb{N}_0$, $f(n) = n + 1$ is computable (as shown earlier):



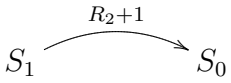
3) The (addition) function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$, $f(m, n) = m + n$ is computable:



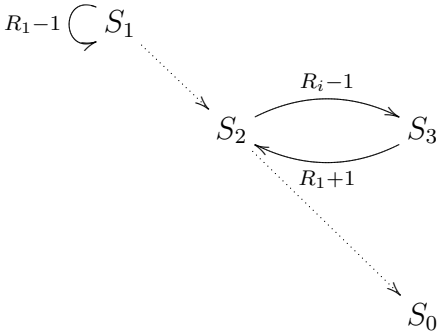
4) The (projection) function $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $f(n_1, \cdots, n_k) = n_i$, for some (given) i, $1 \leq i \leq k$, is computable:

   If $i = 1$, then the register R_1 already contains the final answer, so we do not need to do anything; we simply move to state S_0. However, since an instruction that will move to state S_0 (or any state) is only valid if it is associated with an 'operation' on a register, we are required to have an instruction associated to this move in order to 'validly' terminate the program.

So, we may, for example, use an instruction that adds 1 to any register except R_1, e.g. R_2:



If $i \neq 1$, then we may obtain the required projection function by, for example, first 'clearing' the register R_1 (using the function given in example 1 above), and then adding R_i to R_1 (using a function similar to the one given in the third example above). Combining these functions ensures that the register machine terminates with n_i in R_1, as required:



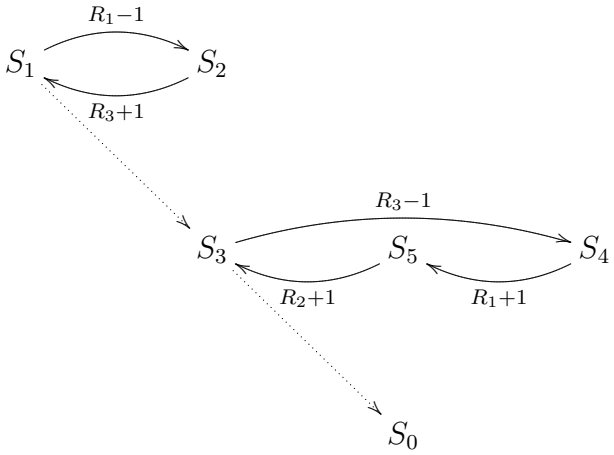
We can use these types of computable functions to generate many other computable functions (in fact, to generate all computable functions).

Before we show some of the different ways of using computable functions to obtain other computable functions, let us give a description of a register machine which 'copies' a nonnegative integer assigned to one register to another register. This type of operation is generally useful, as we shall see in the proofs we give for the results that follow.

Suppose, for example, that we wish to copy the number assigned to register R_1 to register R_2 (so that, at the end of the process, the original number is present in both R_1 and R_2). One way of achieving this result is by using a third register, R_3 say.

Let us assume that R_2 and R_3 both hold zero values at the beginning of the process (in any case, as described above, we may 'clear' both registers).

Then, to copy the value of R_1 to R_2, we may first 'move' the value to R_3, and then move the value from R_3 to both R_1 and R_2, as described diagrammatically below:



At the end of this process, registers R_1 and R_2 will both be assigned the nonnegative integer originally assigned to R_1, while register R_3 will be assigned the value 0. So, the overall effect will be a 'copy' of R_1 to R_2.

We now return to ways of obtaining computable functions from computable functions. One available tool at our disposal is composition. For example, in the construction of the register machine for the general projection function, given above, we essentially composed two of the earlier computable functions ('clearing' and addition) in order to obtain (part of) another computable function (projection). This holds more generally, as we will show in Theorem 4.6.

Note 4.5. *In the proof of the result given below, as well as in the (two) proofs that follow, apart from having an idea of how to generally construct a register machine that achieves the desired result, we must also take some care to ensure that the process we use does not inadvertently change the values of registers we might wish to use later on.*

To this end, we will often choose to move the integers assigned to some registers to some other 'far away registers where we can safely 'store' these values, so that they may be used later, if required. For example, let us suppose that, at the end of an operation, we would like the 'answer' to appear in register R_2, say, but there happens to be a nonnegative integer (that we would like to use later) already present in R_2; in such a case, we may first copy or move the value assigned to R_2 to some other register.

In general, in the proofs that follow, we will assume that we can clear any register, as well as move or copy values from one register to any other register. These are examples of operations for which we have already described suitable register machines.

For example, a computable program always 'returns' the final answer to register R_1, but we may move this answer to any other register, so we may assume that, in the setting of register machines, we can validly use the following kind of operation:

"compute the function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ on the input $(n_1, \cdots, n_k)$, and write the answer in register R_i"

where R_i can be any register (not necessarily R_1), and where the input $(n_1, \cdots, n_k)$ does not necessarily 'start off' in registers $R_1, \cdots, R_k$ (i.e. it is not necessarily the case that, at the beginning of the process, n_1 is in R_1, $\cdots$, n_k is in R_k).

Theorem 4.6. *The composition of computable (partial) functions yields a computable (partial) function; if $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is computable, and $g_1 : \mathbb{N}_0^l \to \mathbb{N}_0$, $\cdots$, $g_k : \mathbb{N}_0^l \to \mathbb{N}_0$ are computable, then $h : \mathbb{N}_0^l \to \mathbb{N}_0$ is computable where, for $(n_1, \cdots, n_l) \in \mathbb{N}_0^l$:*

$$h : (n_1, \cdots, n_l) \mapsto f(g_1(n_1, \cdots, n_l), \cdots, g_k(n_1, \cdots, n_l))$$

Proof. Let us show that h is computable. We will describe how to construct a register machine that starts with input $(n_1, \cdots, n_l) \in \mathbb{N}_0^l$, i.e. with n_1 in register R_1, $\cdots$, n_l in register R_l, and terminates with

$$h(n_1, \cdots, n_l) = f(g_1(n_1, \cdots, n_l), \cdots, g_k(n_1, \cdots, n_l))$$

in register R_1 if $h(n_1, \cdots, n_l)$ is defined, or does not terminate if $h(n_1, \cdots, n_l)$ is undefined.

