PART I

BOOLEAN LOGIC
CHAPTER 1

THE BEGINNING

Mathematical logic, or as we will simply say, “logic”, is the science of mathematical reasoning. Its core consists of the study of the form, meaning, use, and limitations of logical deductions, the so-called proofs.

This volume, which is aimed at upper-level undergraduate university students who follow a course of study in computer science, mathematics, or philosophy, will emphasize mainly the use of proofs—it is written with the interests of the user in mind.

1.0.1 Remark. (Before we Begin) The symbol “\(\exists\)” goes at least as far back as the writings of Bourbaki. It has been made widely accessible to authors—who like to typeset their writings themselves—through the typesetting system of Donald Knuth (known as “\(\LaTeX\)”).

I use these “road signs” as follows: A passage enclosed between two single “\(\exists\)” symbols is purported to be very noteworthy, so please heed!

On the other hand, a passage enclosed between two double signs (“\(\exists\exists\)”) means two things.

\(^0\)This symbol is a stylized typographical version of the "(dangerous) winding-road" road sign.

Mathematical Logic. By George Tourlakis. Copyright (C) 2008 John Wiley & Sons, Inc.
The bad news is that it is rather difficult, or esoteric, or both. The good news is that you do not need to understand (or even read) its contents in order to understand all that follows. It is only there in the interest of the “demanding” reader. Such “doubly dangerous” passages allow me to digest without injuring continuity—you can ignore these digressions!

Learning to use logic, which is what this book is about, is like learning to use a programming language.

In the latter case, probably familiar to you from introductory programming courses, one learns the correct syntax of programs, and also learns what the various syntactic constructs do—that is, their semantics. After that, one embarks—for the balance of the programming course—on a set of increasingly challenging programming exercises, so that the student becomes proficient in programming in said language.

We will do an exactly analogous thing in this volume: We will learn to write proofs, which are nothing else but annotated sequences of formulae and are similar to computer programs in terms of syntactic structure—the annotations playing a role closely similar to that of comments in computer programs.

But to do that, we need to know, to begin with, what are the rules of correctly writing down a formula and a proof? We have to start with the syntax of these objects—formulae and proofs—precisely as it is done in the case of programming and its related objects, the programs.

Thus, we will begin with learning the syntax of the logical language, that is, what syntactically correct formulae and proofs look like. We will also learn what various syntactic constructs “say” (semantics). For example, we will learn that a formula makes a “statement”. A proof also makes a statement, that every formula in it is true in some very intuitively acceptable sense.

We will learn that correctly written proofs are finite and “checkable” means toward discovering mathematical “truths”. We will also learn via a lot of practice how to write a large variety of proofs that certify all sorts of useful truths of mathematics.

The above task, writing proofs—or “programming in logic” if you will—is our main aim. This will equip you with a toolbox that you can use to discover or certify truths. It will be handy in your studies in computer science, and in whatever area of study or research you embark upon and where reasoning is required.

However, we will also look at this toolbox, the logic, as an object of study and study some of its properties. After all, if you want to take up, say, carpentry, then you need to know about tools such as hammers—their properties (e.g., hard and heavy) and limitations (e.g., unfriendly to fingers).

When using the toolbox to prove theorems, you work within logic. On the other hand, when studying the toolbox, you work in logic’s metatheory (in metalogic) to talk and reason about logic.

People often do this kind of study with programming languages, looking at them as objects of study rather than as instruments to write programs with. For example, in an advanced course on the comparative study of programming languages one looks at several programming languages and compares them for features, suitability
for certain programming tasks - for any specific task some are more suitable than others - limitations, etc.

Here is another analogy: In the "real world" that we live in, one builds flight simulators, which we use to simulate flying an airplane, and in the process we learn how to do so. The real world where the simulator is built is the simulator's metatheory, where we can, among other things, study the properties and limitations of simulators and compare several simulators for features such as relative "power" (i.e., how effective or realistic they are), etc. Similarly, formal logic is built within "real mathematics", as we will see in the next section. It, too, is a "simulator" employed to write formal proofs that certify the truth of mathematical statements. These proofs imitate the kind of informal proofs one typically employs in informal mathematics but do so within a precisely specified system of notation (called language), rules, and assumptions. Thus, using formal logic is a means to learn how to write proofs — and not only formal proofs — just as using a flight simulator is a means of learning how to fly a real plane. The metatheory of logic — the "real mathematics" — addresses questions among the deepest of which is the question of how far formal logic can go in discovering mathematical truths.

Let us next look more closely at the similarity between programming languages and programming on one hand and logical languages and proving on the other, and argue that, similar as the two activities may be, the second one is a bit easier!

(1) In programming, you use the syntactic rules to write a program that solves a problem.

(2) In logic, you use the syntactic rules to write a proof that establishes a theorem.

In the latter task you are done as soon as the proof ends. At the end of the proof you have your theorem, exactly as stated.

In the former task, programming, it is not enough to just write a program! You next have to convince your boss, or your instructor, that the program indeed solves the problem; that it is "semantically correct" with respect to the problem's specification.

Note that in proving a theorem you have a purely syntactic task. Once your correctly written proof ends with the theorem you were trying to prove, you are done. There is no messing about with semantics.

There is another reason why programming is harder than proving theorems: Programming has to be painstakingly precise because it involves your writing instructions for a dumb machine to "understand" and follow. You must be absolutely and pedantically clear in your instructions.

On the other hand, you address a proof to a human who knows as much as you do, or more, about the subject. This human will in general accommodate a few shortcuts that you may want to take in your presentation.