We may construct such a register machine by implementing the following algorithm:

1) Copy $R_1, \cdots, R_l$ to $R_{N+1}, \cdots, R_{N+l}$ respectively, for some large enough N (say $N > k + l$).

2) For each $1 \leq i \leq k$, compute g_i on the input values $R_{N+1}, \cdots, R_{N+l}$ and write the answer in R_i (at the end of this process, $g_1(n_1, \cdots, n_l)$ will be in register R_1, $\cdots$, $g_k(n_1, \cdots, n_l)$ will be in register R_k).

3) Compute f on the input values $R_1, \cdots, R_k$ (i.e. compute $f(g_1(n_1, \cdots, n_l), \cdots, g_k(n_1, \cdots, n_l))$) and write the answer in R_1.

Note that, after carrying out this process (starting with n_1 in register R_1, $\cdots$, n_l in register R_l), the above shows that we will end up with $h(n_1, \cdots, n_l) = f(g_1(n_1, \cdots, n_l), \cdots, g_k(n_1, \cdots, n_l))$ in register R_1, if this is defined.

If any of $g_1(n_1, \cdots, n_l)$, $\cdots$, $g_k(n_1, \cdots, n_l)$ or $f(g_1(n_1, \cdots, n_l), \cdots, g_k(n_1, \cdots, n_l))$ is not defined (i.e. if $h(n_1, \cdots, n_l)$ is not defined), then the process will not terminate, as required.

□

We now describe a way of using addition (for which a relevant register machine was given above) to define multiplication.

Let us define a function $h : \mathbb{N}_0^2 \to \mathbb{N}_0$, such that $h(m, n) = mn$. We may define this recursively, using addition. We begin by assuming/noting that $h(m, 0) = 0$ for any input value $m \in \mathbb{N}_0$, If we then want to 'reach' $h(m, n) = mn$, we may add n copies of m to our starting value, $h(m, 0) = 0$. So, we may *recursively* define the multiplication function h, using the following rules:

$$h(m, 0) = 0 \quad , \quad h(m, k + 1) = h(m, k) + m \quad \text{(for } k \in \mathbb{N}_0\text{)}$$

Then, for example, to calculate $h(2, 3) = 6$, we may use a recursive process:

$$h(2, 0) = 0$$

$$h(2, 1) = h(2, 0) + 2 = 0 + 2 = 2$$

$$h(2, 2) = h(2, 1) + 2 = 2 + 2 = 4$$

$$h(2, 3) = h(2, 2) + 2 = 4 + 2 = 6$$

This is a particularly simple recursion, in the sense that each step of the recursion involves the same process: we 'add m to the answer from the previous step'.

In general, the operation performed at each stage may depend on the stage we are considering. For example, let us define a factorial function $h : \mathbb{N}_0 \to \mathbb{N}_0$, i.e. a function h such that $h(n) = n!$. We may define this recursively, using multiplication, but we will be multiplying by a different number at each stage. We begin by assuming/noting that $h(0) = 1$. If we then wish to reach $h(n) = n!$, we may multiply the result from the previous step by the 'next' number. So, we may *recursively* define the factorial function h, using the following rules:

$$h(0) = 1 \quad , \quad h(k + 1) = (k + 1)h(k)$$

Therefore, in general recursion, the recursive step may depend on the input from the previous step and on the step we find ourselves in. Let us now show that, if this form of recursion, which will be referred to as *primitive recursion*, is applied to computable (partial) functions (as in the examples above), then the resulting function is (also) a computable (partial) function:

Theorem 4.7. *Applying* primitive recursion *to computable (partial) functions leads to a computable (partial) function; if $f : \mathbb{N}_0^k \to \mathbb{N}_0$ and $g : \mathbb{N}_0^{k+2} \to \mathbb{N}_0$ are computable, then $h : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ is computable where, for $n_1, \cdots, n_k \in \mathbb{N}_0$ and $n \in \mathbb{N}_0$, h is defined recursively via:*

$$h(n_1, \cdots, n_k, 0) = f(n_1, \cdots, n_k) \quad , \quad h(n_1, \cdots, n_k, n+1) = g\left(n_1, \cdots, n_k, n, h(n_1, \cdots, n_k, n)\right)$$

Note 4.8. *Note that the statement of the theorem refers to (partial) functions that are more general than the ones considered in the examples above, even though they rely on the same principle. For example, we are allowing the original value of the function to depend on the input, unlike, say, in the case of multiplication, when the initial value is always zero ('m times 0 is equal to 0, whatever value of m we choose').*

Proof. Let us show that h is computable. We will describe a register machine that starts with input $(n_1, \cdots, n_k, n) \in \mathbb{N}_0^{k+1}$, i.e. with n_1 in register R_1, $\cdots$, n_k in register R_k, n in register R_{k+1}, and terminates with $h(n_1, \cdots, n_k, n)$ in register R_1 if $h(n_1, \cdots, n_k, n)$ is defined, or does not terminate if $h(n_1, \cdots, n_k, n)$ is undefined.

We may construct such a register machine by implementing the following algorithm:

1) Copy $R_1, \cdots, R_{k+1}$ to $R_{N+1}, \cdots, R_{N+k+1}$ respectively, for some large enough N (say $N > k + 2$).
2) Compute f on the input values $R_1, \cdots, R_{k+1}$ and write the answer in R_1.
3) Subtract 1 from R_{N+k+1}; if the value of the register R_{N+k+1} is 0, then terminate the program (go to state S_0). (This indicates that the program will terminate if and when the last stage of the recursion is reached.)
4) Copy R_1 to R_{k+2}.
5) Copy $R_{N+1}, \cdots, R_{N+k}$ to $R_1, \cdots, R_k$ respectively.
6) Copy R_{N+k+2} to R_{k+1} (note that, at the start, the register R_{k+1} holds the value 0).
7) Add 1 to register R_{N+k+2}. (This register holds the counter that is used as the input value 'n' in the function g, and which serves as a counter for the number of 'recursive steps' for the given computation; at the start of the process, the register R_{N+k+2} holds the value 0).
8) Compute g on the input values $R_1, \cdots, R_{k+2}$ and write the answer in R_1.
9) Return to step 3 above.

Note that, after carrying out this process (starting with n_1 in register R_1, $\cdots$, n_k in register R_k, n in register R_{k+1}), the above shows that we will end up with $h(n_1, \cdots, n_k, n)$ in register R_1, if $h(n_1, \cdots, n_k, n)$ is defined.

If any of $f(n_1, \cdots, n_k)$ or $g\left(n_1, \cdots, n_k, m, h(n_1, \cdots, n_k, m)\right)$ (for $m \leq n$) is not defined (i.e. if $h(n_1, \cdots, n_k, n)$ is not defined), then the process will not terminate, as required.

□

We will also show that a type of 'minimalisation' of a function leads to something computable. Suppose that we consider a (partial) function $f : \mathbb{N}_0 \to \mathbb{N}_0$, which takes as input a nonnegative integer, and returns as output a nonnegative integer. Then, minimalisation is the process which allows us to determine the smallest nonnegative integer input which is mapped to 0 by the function f. To be precise, it allows us to find $n \in \mathbb{N}_0$ such that $f(n) = 0$ and, for any $m < n$, $f(m)$ is defined and $f(m) > 0$ (if such an n exists).

We may encode this process in a minimalisation (partial) function, which will return this value of n or which will be undefined if no such minimalising value of n exists. Furthermore, we may extend this process to a minimalisation on a function with (possibly) more than one input, e.g. $f : \mathbb{N}_0^k \to \mathbb{N}_0$ say, where minimalisation is applied to one of the k inputs of such a function.

Let us show that, if minimalisation is applied to a computable (partial) function, then the resulting function is a computable (possibly partial) function:

Theorem 4.9. *Applying minimalisation to a computable (partial) function leads to a computable (possibly partial) function; if $g : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ is computable, then $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is computable where, for $n_1, \cdots, n_k \in \mathbb{N}_0$ and $n, m \in \mathbb{N}_0$, f is defined via:*

$$f(n_1, \cdots, n_k) = n \quad , \quad \textit{if} \quad g(n_1, \cdots, n_k, n) = 0 \quad \textit{and} \quad g(n_1, \cdots, n_k, m) > 0 \quad \textit{for} \quad m < n$$

while $f(n_1, \cdots, n_k)$ is undefined otherwise.

Proof. Let us show that f is computable. We will describe a register machine that starts with input $(n_1, \cdots, n_k) \in \mathbb{N}_0^k$, i.e. with n_1 in R_1, $\cdots$, n_k in R_k, and terminates with $f(n_1, \cdots, n_k)$ in R_1 if $f(n_1, \cdots, n_k)$ is defined, or does not terminate if $f(n_1, \cdots, n_k)$ is undefined.

We may construct such a register machine by implementing the following algorithm:

1) Copy $R_1, \cdots, R_k$ to $R_{N+1}, \cdots, R_{N+k}$ respectively, for some large enough N (say $N > k+1$).
2) Copy R_{N+k+1} (with starting value 0) to R_{k+1}.
3) Compute g on the input values $R_1, \cdots, R_{k+1}$ and write the answer in R_1.
4) Subtract 1 from R_1; if the value of the register R_1 is 0, then copy R_{N+k+1} to R_1 and terminate the program (go to state S_0). (This indicates that the program will terminate if and when we reach a stage where, for some $n \in \mathbb{N}_0$, $g(n_1, \cdots, n_k, n) = 0$.)
5) Add 1 to R_{N+k+1}.
6) Copy $R_{N+1}, \cdots, R_{N+k}$ to $R_1, \cdots, R_k$ respectively.
7) Return to step 2 above.

Note that, after carrying out this process (starting with n_1 in register R_1, $\cdots$, n_k in register R_k), the above shows that we will end up with $f(n_1, \cdots, n_k)$ in register R_1, if $f(n_1, \cdots, n_k)$ is defined.

If $f(n_1, \cdots, n_k)$ is undefined, i.e. if $g(n_1, \cdots, n_k, m)$ is not defined for some $m \leq n$ or if $g(n_1, \cdots, n_k, n)$ is not equal to zero for any $n \in \mathbb{N}_0$, then the process will not terminate, as required.

□

4.2 Recursive (partial) functions

Let us now introduce a notion related to that of a computable function. We will define and study functions referred to as recursive functions. They will not be defined with specific reference to register machines, but in an inductive way, which, nevertheless, begins with functions that are computable and uses processes that are 'well-behaved' with respect to computability, such as composition, primitive recursion and minimalisation.

With reference to the notions of truth and provability in earlier chapters, it may be instructive to think of computable functions as being semantic, in that they refer to functions that can be 'realised' using register machines, and to think of recursive functions as being syntactic, in that they are functions which can be theoretically constructed using a set of rules. Such an analogy might be particularly helpful later, when we see that the notions of computable and recursive functions are in agreement, much like those of true and provable propositions and (suitably defined) formulae (as shown in the Completeness Theorems of chapters 2 and 3 respectively).

Let us define the notion of a recursive (partial) function:

Definition 4.10. *The set of* recursive *partial functions is defined, inductively, as follows:*

1) *The (zero) function* $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $f(n_1, \cdots, n_k) = 0$ *is recursive.*
2) *The (successor) function* $f : \mathbb{N}_0 \to \mathbb{N}_0$, $f(n) = n + 1$ *is recursive.*
3) *The (projection) function* $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $f(n_1, \cdots, n_k) = n_i$ *for some* $1 \leq i \leq k$, *is recursive.*
4) *The composition of recursive (partial) functions leads to a recursive (partial) function.*
5) *Applying primitive recursion to recursive (partial) functions leads to a recursive (partial) function.*
6) *Applying minimalisation to a recursive (partial) function leads to a recursive (possibly partial) function.*

Our next goal is to describe the equivalence of the notions of computable and recursive functions, as mentioned above. In other words, we aim to see that:

A (partial) function is recursive if and only if it is computable.

In essence, we have already described and proven the main ingredients of the proof of one direction of this result:

Theorem 4.11. *If $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is a recursive (partial) function, then it is a computable (partial) function.*

Proof. We showed earlier that each of the three basic recursive functions is computable: the zero, successor and projection functions were given as examples of computable functions and (diagrammatic descriptions of) register machines were provided for these functions.