In short, proofs are read by "intelligent" humans, while programs are read by "dumb" computers. We need to work really hard to speak at the level of the latter.

Will you ever need to deal with semantics in logic? Yes! Semantics is useful when you want to disprove (or refute) something, that is, to prove that it is a false
statement, a fallacy. We will talk about semantics later—three times: once under Boolean logic, once under predicate logic, and one last time in the Appendix.

There are many methodologies or paradigms (and corresponding programming languages suitable for the task) for writing programs. For example (add the word programming after each italicized keyword), \textit{procedural} (Algol, Pascal, Turing), \textit{functional} (LISP), \textit{logic} (Prolog), and \textit{object-oriented} (C++, Java). Most computer science departments will expose their students to many of the above.

Similarly there are several methodologies for writing proofs. For example (add the word style after each italicized keyword), \textit{equational} (the one favored by [17]), \textit{Hilbert} (favored by the majority of the mathematics, computer science, and logic literature), \textit{Gentzen's natural deduction}, etc.

My aim is to assist the reader to become an able user of the first two styles: the equational and the Hilbert style of proof.

In both methodologies, an important required component is the systematic annotation of the proof steps. Such annotation explains why we do what we do, and has a function similar to that of comments in a program.

Okay; one can grant that a computer science student needs to learn programming. But logic? You see, the proper understanding of propositional logic is fundamental to the most basic levels of computer programming, while the ability to correctly use variables, scope, and quantifiers is crucial in the use of loops, and subroutines, and in software design. Logic is used in many diverse areas of computer science, including digital design, program verification, databases, artificial intelligence, algorithm analysis, computability, complexity, and software specification. Besides, any science that requires you to reason correctly to reach conclusions uses logic.

When one is learning a programming language, one often starts by learning a small \textit{subset} of the language, just to smooth the learning curve. Analogously, we will first learn—and practice—a subset of the \textit{logical language}. This we will do not due to some theoretical necessity, but due to pedagogical prudence. This particular, “easy” subset of (the “full”) logic that we will embark upon learning goes by many names: \textit{Boolean logic, propositional logic, sentential logic, sentential calculus, and propositional calculus}.

The “full logic” we will call by any of the names \textit{predicate calculus, predicate logic, or first-order logic}.

I like the \textit{calculus} qualifier. It connotes that there is a \textit{precise way to “calculate”} within logic. It emphasizes that building proofs is an algorithmic and precise process, just like programming.

Indeed, it turns out that you can write a program, say, in Pascal, that will accept no input, but if it is allowed to run forever it will print all the theorems of logic\textsuperscript{7} (and not just those of the Boolean variety)—and never print a non-theorem!—in some order, possibly with some repetitions (cf. A.4.7 on p. 270).

\textsuperscript{7}We will soon appreciate that there are infinitely many theorems in logic.
Equivalently, we can write a program that is a theorem verifier. That is, given as input a theorem, the program will verify that it is so, in a finite number of steps. If the input is a non-theorem, our verifier will decline an answer—it will run forever.

Thus, proving theorems is a mechanical process!

**Digression:** The above assertion is an example of a true assertion about the logic, not one that we can prove using exclusively the tools of logic as a tool. It is a metatheorem of logic as we say, not a theorem.

The proof of this metatheorem requires techniques much more powerful than—indeed external to—those that the logic provides. We will prove this metatheorem in the Appendix to Part II (A.4.6).

So metatheorems are truths about the logic that we prove with tools external to the logic, while theorems are truths that the logic itself is capable of proving.

There is some danger that the above statement, “proving theorems is a mechanical process”, may be misinterpreted by some as one advocating that we build proofs by mindlessly shuffling symbols. *Nothing is further from reality.*

The statement must be understood precisely as written. It says that there is a “mindless” way, a programmable way, to generate and print all possible theorems of logic, and, equivalently, also a programmable way to verify all theorems, which, however, refuses to verify any non-theorem by “looping” forever when presented with any such as input.

But it is not a recipe for how we ought to behave when we write proofs. This is not the way a mathematician, or you or I, go about proving things—mindlessly. In fact, if we do not understand what is going on, we cannot go too far.

Moreover, interesting, even important, as this result (about the existence of theorem verifiers) may be theoretically, it is useless practically, as we further discuss below.

Our task is different. In general, we are more inquisitive. Given an arbitrary (mathematical) statement, we do not know ahead of time if it is a theorem or not. This italicized statement, the so-called decision problem of logic, is what we normally are interested in. Thus, our “verifier” is not very helpful, for if the statement that we present it as input is not a theorem, then the verifier will run forever, not giving an answer.

**Hmm.** Can we not write a decider for logic? The answer to this is interesting, but also reassuring to mathematicians (and all theorists): Their jobs are secure!

(1) For Boolean logic, we can, since the question “Is this statement a theorem?” translates to “Is this statement a tautology?” (cf. 3.2.1). The latter can be settled algorithmically via truth tables. But there is a catch: Checking a formula (the formal counterpart of a “statement”) for tautology status is an infeasible problem. So we can do it in principle, but this fact is devoid of any practical value.

---

8That this formulation of the claim is equivalent to the preceding one is a standard result of computability. Cf. Appendix to Part II, Remark A.3.91 on p. 362.

9The term *infeasible*—also *impossible*—has a technical connotation in complexity theory: It means a problem for which we know of no algorithm that runs in polynomial time as a function of the input.
(2) For predicate logic, the answer is more pleasing to mathematicians.

First, there exists no decider for this logic if we expand it minimally so that it can reason about the theory of natural numbers (this is Alonzo Church's theorem, [3, 4]).