Furthermore, Theorems 4.6, 4.7, 4.9 show that applying the processes of composition, primitive recursion and minimalisation (respectively) to computable (partial) functions leads to computable (possibly partial) functions.

Therefore, any recursive (partial) function, i.e any (partial) function that may be constructed using the rules given in the above definition, must necessarily be a computable (partial) function. □

In retrospect, some of the examples of computable (partial) functions given earlier in this chapter were provided partly with the definition of recursive functions in mind, and with a view to leading to the proof of the theorem that has just been described.

In a similar way, we will now give a list of examples of recursive functions, which is partly chosen so as to allow us to describe a proof of the other direction of the theorem expressing the equivalence of recursive and computable (partial) functions. In order to see that every computable (partial) function is recursive, we will try to use natural numbers in order to 'encode' each state of a register machine at any given time, and then to eventually 'encode' a complete description of a register machine with specified 'input values', using a single natural number.

In order to use such an encoding to deduce that a computable (partial) function is recursive, we will require the ability to 'factorise' the encoded natural number.

In particular, for a fixed prime number p, we will require the use of a function that takes a natural number, n say, as input and returns, as output, the largest power of p that divides n. The list of examples of recursive functions given below allows us to 'build up' to such a function, for $p = 2$; a similar construction can be achieved for any other given prime number (in particular, later, we will assume that such a construction also works for $p = 3$).

With this in mind, let us give some examples of recursive functions:

1) The following (constant) function, for any $n \in \mathbb{N}_0$, is recursive:

$$f_1 : \mathbb{N}_0^k \to \mathbb{N}_0 \quad , \quad f_1(n_1, \cdots, n_k) = n$$

For example, we may obtain this function using the composition of n successor functions with a zero function, i.e. by applying a zero function and then adding 1 n times.

2) The following (addition) function is recursive:

$$f_2 : \mathbb{N}_0^2 \to \mathbb{N}_0 \quad , \quad f_2(m, n) = m + n$$

For example, we may obtain this function by applying primitive recursion:

- $f_2(m, 0) = m$, which we may obtain using a projection function, i.e. by projecting (to) the 'first input', m.
- $f_2(m, k+1) = f_2(m, k) + 1$, which we may obtain using the successor function.

3) The following (multiplication) function is recursive:

$$f_3 : \mathbb{N}_0^2 \to \mathbb{N}_0 \quad , \quad f_3(m, n) = mn$$

For example, we may obtain this function by applying primitive recursion:

- $f_3(m, 0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'second input', 0.
- $f_3(m, k+1) = f_3(m, k) + m$, which we may obtain using the addition function f_2, from example 2 above.

We may also write the recursive step in the notation of the addition function from example 2 above: $f_3(m, k+1) = f_2\left(f_3(m, k), m\right)$

4) The following (exponentiation) function is recursive:

$$f_4 : \mathbb{N}_0^2 \to \mathbb{N}_0 \quad , \quad f_4(m, n) = m^n$$

For example, we may obtain this function by applying primitive recursion:

- $f_4(m,0) = 1$, which we may obtain using the composition of a successor function with a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'second input', 0, and then adding 1; equivalently, we may obtain this using a constant function, as introduced in example 1 above.
- $f_4(m,k+1) = mf_4(m,k)$, which we may obtain using the multiplication function f_3, from example 3 above.

We may also write the recursive step in the notation of the multiplication function from example 3 above: $f_4(m,k+1) = f_3\left(m, f_4(m,k)\right)$

The following examples lead to a form of subtraction in the setting of recursive functions:

5) The following function is recursive:

$$f_5 : \mathbb{N}_0 \to \mathbb{N}_0 \quad , \quad f_5(n) = 0 \text{ if } n = 0 \ , \ f_5(n) = 1 \text{ if } n \neq 0$$

For example, we may obtain this function by applying primitive recursion:

- $f_5(0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'first input', 0.
- $f_5(k+1) = 1$, which we may obtain using the composition of a successor function with a zero function, i.e. by applying a zero function and then adding 1; equivalently, we may obtain this using a constant function, as introduced in example 1 above.

The function f_5 is an example of an *indicator function*. In general, for a subset A of $\mathbb{N}_0$, the indicator function $\chi_A : \mathbb{N}_0 \to \mathbb{N}_0$ is defined by $\chi_A(n) = 1$ if $n \in A$, $\chi_A(n) = 0$ if $n \notin A$. Using this notation, the function f_5, described above, corresponds to to the indicator function $\chi_{\{0\}^c}$, where $\{0\}^c$ denotes the set $\mathbb{N}_0 \backslash \{0\} = \{1, 2, 3, \cdots\}$.

6) The following function is recursive:

$$f_6 : \mathbb{N}_0 \to \mathbb{N}_0 \quad , \quad f_6(n) = 0 \text{ if } n = 0 \ , \ f_6(n) = n - 1 \text{ if } n \neq 0$$

For example, we may obtain this function by applying primitive recursion:

- $f_6(0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'first input', 0.
- $f_6(k+1) = f_6(k) + f_5(k)$, which we may obtain using the addition function f_2, from example 2 above, and the indicator function f_5, from example 5 above.

We may also write the recursive step in the notation of the addition function from example 2 above: $f_6(k+1) = f_2\left(f_6(k), f_5(k)\right)$

7) The following (subtraction) function is recursive:

$$f_7 : \mathbb{N}_0^2 \to \mathbb{N}_0 \quad , \quad f_7(m,n) = m - n \text{ if } m \geq n \ , \ f_7(m,n) = 0 \text{ if } m < n$$

For example, we may obtain this function by applying primitive recursion:

- $f_7(m,0) = m$, which we may obtain using a projection function, i.e. by projecting (to) the 'first input', m.
- $f_7(m,k+1) = f_6\left(f_7(m,k)\right)$, which we may obtain using the function f_6, from example 6 above.

We have now obtained a form of subtraction in the setting of recursive functions. The following examples lead to a recursive function that allows us to identify the largest power of 2 that divides a nonnegative integer (as required):

8) The following function is recursive:

$$f_8 : \mathbb{N}_0 \to \mathbb{N}_0 \quad , \quad f_8(n) = 1 \text{ if } n \text{ is odd} \ , \ f_8(n) = 0 \text{ if } n \text{ is even}$$

For example, we may obtain this function by applying primitive recursion:

- $f_8(0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'first input', 0.
- $f_8(k+1) = 1 - f_8(k)$, which we may obtain using the subtraction function f_7, from example 7 above.

We may also write the recursive step in the notation of the subtraction function from example 7 above: $f_8(k+1) = f_7\left(1, f_8(k)\right)$

Note that the function described here is, as in the case of the function f_5 described in example 5, an indicator function. Using the notation introduced earlier, the function f_8 described here corresponds to the indicator function $\chi_{\{\text{odd}\}}$, where '$\{\text{odd}\}$' denotes the set of odd numbers in $\mathbb{N}_0$, $\{1, 3, 5, \cdots\}$.

9) The following function is recursive:

$$f_9 : \mathbb{N}_0 \to \mathbb{N}_0 \quad , \quad f_9(n) = \left\lfloor \frac{n}{2} \right\rfloor$$

This is a 'floor' function, which, given input n, returns the largest nonnegative integer smaller than or equal to $\frac{n}{2}$, e.g. $f_9(0) = 0, f_9(1) = 0, f_9(2) = 1, f_9(3) = 1, f_9(4) = 2$ etc.

For example, we may obtain this function by applying primitive recursion:

- $f_9(0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'first input', 0.
- $f_9(k+1) = f_9(k) + f_8(k)$, which we may obtain using the addition function f_2, from example 2 above, and the indicator function f_8, from example 8 above.

We may also write the recursive step in the notation of the addition function from example 2 above: $f_9(k+1) = f_2\left(f_9(k), f_8(k)\right)$

10) The following function is recursive:

$$f_{10} : \mathbb{N}_0 \to \mathbb{N}_0 \quad , \quad f_{10}(n) = \frac{n}{2} \text{ if } n \text{ is even} \ , \ f_{10}(n) = 0 \text{ if } n \text{ is odd}$$

For example, we may obtain this function using the multiplication function f_3, from example 3 above, the characteristic function f_8, from example 8 above, and the 'floor' function f_9, from example 9 above:

$$f_{10}(n) = f_8(n+1)f_9(n)$$

We may also write this in the notation of the multiplication function from example 3 above: $f_{10}(n) = f_3\left(f_8(n+1), f_9(n)\right)$

11) The following function is recursive:

$$f_{11} : \mathbb{N}_0^2 \to \mathbb{N}_0 \quad , \quad f_{11}(n,m) = \frac{n}{2^m} \text{ if } 2^m \text{ divides } n \ , \ f_{11}(n,m) = 0 \text{ if } 2^m \text{ does not divide } n$$

For example, we may obtain this function by applying primitive recursion:

- $f_{11}(m,0) = m$, which we may obtain using a projection function, i.e. by projecting (to) the 'first input', m.
- $f_{11}(m,k+1) = f_{10}\left(f_{11}(m,k)\right)$, which we may obtain using the function f_{10}, from example 10 above.

12) The following function is recursive:

$f_{12} : \mathbb{N}_0^2 \to \mathbb{N}_0$, $f_{12}(m, n) = 1$ if 2^n divides m , $f_{12}(m, n) = 0$ if 2^n does not divide m

For example, we may obtain this function by applying composition, and using the indicator function f_5, from example 5 above, and the function f_{11}, from example 11 above:

$$f_{12}(n, m) = f_5(f_{11}(n, m))$$

13) The following function is recursive:

$f_{13} : \mathbb{N}_0^2 \to \mathbb{N}_0$, where $f_{13}(m, n)$ is the number of elements in $\{2^1, 2^2, \cdots, 2^n\}$ that divide m

For example, we may obtain this function by applying primitive recursion:

- $f_{13}(m, 0) = 0$, which we may obtain using a zero or projection function, i.e. by applying a zero function or by projecting (to) the 'second input', 0.
- $f_{13}(m, k + 1) = f_{13}(m, k) + f_{12}(m, k + 1)$, which we may obtain using the addition function f_2, from example 2 above, and the function f_{12}, from example 12 above.

We may also write the recursive step in the notation of the addition function from example 2 above: $f_{13}(m, k + 1) = f_2\left(f_{13}(m, k), f_{12}(m, k + 1)\right)$

We may now give our final example, which will return the largest power of 2 that divides a nonnegative integer.

14) The following function is recursive:

$f_{14} : \mathbb{N}_0 \to \mathbb{N}_0$, where $f_{14}(n)$ is the largest (exponent of a) power of 2 that divides n

For example, we may obtain this function by using the function f_{13}, from example 13 above (and keeping in mind that $2^n > n$):

$$f_{14}(n) = f_{13}(n, n)$$

We will denote the function given in the final example, f_{14}, by E_2, and will generally refer to the function $E_m : \mathbb{N}_0 \to \mathbb{N}_0$, for any natural number $m \geq 2$, as the function defined so that $E_m(n)$ is the largest power of m which divides n. We note that it may also be shown that E_m is a recursive function, for any $m > 2$ (above, it is shown that E_2 is recursive).

In particular, for a prime number p, E_p is recursive, and we will use (instances of) this fact in the (sketch of the) proof of the following result, which completes the demonstration of the correspondence between computable and recursive (partial) functions. We will show that any computable (partial) function is recursive, by encoding and decoding such a function using natural numbers, by implementing procedures that are recursive:

Theorem 4.12. *If $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is a computable (partial) function, then it is a recursive (partial) function.*

Sketch of proof:

We will use a natural number to represent the state of a register machine for f at any given time/stage.

Suppose that we wish to completely 'encode', in this way, what the register machine for f does 'at any given time/stage t' (i.e. after t steps; after t instructions have been performed), when started with input(s) $(n_1, \cdots, n_k) \in \mathbb{N}_0^k$.

We may do so by defining the following function

$$g : \mathbb{N}_0^{k+1} \to \mathbb{N}_0 \qquad , \qquad g(n_1, \cdots, n_k, t) \; = \; 2^q 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$$

where:

- q is the number corresponding to the state the machine is at time t
- $2, 3, \cdots, p_l$ are the first $l + 1$ prime numbers (so that $p_0 = 2$, $p_1 = 3$, etc.)
- r_i is the nonnegative integer in register R_i at time t.