Second, even if one were to be satisfied simply with a verifier for theorems, then we still would have no general solution of any practical value in hand. Indeed, again considering the logic augmented so that it can "do number theory," any chosen verifier \( V \) for this logic would be extremely slow in providing answers in the following precise sense: For any choice of a step-counting function \( f(n) \), there is an infinite subset \( S \) of the set of theorems of number theory, such that each theorem-member, \( T \), of \( S \) that is composed of \( n \) symbols requires for its verification more than \( f(n) \) steps to be performed by \( V \).\(^{10}\) This is a result of Hartmanis ([19]).

Let us stop digressing for now. In the next section we begin the study of the sublogic known as propositional calculus.

1.1 **BOOLEAN FORMULAE**

We will continue stressing the algorithmic nature of the discipline of proving, just as it is the case in the discipline of programming.

In particular, just as in serious programming courses the programming language is introduced via precise formation rules that allow us to write syntactically correct programs, we will be every bit as serious by introducing very precisely the rules for writing syntactically correct (1) formulae and (2) proofs.

Once again, the syntax of the logical language is much simpler to describe than that of any commercially available programming language.

So, how does one build—i.e., what are the rules for writing down correctly--formulae?

Continuing with the programming analogy, you will recall that to define a programming language, i.e., the syntax of its programs, one starts with the list of admissible symbols, the so-called alphabet. In some languages, the alphabet includes symbols such as "3, 4, 0, ., A, B, c, d, E, +, ×, −" and "keywords"—that is, multiple-character symbols—such as if, then, else, do, begin.

Similarly, in Boolean logic, we start with the basic building blocks, which collectively form what is called the alphabet (for formulæ). Namely,

\[^{10}\text{For example, consider } f(n) = 2^{22n}. \text{ If we think of } f(n), \text{ for each } n, \text{ as representing picoseconds of run time of the verifier } V \text{ (1 picosecond is } 10^{-12} \text{ seconds), then every member of } S \text{ of length more than 4 symbols will require the verifier } V \text{ to run for more than } 5.76045 \times 10^{296} \text{ years!}
A1. Symbols for variables, called the **Boolean** or **propositional** or **sentential** variables. These are \( p, q, r \), with or without primes or subscripts (i.e., \( p', q_{13}, r'' \) are also symbols for variables).

We often need to write down expressions such as \( A[p := B] \), to be defined later (1.3.15), but do not wish to restrict them to the specific variable \( p \). Nor can we say things such as "for any Boolean variable \( p \) we consider \( A[p := B] \ldots \)" as there is only one specific \( p \).

We get around this difficulty by employing so-called **metavariables** or **syntactic variables**—i.e., symbols outside the alphabet that we can use to refer to or point, generally, to any variable. We adopt the names for those to be the boîdface \( p, q, r \) with or without primes or subscripts. Thus \( p_{13}^0 \) names any variable \( p, q, r'' \), \( q_{157}^0 \), etc. Rarely if ever in this volume will we need to use more Boolean metavariables than these two: \( p, q \).

We can now use the expression "for every Boolean variable \( p \) we consider \( A[p := B] \ldots \)" referring to what \( p \) names rather than to \( p \) itself. Two analogous examples are, from algebra, "for every natural number \( n \)" (\( n \) is not a natural number!) and, from programming, where we might say about Algol, "for each variable \( x \), the instruction \( x := x + 1 \) means to increase the value of \( x \) by one." Again, \( x \) is not a variable of Algol; \( X \subseteq \{ X \subseteq 399 \), though, are. But it would be meaningless to offer the general statement "for each variable \( X \subseteq \), the instruction \( X \subseteq := X \subseteq + 1 \), etc." since \( X \subseteq \) is a specific variable of the Algol syntax. The programming language metavariable \( \alpha \) allows us to speak of all of Algol’s variables collectively!

On the other hand, the expression "for every Boolean metavariable" refers to the set of metavariables themselves. \( \{ p, q, r_{13} \ldots \} \) and will be rarely, if ever, used. The expression "for every Boolean metavariable \( p \)" is as nonsensical as "for every Boolean variable \( p \)".

A2. Two symbols for **Boolean constants**, namely \( \top \) and \( \bot \). These are pronounced variously in the literature: \textit{verum} (also \textit{top}, or symbol "true") and \textit{falsum} (also \textit{bottom}, or symbol "false").

A3. Brackets, namely, \( ( \) and \( ) \).

A4. "Boolean connectives", namely, the symbols listed below, separated by commas

\[ \neg, \wedge, \vee, \rightarrow, = \]

Let us denote by \( \mathcal{V} \) the alphabet consisting of the symbols described in A1–A4.

---

11Usually, the qualifier \textit{symbol} is dropped and then the context is called upon to distinguish between "true/false" the symbols vs. "true/false" the **Boolean values** of the metatheory (introduced in Section 1.3). In particular, cf. Definition 1.3.2 and Remark 1.3.3.
1.1.1 Remark. (1) Even though I say very emphatically that $p, q, r$, etc., and also $\top$ and $\bot$, are just symbols,\footnote{Some logicians put it more emphatically: "meaningless symbols".}—the former standing for variables, the latter for constants—yet, I will stop using the qualification symbols, and just say variables and constants. This entails an agreement: I always mean to say symbols, I just don't say it.

(2) Most variable symbols are formed through the use of "subsymbols"—such as $0, 1, 2, 3$—that are not members of the alphabet $\mathcal{V}$ themselves; e.g., $P_{1,0,0,3}$. This does not detract from the fact that each variable (name) is a single symbol of $\mathcal{V}$. Entirely analogously with, say, the keywords of Algol if, then, begin, for, etc.