Then, for any given $t \in \mathbb{N}_0$, we may treat the natural number $g(n_1, \cdots, n_k, t)$ as giving us a description of the register machine t steps after it is started with inputs $n_1, \cdots, n_k$ (i.e. after it is started with n_1 in register R_1, $\cdots$, n_k in register R_k).

In order to show that f is a recursive function, we will refer back to the examples of recursive functions constructed earlier.

We will first show that the function describing the register machine of f as above, i.e. the function g, is recursive (with respect to t). Suppose that we start with $g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$. This signifies that, at time t, the function is in state i (exponent of 2), and the registers $R_1, \cdots, R_l$ hold the values $r_1, \cdots, r_l$ respectively (all other registers hold zero values).

- If $t = 0$, then $r_i = n_i$ for $1 \leq i \leq k$ and $r_i = 0$ for $i > k$, so we obtain:

  $$g(n_1, \cdots, n_k, 0) = 2^1 3^{r_1} 5^{r_2} \cdots p_k^{r_k}$$

  and this is a recursive function (it involves exponentiation and multiplication, which were both shown to be recursive earlier; as well as the composition of such functions, which is allowed by the definition of a recursive function).

- Let us now show how we may obtain $g(n_1, \cdots, n_k, t+1)$ from $g(n_1, \cdots, n_k, t)$. We must use the 'information' present in $g(n_1, \cdots, n_k, t)$ to identify the state of the machine at this time, and then carry out the fixed instruction associated to that state.

  By construction of the function g, it is the exponent of 2 in $g(n_1, \cdots, n_k, t)$ that identifies which state we are in at time t. So, we may retrieve the state by determining the largest power of 2 dividing $g(n_1, \cdots, n_k, t)$. Such a function is an example of a recursive function, as shown in example 14 above. It corresponds to the function E_2, in the notation given above, so that

  $$E_2\left(g(n_1, \cdots, n_k, t)\right)$$

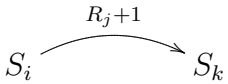
  identifies the 'current' state of the program (i.e. the state at time t).

  It remains to apply the instruction associated to the current state, in order to reach the next step and therefore reach $g(n_1, \cdots, n_k, t+1)$.

  By the definition of a register machine, one of two types of instructions is associated to each state of a program. Let us show that each of these instructions may be realised in the setting of recursive functions. We suppose that the current state is S_i, i.e. that

  $$E_2\left(g(n_1, \cdots, n_k, t)\right) = i$$

  - The instruction associated to state S_i adds 1 to R_j, and then moves to state S_k.

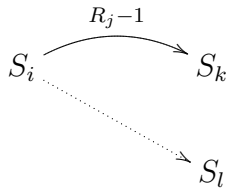
    Then, we are required to add 1 to register R_j, which corresponds to increasing the exponent of the prime p_j by 1, and to move to state S_k, which corresponds to changing the exponent of 2, from j to k.

    We may achieve the overall required result by therefore multiplying $g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$ by p_j (thus increasing the exponent of p_j by 1), and by also dividing

$g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$ by 2^i and then multiplying the result by 2^k (so that the exponent of 2 is k at the end of this process). The operations of multiplication and division involved here correspond to recursive functions, as shown earlier, so the overall process is recursive in this case, and leads to the required final answer:

$$g(n_1, \cdots, n_k, t+1) = 2^k 3^{r_1} \cdots p_j^{r_j+1} \cdots p_l^{r_l}$$

- The instruction associated to state S_i is as follows:
  - If $R_j > 0$, subtract 1 from R_j, and then move to state S_k.
  - If $R_j = 0$, move to state S_l.



We may achieve the overall result by first determining if the prime p_j divides $g(n_1, \cdots, n_k, t)$ (i.e. if $R_j > 0$). A recursive function may achieve this (in fact, the recursive function E_{p_j} would have a value of zero precisely when $R_j = 0$).
If $R_j > 0$, then we may divide $g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$ by p_j (thus decreasing the exponent of p_j by 1), and also divide $g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$ by 2^i and then multiply the result by 2^k (so that the exponent of 2 is k at the end of this process). The operations of multiplication and division involved here correspond to recursive functions, as shown earlier, so the overall process is recursive in this case, and leads to the required final answer:

$$g(n_1, \cdots, n_k, t+1) = 2^k 3^{r_1} \cdots p_j^{r_j-1} \cdots p_l^{r_l}$$

If $R_j = 0$, then we may divide $g(n_1, \cdots, n_k, t) = 2^i 3^{r_1} 5^{r_2} \cdots p_l^{r_l}$ by 2^i and then multiply the result by 2^l (so that the exponent of 2 is l at the end of this process). As with the case above, the operations involved here correspond to recursive functions, as shown earlier, so the overall process is recursive in this case, and leads to the required final answer:

$$g(n_1, \cdots, n_k, t+1) = 2^l 3^{r_1} \cdots p_j^{r_j} \cdots p_l^{r_l}$$

The above argument indicates that the 'encoding function' g is a recursive function.

Let us now complete the (sketch of the) proof of this result by showing that this leads us to conclude that the original function f is also recursive. We wish to construct a recursive function which simulates f, i.e. a recursive function from $\mathbb{N}_0^k$ to $\mathbb{N}_0$, which, when started with input $(n_1, \cdots, n_k)$ returns as output $f(n_1, \cdots, n_k)$, if this is defined.

Essentially, the function g constructed above already encodes this information. If and when the program reaches state 0, i.e. the value of t for which the exponent of 2 in $g(n_1, \cdots, n_k, t)$ becomes 0 (if it exists), is the time at which the program ends, and the value of register 1 at this time, i.e. the exponent of 3 for this value of t, is the output of the function, $f(n_1, \cdots, n_k)$.

In order to pinpoint the time when the program reaches state 0 (if such a times exists), we may first consider the function which identifies the exponent of 2 in $g(n1, \cdots, n_k, t)$, that is the function $E_2\left(g(n1, \cdots, n_k, t)\right)$ (which is a composition of recursive functions and therefore recursive), in the notation from earlier. We would like to identify the smallest value of t for which

$$E_2\left(g(n_1, \cdots, n_k, t)\right) = 0$$

(if this occurs at all), i.e. to minimalise $E_2\left(g(n_1, \cdots, n_k, t)\right)$ with respect to t.

Minimalising a recursive function yields a recursive function, by definition, so if we refer to $h : \mathbb{N}_0^k \to \mathbb{N}_0$ as the 'minimalising function', then $h(n_1, \cdots, n_k)$ is the required smallest value of t for which the function f terminates, when started with input $(n_1, \cdots, n_k)$ (and is undefined if f does not terminate, when started with input $(n_1, \cdots, n_k)$).

Then, $f(n_1, \cdots, n_k)$ should simply be the value in register R_1 when $t = h(n_1, \cdots, n_k)$, i.e. the exponent of 3 of

$$g\left(n_1, \cdots, n_k, h(n_1, \cdots, n_k)\right)$$

So, we may apply the function E_3 (in the notation from earlier) to finally identify the required function (which is a composition of recursive functions, and therefore recursive):

$$f(n_1, \cdots, n_k) \;=\; E_3\left(g\left(n_1, \cdots, n_k, h(n_1, \cdots, n_k)\right)\right)$$

Note also that function on the right hand side above is undefined precisely when the register machine for $f(n_1, \cdots, n_k)$ does not terminate, i.e. precisely when $f(n_1, \cdots, n_k)$ is undefined.

So, we may completely identify $E_3\left(g\left(n_1, \cdots, n_k, h(n_1, \cdots, n_k)\right)\right)$ with $f(n_1, \cdots, n_k)$.

Therefore, as required, the computable (partial) function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is recursive.

This result, which is in some ways analogous to the Completeness Theorems for propositional and first order predicate logic, exemplifies what is known as *Church's thesis*:

"In general, recursive functions and computable functions should coincide (if 'sensibly' defined)"

4.3 Encoding and the halting problem

The sketch of the proof of Theorem 4.12, described above, relies on encoding the sequence of 'complete states' of a register machine using the natural numbers. For the remainder of this chapter, we will study some more consequences of this general type of 'encoding'.

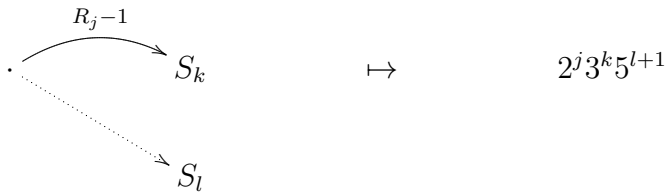
Let us start by extending the scope of our encoding. In the sketch of the proof of Theorem 4.12, a natural number is used to encode each state of a given register machine, with given input.

Let us now see how a single natural number may be used to encode a computable function 'as a whole', i.e. how a single number may be used to 'capture' the information present in all of the instructions associated to a given program.

Let us show how we may encode an instruction itself, by giving rules to encode each of the two types of instructions that may be present in a program:

- An instruction which adds 1 to R_j, and then moves to state S_k may be encoded by the number $2^j 3^k$:

- An instruction which, if $R_j > 0$, subtracts 1 from R_j, and then moves to state S_k, while, if $R_j = 0$, moves to state S_l, may be encoded by the number $2^j 3^k 5^{l+1}$:



The above encoding ensures that there cannot exist two different instructions with the same code. For example, any instruction of the second type, involving subtraction, would always contain a factor of 5, thus distinguishing it from an instruction of the first type, involving addition; the presence of the exponent '$l+1$' of 5 (rather than of the exponent 'l' say) in the second type of instruction ensures that there is a power of 5 present even when the terminal state, S_0, is involved, i.e. even when $S_l = S_0$ above.

Each state of a program has an associated instruction, which we are now able to encode. By introducing a second 'layer' of prime powers, we may encode a whole program, by encoding the instructions associated to particular states.

Suppose that a program consists of states $S_0, \cdots, S_n$, and that the code of the instruction associated to state S_i, for $1 \leq i \leq n$, is c_i. Then, we can encode a whole program by the following code:

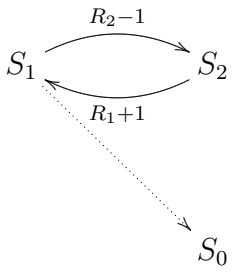
$$2^{\text{code of instruction in state } S_1} 3^{\text{code of instruction in state } S_2} \cdots p_{n-1}^{\text{code of instruction in state } S_n} = 2^{c_1} 3^{c_2} \cdots p_{n-1}^{c_n}$$

where, using the earlier notation for prime numbers: $p_0 = 2, p_1 = 3, p_2 = 5, \cdots$

Note that the terminal state S_0 has no associated instruction, by definition, and therefore does not require to be encoded in the same way as the states $S_1, \cdots, S_n$.

Let us see two example of programs being encoded using the method described above:

1) Suppose that we wish to encode the program described diagrammatically below; this program corresponds to the function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$, $f(m, n) = m + n$:



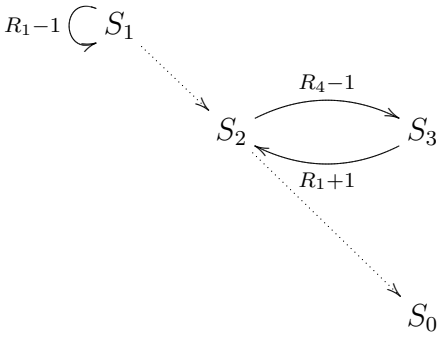
Then, the instruction associated to state S_1 is a 'subtraction type' instruction, which may be encoded by $c_1 = 2^2 3^2 5^1 = 180$.

Similarly, the instruction associated to state S_2 is an 'addition type' instruction, which may be encoded by $c_2 = 2^1 3^1 = 6$.

So, we may encode the whole program using the natural number

$$2^{c_1} 3^{c_2} = 2^{180} 3^6$$

2) Suppose that we wish to encode the program described diagrammatically below; this program corresponds to the projection function $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $f(n_1, \cdots, n_k) = n_4$, for $k \geq 4$:



Then, the instruction associated to state S_1 is a 'subtraction type' instruction, which may be encoded by $c_1 = 2^1 3^1 5^3$.