(3) Readers who have done some elementary course in logic, or in the context of a programming course, may have learned that $\land, \lor$ are the only connectives one really needs since the rest can be expressed in terms of these two. Thus we have deliberately introduced redundancy in the adopted set of connectives (2) above. This choice in the end will prove to be user-friendly and will serve our aim to give a prominent role to the connective $\equiv$ in the axioms and in rules of inference (Section 1.4).

1.1.2 Definition. (Strings or Expressions; Substrings) We call a string (also word or expression), over a given alphabet, any ordered sequence of the alphabet's symbols, written adjacent to each other without any visible separators (such as spaces, commas, or the like).

For example, $aabba$ is a string of symbols over the alphabet $\{a, b, c, 0, 1, 2, 3\}$ (note that you don't have to use all the alphabet symbols in any given string, and, moreover, repetitions are allowed). Ordered means that the position of symbols in the string matters; e.g., $aab \neq aba$.

We denote arbitrary strings over the alphabet $A1-A4$ by string variables, i.e., names that stand for arbitrary or specific strings. Specific strings, or string constants, are sometimes enclosed in double quotes to avoid ambiguity. For example, if we say

Let $A$ be the string $aab$.

we need to know whether the period is part of the string or not. If it is not we symbolically indicate so by writing

Let $A$ be the string "aab".

If it were part of the string, then we would have written instead

Let $A$ be the string “aab”.

String variables—by agreement—will be denoted by uppercase letters $A, B, C, D, E, F, P, Q, R, S, W$, etc., with or without primes or subscripts. In particular, since Boolean expressions (and theorems) are strings, this naming is valid for this special case, too.

\footnote{\textit{E.g.}, "Let $A$ be any string".}

\footnote{\textit{E.g.}, "Let $A$ stand for ( $(p \land q)$ )".}
The major operation on strings is *concatenation*, that is, juxtaposition. To concatenate the strings (named) $A$ and $B$, in that order, is to form the string (named) $AB$ that consists of the symbols in $A$, from left to right, immediately followed by the symbols in $B$, from left to right. Thus, if $A$ is $001$ and $B$ is $10$, then $AB$ is $00110$.

Clearly, concatenation is an associative operation, i.e., $(AB)C = A(BC)$. Hence, when we omit brackets, as we normally do, and simply write $ABC$, there is no ambiguity since wherever we may insert the brackets that we “forgot” makes no difference!

There is a very special string that we call the *empty string* and denote by $\epsilon$ (this being a specific string, a constant, we deviate from the naming convention $A$, $B$, $C$, . . . above). What is special about it is that it contains no symbols, so that $\epsilon A = cA = A$.

"$B$ is a *substring* of $A"$ means that for some strings $C$ and $D$ we have $A = CBDA$.

For example, over the alphabet $\{a, b\}$ we have that $a$ is a substring of $aab$. Indeed there are two occurrences of $a$ in $aab$ as substrings: A first (shown boxed) is justified by noting $aab = a[a]ab$ and a second is justified by noting $aab = a[\overline{a}]b$.

We can build all sorts of *expressions* over our Boolean alphabet $\mathcal{V}$, such as $p, q, r, \neg a, (p \rightarrow r)$, and a lot of others.

Some such strings (e.g., the last one above) are *well-formed formulae* (in short, wff), the rest being gibberish.

*Hm*mm. How can we tell? For example, if we asked an unsuspecting (not logically trained) passerby which of the following are “well-formed”

$$p \equiv p$$
$$p \rightarrow ((p \lor q) \equiv q) \equiv ((p \land q) \equiv p)$$

we would have no right to expect any better than lucky guesses from him (we can check, by asking him “why?” in each case).

So, how can we tell? The obvious (silly) answer would be, "Why not tabulate all formulae? Then we can check any string for formula status by table look-up. If the string is in the table, then it is a formula; otherwise it is not a wff."

Of course this is silly, for we cannot write down an infinitely long table such as a table of all formulae would be.

We must find a way to define a set of infinitely many strings (the formulae) *by a finite text*.

Pause. Can we do such a thing?

Absolutely. We will give a precise process that every time it is applied builds a formula, and will never build a nonformula. Moreover, it is "general enough" so that if it is applied over and over, for ever, it will build all formulae.

We are ready to define *formula-calculation*.

**1.1.3 Definition.** *(Formula-Calculation or Formula-Parse)* We will call *formula-calculation* (or formula-parse) any *finite* (ordered) sequence of strings that we may write respecting the following three requirements:
(1) At any step we may write any symbol from A1 or A2 of the alphabet (p. 9).

(2) At any step we may write the string \((-A)\), provided we have already written the string A.

(3) At any step we may write any of the strings \((A \land B)\), \((A \lor B)\), \((A \rightarrow B)\), \((A \equiv B)\), provided we have already written the strings A and B.

1.1.4 Example. In the first step of any formula-calculation, only requirement (1) of Definition 1.1.3 is applicable, since the other two require the existence of prior steps. Thus, in the first step, we may write only a variable or a constant. In all other steps, all the requirements (1)-(3) are applicable.

Here is a calculation (the comma is not part of the calculation, it just separates strings written in various steps):

\[ p, \top, (\neg \top), q \]

Verify that the above obeys Definition 1.1.3.

Here is a more interesting one:

\[ p, q, (p \lor q), (p \land q), ((p \lor q) \equiv q), (p \land q) \equiv p, (((p \lor q) \equiv q) \equiv ((p \land q) \equiv p)) \]

1.1.5 Definition. (Boolean Expressions or wff or Formulæ) A string A over the alphabet A1-A4 will be called a Boolean expression or a well-formed-formula if\(^{15}\) it is a string written at some step of some formula-calculation.

The set of Boolean expressions we will denote by \( \text{WFF} \). A member of \( \text{WFF} \) is often called a wff (a formula).

1.1.6 Remark. (1) The idea of presenting the definition of formulæ as a "construction" or "calculation" goes at least as far back as [2, 21].

(2) We used, in the interest of user-friendliness, active or procedural language in Definition 1.1.3\(^{16}\) (i.e., that we may do this or that in each step). A mathematically more austere (hence "colder" (!)) approach that does not call upon anyone to write anything down—and does not speak of "steps"—would say exactly the same thing as Definition 1.1.3 rephrased as follows:

A formula-calculation (or formula-parse) is any finite (tensored) sequence of strings, \( A_1, A_2, \ldots, A_n \) such that— for all \( i = 1, \ldots, n \)—\( A_i \) is one of:

(i) Any symbol from A1 or A2

(ii) \((\neg A)\), provided A is the same string as some \( A_j \), where \( 1 \leq j < i \)

\(^{15}\) if and only if.

\(^{16}\) Exactly as (21) does.
(III) Any of the strings \((A \wedge B), (A \vee B), (A \rightarrow B), (A \equiv B)\), provided \(A\) is the same string as some \(A_j\), where \(1 \leq j < i\) and \(B\) is the same string as some \(A_k\), where \(1 \leq k < i\) (it is allowed to have \(j = k\), if needed)

(3) There is an advantage in the procedural formulation 1.1.3. It makes it clear that we build formulae in stages (or steps), each stage being a calculation step.

In each step where we apply requirement (2) or (3) of 1.1.3, we are building a more complex formula from simpler formulae by adding a Boolean connective.

Moreover, we are building a formula from previously built formulae.

These last two remarks are at the heart of the fact that we can prove properties of formulae by induction on the number of steps (stages) it took to build it, or more simply, by induction on its “complexity” (that is, the total numbers of connectives in the formula, counting repetitions; see next section).

The concluding remark above motivates an “inductive” or “recursive” definition of formulae, which is the favorite definition in the “modern” literature, and we should become familiar with it.

1.1.7 Definition. (Alternative (Recursive) Definition of WFF) The set of all well-formed-formulae is the smallest set of strings, WFF, that satisfies

1. All Boolean variables are in WFF, and so are the symbols \(\top\) and \(\bot\). We call such formulae atomic.

2. If \(A\) and \(B\) are any strings in WFF, then so are the strings \((\neg A), (A \wedge B), (A \vee B), (A \rightarrow B), (A \equiv B)\). □

1.1.8 Remark. (a) Why “recursive”? Because item (2) in 1.1.7 defines the concept formula in terms of a smaller, or earlier, instance of “itself”: It says, in essence, “... \((A \vee B)\) is a formula, provided we know that \(A\) and \(B\) are formulae. ...

In programming terms, confronted with, say, the 3rd subcase of case (2) of the definition, we “call” the definition recursively, twice, to settle the questions “Is \(A\) a wff?” and “Is \(B\) a wff?” If “yes” for both, then we proclaim that \((A \vee B)\) is a wff.

(b) Part (1) in 1.1.7 defines the most basic, most trivial formulae. This part constitutes what we call the Basis of the inductive (recursive) definition, while part (2) is called the inductive, or recursive, part of the definition.

(c) 1.1.7 and 1.1.5 say the same thing, looking at it from opposite ends: Indeed, suppose that we want to establish that a given string \(D\) is a formula. If we are using 1.1.5, we will try to build \(D\) via a formula-calculation, starting from atomic

17Each of \(A\) and \(B\) are substrings (cf. 1.1.2) of \((A \vee B)\), so they are “smaller” than the latter. They are also “earlier” in the sense that we must already have them—all, know that they are formulae—in order to proclaim \((A \vee B)\) a formula.
ingredients and building as we go successively more and more complex formulae until finally, in the last step, we obtain $D$.

If on the other hand we are using 1.1.7, we are working backwards (and build a formula calculation in reverse). Namely, if $D$ is not atomic, we try to guess—from its term—what was the last connective applied. Say we think that it was $\rightarrow$, that is to say, $D$ is $(A \rightarrow B)$ for some strings $A$ and $B$. Now we have to verify our guess! This requires that the strings $A$ and $B$ are formulae. Thus, taking each of $A$ and $B$ in turn as (a new, smaller) $D'$ we repeat this process. And so on. This is a terminating process since the new strings we obtain (for testing) are always smaller than the originals.

Of course, I did not prove here that the two definitions define the same set WFF. But they do!

Technically, the term smallest is crucial in 1.1.7 and it corresponds to the similarly emphasized iff of 1.1.5. A proof that the two definitions are equivalent is beyond our syllabus. □

1.1.9 Example. Let us verify using 1.1.7 that $(p \lor q) \lor r$ is a formula.

call #1 We guess that the rightmost “$\lor$” is the last to apply; thus, using (2) in 1.1.7 we must now verify that $(p \lor q)$ and $r$ are formulae. Well, $r$ is by (1) in WFF. However, $(p \lor q)$ leads to call #2.

call #2 Again, using (2) this time we do not need any guessing; there is only one connective—we must verify that $p$ and $q$ are formulae. This is so by (1) in 1.1.7.

When we use Definition 1.1.7 to verify that a string is a formula, we say that we parse the string top-down. On the other hand, when we build the formula using Definition 1.1.5, then we are parsing it bottom-up.