The instruction associated to state S_2 is also a 'subtraction type' instruction, which may be encoded by $c_2 = 2^4 3^3 5^1$.

Finally, the instruction associated to state S_3 is an 'addition type' instruction, which may be encoded by $c_3 = 2^1 3^2$.

So, we may encode the whole program using the natural number

$$2^{c_1} 3^{c_2} 5^{c_3} = 2^{2^1 3^1 5^3} 3^{2^4 3^3 5^1} 5^{2^1 3^2}$$

Note that not every natural number encodes a program. For example, the code associated to an instruction is always an even number, as is the code associated to a program, so that no odd natural number encodes a program (using the above version of encoding).

Similarly, note that, at the level of functions, the same computable function may correspond to two different programs, and, therefore, to two different codes. (For instance, the successor function $f : \mathbb{N}_0 \to \mathbb{N}_0$, $f(n) = n+1$, may be expressed using the program that simply adds 1 to register R_1, as well as, for example, the program that adds 1 to R_1, then subtracts 1 from R_1, and finally adds 1 to R_1 once again.)

However, since each computable function can be expressed, by definition, in terms of a program, a list of all the natural numbers that encode programs will definitely include a natural number corresponding to each computable function.

Since such a list is countable, we may deduce that the set of computable functions is countable.

Therefore, using Theorem 4.11, we may deduce that the list of recursive functions is countable.

This will allow us to give examples of functions that are not recursive; in particular, it will provide us with a way of proving that there exist functions that are not recursive (or computable).

We first introduce the following notation: the function f_n is the recursive (partial) function defined by a program encoded by the natural number n.

A consequence of this is that, since two different programs may correspond to the same recursive function, it might be possible that $f_i = f_j$, for some $i, j \in \mathbb{N}_0$, $i \neq j$.

We will extend our notation to numbers that do not encode programs, by setting f_n to be undefined if the number $n \in \mathbb{N}_0$ does not encode a program (and, therefore, if the number n does not correspond to a recursive function).

So, we may now write down a list of all recursive (partial) functions from $\mathbb{N}_0$ to $\mathbb{N}_0$, possibly with repetitions, and possibly including instances of a partial function that is not defined anywhere:

$$f_1, f_2, f_3, \cdots$$

Using this notation, we may provide examples of functions that are not recursive, by checking that they are not on the above list; this chapter concludes with a description of such a function, as well as a mention of the consequences of this observation for first order predicate logic.

Let us start by defining a function that determines which natural numbers encode recursive partial functions; the process involved will be one that is used (implicitly) in (the proof of) Proposition 4.13 below.

Consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0$, such that, for $n \in \mathbb{N}_0$, $f(n) = 1$ if n codes a program, while $f(n) = 0$ if n does not code a program.

Then, f is a recursive function. We may determine which values of n encode a program by, for example, considering the power of each prime present in n. If n encodes a program, then, as described earlier, for each prime p dividing n, the largest power of p dividing n must be even, and this power can only be divisible by the primes 2, 3 and/or 5 (these are the only primes that could possibly appear in the number encoding an instruction). As we saw earlier, the function $E_p : \mathbb{N}_0 \to \mathbb{N}_0$, which determines the largest power of a prime p dividing a natural number, is recursive (for each prime p), so we may form a recursive function that first determines such powers and then tests to verify whether they could correspond to powers arising from 'codes' of programs.

We now give an example of a function that is not recursive; its definition involves a type of 'diagonal construction', such as the one used to show (by contradiction) that the real numbers are not countable (by showing that, given a list of all such numbers, between 0 and 1 say, it is possible to construct a 'new' real number, which differs from the nth real number in the list at the nth decimal place).

Proposition 4.13. *The function $g : \mathbb{N}_0 \to \mathbb{N}_0$, where $g(n) = f_n(n) + 1$ if $f_n(n)$ is defined, and $g(n) = 0$ if $f_n(n)$ is undefined, is not recursive.*

Proof. Let us prove that g is not recursive by contradiction. Suppose that g is a recursive function. Then it must appear in the list $f_1, f_2, \cdots$ given above, i.e. $g = f_n$ for some $n \in \mathbb{N}$, and some f_n defined everywhere. However $g(n) = f_n(n) + 1$, so that $g(n) \neq f_n(n)$, i.e. g and f_n differ in their value at/for the number n. So, it cannot be the case that $g = f_n$, which leads to the required contradiction. Hence, g is not a recursive function. □

We may use this to give examples of related nonrecursive functions. One that is of particular interest is a function that may be described as a *halting function*.

This is the type of function that may be used to try to identify whether or not a given function is defined, at a given 'input value'. In the setting of computable functions, the halting function is a function that may be used to identify whether or not a register machine will terminate, given certain input.

The problem of deciding whether we are not it is possible to determine if any register machine will halt when started with any given input is often known as the *halting problem*.

The previous proposition may be used to show that the halting problem is undecidable. It is not possible to use a computable function in order to decide, in a finite number of steps, if computable (partial) functions are defined (on given input values).

This also gives a way of showing that first order logic is undecidable. In the setting of first order logic, decidability is defined as in the case of propositional logic (e.g. see the end of chapter 2). First order logic would be decidable if, given any finite set of formulae, S say, in a first order predicate language, $\mathcal{L}$, and any formula, α say, in $\mathcal{L}$, there was an algorithm that (in a finite number of steps) allowed us to determine whether or not there exists a proof of α from S (i.e. whether or not $S \vdash \alpha$).

This does not generally hold in the setting of first order logic, since we may transfer the undecidability of the halting problem to any first order theory that includes a 'copy' of the natural numbers.

To complete this chapter, let us give a brief, informal, overview of how this might be achieved:

1) Consider a first order predicate language $\mathcal{L}$, and define a suitable theory in this language.

2) Use an encoding (similar to the one used to encode programs earlier in this chapter) to encode each formula that can be formed within this setting (using prime powers say).

3) Each formula now corresponds to a number, so that a proof may simply be encoded by a sequence of numbers.

4) For a suitably chosen first order theory, the halting problem may be described within the setting of the theory. In this way, the problem of determining, in general, whether or not there exists a proof of a given formula in the setting of such a first order theory, will be undecidable.