Can we parse the above string in another top-down way? Obviously, our recursive call to the definition hinges around one of the two “$\lor$” symbols in the string. We must guess which is “the right one” (as the last connective to apply). Why is the leftmost connective not “right”?

Here is where metamathematical analysis comes in: The leftmost connective will work as the “last one to apply” iff (according to 1.1.7) “$(p \land q) \lor r$” are both formulae. The metamathematical analysis of formula syntax[^17] (see next section)

[^18]: Only hand-waved to that effect, arguing that for any string $D$ in WFF, 1.1.5 builds a calculation the normal way, while 1.1.7 builds it backwards. I conveniently swept under the rug the case where $D$ is not in WFF, i.e., is not correctly formed.

[^17]: I know that the separation of “mathematics” from “metamathematics” is at first tricky. Think of the hammer analogy: You do “theory” (or “mathematics” or “logic”) when you use the hammer. On the other hand, when articulating a principle such as “It is inevitable that I will hit my finger with the hammer”, then you are doing an analysis of the hammer’s behavior. You are doing “metatheory” (or “metamathematics” or “metalogic”).

Similarly, you do logic when you generate a formula according to 1.1.5, or backwards according to 1.1.7. However, articulating principles such as “There is only one way to parse a formula”, or “every formula has balanced brackets”, *adieu*, does not build formulae and thus lies within the metalogic.
tells us that every formula must have a balance of left and right brackets. So none of these two is a formula, and therefore the leftmost $\lor$ cannot be the last connective to apply.

In the course of a formula-calculation (1.1.3), we write some formulae down without looking back (step of type (1)). Some others we write down by combining via one of the connectives $\land$, $\lor$, $\to$, $\equiv$ two formulae $A$ and $B$ already written, or by prefixing one already written formula, $C$, by $\neg$.

In terms of the construction by stages, the formula built in this last stage had as immediate predecessors $A$ and $B$ in the first case, or just $C$ in the second case.

One can put this elegantly via the following definition:

1.1.10 Definition, (Immediate Predecessors) None among the constants $\top$ and $\bot$, or among the variables, have any immediate predecessors.

Any of the formulae $(A \land B), (A \lor B), (A \to B), (A \equiv B)$ have $A$ and $B$ as immediate predecessors. $A$ is an immediate predecessor of $(\neg A)$. Sometimes we use the acronym i.p. for immediate predecessor.

It turns out that a formula uniquely determines its i.p. We give a proof later (1.2.5).

1.1.11 Remark, (Priorities) In practice, too many brackets make it hard to read complicated formulae. Thus, texts (and other writings in logic) often come up with an agreement on how to be sloppy, but get away with it.

This agreement tells us what brackets are redundant—and hence can be removed—from a formula written according to Definitions 1.1.5 and 1.1.7, still allowing the formula to "say" the same thing as before:

1. Outermost brackets are redundant.

For the remaining two cases, it is easiest to think of the process in reverse: How to reinsert correctly (as per Definition 1.1.5) any omitted brackets.

2. Any other pair of brackets is redundant, if its presence (as dictated by 1.1.5) can be understood from the priority, or precedence, of the connectives. Higher-priority connectives bind before lower-priority ones. That is, if we have a situation where a subformula$^{20}$ $A$ of a formula has already been reconstructed as per 1.1.5, and is claimed by two distinct connectives $\circ$ and $\circ$, among those in (1) below, as in "\( \ldots \circ A \circ \ldots \)"; then the higher-priority connective "glues" first. This means that the implied brackets are (reinserted as) "\( \ldots (A \circ \ldots) \)" or "\( \ldots (A \circ \ldots) \)" according as $\circ$ or $\circ$ has the higher priority, respectively.

The order of priorities (decreasing from left to right) is agreed to be: $^{21}$

\[ \neg, \land, \lor, \to, \equiv \]  

\[ (\ast) \]

$^{20}$A subformula of a formula $B$ is a substring of $B$ that is a formula.

$^{21}$Other agreements for priorities are possible. I offered the one that most people use. But remember: It is only an agreement, which means (1) you must stick to it, and (2) it is neither "more right" nor "more wrong" than any alternative agreement.
(3) In a situation like "... o A o ..."—where A has already been reconstructed as in 1.1.5, and o is any connective listed in (1) above, other than —— the right o acts before the left. Thus the implied bracketing is "... o (A o ...)".

Similarly, in ¬¬¬A is short for \([-\{(\neg \neg A)\}]\).

We say that all connectives are right associative.

It is important to emphasize:

(a) This “agreement” results in a shorthand notation. Most of the strings depicted by this notation are not correctly written formulae, but this is fine: Our agreement allows us to decipher the shorthand and uniquely recover the correctly written formula we had in mind.

(b) I gave above the convention that is followed by 99.9% of writings in logic, and in (almost) all programming language definitions (when it comes to “Boolean expressions” or “conditions”).

(c) The agreement on removing brackets is a syntactic agreement.

In particular, right associativity says simply that, e.g., \(p \lor q \lor r\) is shorthand for \((p \lor (q \lor r))\) rather than \(((p \lor q) \lor r)\).

However, no claim is either made or implied that \((p \lor (q \lor r))\) and \(((p \lor q) \lor r)\) mean different things. At this point meaning (Boolean values) has not yet been introduced. When it is later on, we will easily see that \((p \lor (q \lor r))\) and \(((p \lor q) \lor r)\) mean the same thing. \(\square\)

1.1.12 Example. \(p\) stands for \(p\).

\(-p\) stands for \((\neg p)\).

\(p \rightarrow q \rightarrow r\) stands for \((p \rightarrow (q \rightarrow r))\).

If I want to simplify \(((p \rightarrow q) \rightarrow r)\), then \((p \rightarrow q) \rightarrow r\) is as simple as I can get it to be.

\(-p \lor q \lor r\) is short for \(((\neg p) \lor q) \lor r)\).

If in the previous I wanted to have \(\neg p\) act last, and \(\lor\) to act first, then the minimal set of brackets necessary is: \(-\{(p \lor (q \lor r))\}\). \(\square\)

A connection with things to come in a degree program in computer science.

Any set of rules that tell us how to correctly write down strings constitutes a so-called grammar. Formal language theory studies grammars, the sets of strings that grammars define (called formal languages), and the procedures (or “machines”) that are appropriate to parse these strings.

In an introductory course on “automata and formal language theory”, a student learns about formal languages. Such a student would quickly realize that Definition 1.1.7 is, in effect, a definition of a grammar for the “language” (i.e., set of
strings) WFF. He would utilize a neat notation,\(^{23}\) such as

\[
E ::= A \mid (E \land E) \mid (E \lor E) \mid (E \rightarrow E) \mid (E \equiv E) \mid (\neg E)
\]

\[
A ::= \lbrack p \mid q \mid r \mid p' \lbrack \ldots
\]

where \(E\) stands for (Boolean) Expression, \(A\) for Atom. "\(:=\)" is read as "is defined to be" and "\(\mid\)" is read as "or", separating alternatives in an "is defined to be"-list.

Thus, the first line says, in English, "A (Boolean) expression is defined to be an atom, or (\(\mid\) followed by an expression, followed by \(\land\) followed by an expression followed by \(\rightarrow\))", or, etc."

The second line defines atom as any of the constants or the variables (note the separating or's).

\[\text{\textcopyright symbol} \text{\textcopyright symbol}\]

1.2 INDUCTION ON THE COMPLEXITY OF WFF: SOME EASY PROPERTIES OF WFF

Suppose now that we want to prove that every \(A \in \text{WFF}\)\(^{23}\) has a "property" \(\mathcal{P}\).

The technique is to associate a natural number with each member of \(\text{WFF}\) and prove the property by induction on numbers. The most obvious number one may associate with a formula \(A\) is the formula's complexity:

1.2.1 Definition. (Complexity of a Formula) The complexity of a formula is the number of connectives—counting repetitions—occurring in the formula.

1.2.2 Example. Note that we can read the complexity accurately even if we write formulae in least parenthesized notation.

Every atomic formula has complexity 0. The complexities of \(p \rightarrow q \rightarrow p'\), \(\neg p \lor q \lor s\), and \(\neg p \land p' \lor p'' \rightarrow (p''' \equiv q)\) are 2, 3, and 5 respectively.

Brackets do not contribute to complexity.\(^{24}\)

\[\text{\textcopyright symbol} \text{\textcopyright symbol}\]

A crash course on induction. First off, let us recall what we call strong or course-of-values induction on the natural numbers (also known as complete induction):

Suppose that \(\mathcal{P}(n)\) is a property of the natural number \(n\). To prove that \(\mathcal{P}(n)\) holds for all \(n \in \mathbb{N}\)\(^{24}\) it suffices, in principle, to prove for the arbitrary \(n\) that \(\mathcal{P}(n)\) holds.

What we mean by "arbitrary" is that we do not offer the proof of \(\mathcal{P}(n)\) for some "biased" \(n\) such as \(n = 42\), or \(n\) even, or \(n\) with 105 digits, etc. If the proof indeed

\(^{23}\) Known as BNF notation, or Backus-Naur-Form notation.

\(^{24}\) \(\mathbb{N}\) denotes the set of all natural numbers \(\{0, 1, 2, 3, \ldots\}\). Thus "for all \(n \in \mathbb{N}\)" is elegant notation that says "for \(n = 0, 1, 2, 3, \ldots\)."
has not cheated by using some property of \( n \) beyond \( "n \in \mathbb{N}" \), then our proof is 
equality valid for any \( n \in \mathbb{N} \): we have succeeded in effect to prove \( \mathcal{P}(n) \), for all \( n \in \mathbb{N} \).

Now the above endeavor is not always easy. It would probably come as a 
surprise to the uninitiated that we can pull an extra assumption out of the blue 
and use it toward proving \( \mathcal{P}(n) \), and that when all is said and done this process 
is as good as if we proved \( \mathcal{P}(n) \) without the extra assumption!

This out-of-the-blue assumption is that

\[
\mathcal{P}(k) \text{ holds for all } k < n \tag{I}
\]

or, another way of putting it, that the history or course-of-values of \( \mathcal{P}(n) \),

\[
\mathcal{P}(0), \mathcal{P}(1), \ldots, \mathcal{P}(n-1) \tag{II}
\]

holds—that is, it is a sequence of valid statements. It goes by the name induction 
hypothesis (I.H.), and the technique is that of “proof by strong induction”.

A couple of comments:

1. As before, we still have to prove \( \mathcal{P}(n) \) for the arbitrary \( n \), although now we 
   have the I.H. as extra help.

2. We note that the history, (II), of \( \mathcal{P}(n) \) is empty if \( n = 0 \). Thus every proof by 
   strong induction has two cases to consider: the one where the history helps, because 
   it exists, i.e., when we have \( n > 0 \), and the one where the history does not help, 
   because it simply does not exist, i.e., when \( n = 0 \).

In summary, strong induction proofs have two cases:

- **I.S.** Where \( n > 0 \) and we are helped by the I.H. ((I) or (II) above). I.S. is an 
  abbreviation for indentation step.

- **Basis.** Where \( n = 0 \) and we are on our own! The proof for \( n = 0 \) is called the basis 
  step of the induction.

Since on occasion we will also employ “simple” induction in this book, let me 
remind the reader that in this kind of induction the I.H. is not the assumption of 
validity of the entire history, but that of just \( \mathcal{P}(n-1) \). As before, simple induction 
is carried out for the arbitrary \( n \), so we need to work out two cases: when the I.H. is 
really there \( (n > 0) \) and when it is not \( (n = 0) \). The case of proving \( \mathcal{P}(0) \) directly 
is still called the basis of the (simple) induction.

Tradition has it that in performing simple induction the majority of users in the 
literature take as I.H. \( \mathcal{P}(n) \) while the I.S. involves proving \( \mathcal{P}(n+1) \).

Correspondingly, we organize proofs of properties of formulae \( X \), \( \mathcal{P}(X) \), into 
two main cases (rather three, in practice: see below)—essentially carrying out a 
strong induction with the complexity \( n \) of \( X \) as a “proxy”. However, the complexity 
\( n \) of \( X \) is well hidden in the background of the argument and we do not mention it:

1. Case of atomic formulae (these are the only ones with complexity \( n = 0 \)) where 
   the proof is direct, without the benefit of the I.H.
(ii) Case of nonatomic formulae (corresponding to a complexity of \(X\) \(n > 0\))

where we will benefit from the I.H. that \(\mathcal{P}(A)\) holds for all formulae \(A\) that are less complex than \(X\) (i.e., they have complexity \(k < n\)).

In case (ii), if \(A\) is any formula less complex than \(X\), we will often say that "the I.H. applies to \(A\)", meaning precisely that "by the I.H., \(\mathcal{P}(A)\) holds".

Let us apply (i)–(ii) to obtain a framework of proofs by induction on the set of formulae or, as we say more simply, by induction on formulae.

Now, since every proper\(^{25}\) subformula \(A\) of \(X\) has a lesser complexity than \(X\), the I.H. applies on \(A\). In particular, the I.H. applies on all the i.p. of \(X\) (the definition of i.p. was given in Definition 1.1.10).

Thus, in practice, (i)–(ii) translate into the following simple framework for proofs by induction on formulae:

(a) \(X\) is atomic: Give a direct proof.

(b) \(X\) has the form \((\neg A)\). Give a proof on the assumption (I.H.) that \(\mathcal{P}(A)\) holds.

(c) \(X\) has the form \((A \circ B)\) — where \(\circ \in \{\land, \lor, \equiv, \rightarrow\}\). Give a proof for each case of \(\circ\) on the assumption (I.H.) that \(\mathcal{P}(A)\) and \(\mathcal{P}(B)\) hold.

Let us now prove a few properties of formulae by induction on formulae.

All the "theorems" (and their corollaries) of this section are about formulae and their syntax. They are not theorems of logic, but are metatheorems.

1.2.3 Theorem. Every Boolean formula \(A\) has the same number of left and right brackets.

Proof. The theorem is about formulae written properly, as per Definition 1.1.5, that is, before our agreements to simplify bracketing are applied.

We prove the property by induction on formulae, \(A\).

(1) Basis: \(A\) is atomic. Each atomic formula has 0 left and 0 right brackets. We are okay.

(2) \(A\) has the form \((\neg B)\). The I.H. applies on the less complex \(B\). So let \(B\) have \(m\) left and \(m\) right brackets. Then \(A\) — i.e., \((\neg B)\) — has \(m + 1\) of each.

(3) \(A\) has one of the forms \((B \land C)\), \((B \lor C)\), \((B \rightarrow C)\) and \((B \equiv C)\).

The I.H. applies to the less complex subformulae \(B\) and \(C\). So let them have \(m\) left/right and \(r\) left/right brackets respectively. Thus \(A\) has \(m + r + 1\) left, and as many right, brackets.

\(^{25}\) That is, not the same string as \(X\).
Note. A string $B$ is a prefix of a string $A$ if there is a string $C$ such that $A = BC$. The prefix is empty if it is the empty string (i.e., it has no symbols in it; it has length 0). It is proper if $A \neq B$.

1.2.4 Corollary. Any nonempty proper prefix of a Boolean expression $A$ has more left than right brackets.\footnote{Corollary is Latin that mathematicians, logicians, computer scientists, philosophers—and other reasoning people—use to characterize a statement that needs proof, but whose proof follows easily from another proved statement, or from the latter’s proof. One then speaks of “$A$ is a corollary of $B$,” meaning that $A$ easily follows from $B$ (and/or $B$’s proof).}

Proof. Induction on $A$.

Since $A$ denotes an arbitrary formula, “induction on formulae” can be rephrased as “induction on $A$”. Compare with “induction on natural numbers” and “induction on $n$”.

(1) $A$ is atomic. Note that none of the atomic formulae has any nonempty proper prefixes, so we are done without lifting a finger.

This is an instance of a statement being “vacuously true”: The statement has a typical instance that says, “All nonempty proper prefixes of $p$ have more left than right brackets.” Is this true? Absolutely! If you think otherwise, then show me just one nonempty proper prefix of $p$ that does not have an excess of left brackets. You cannot, because there are no nonempty proper prefixes of $p$. (The only nonempty prefix of $p$ is $p$, but this is improper.)

(2) $A$ has the form $(-B)$. The I.H. applies to $B$. Well, let’s check the nonempty proper prefixes of $(-B)$. These are (quotes not included, of course):

(a) “$C$”. Okay, by inspection.

(b) “$-$. Ditto.

(c) “$(-C)$”, where $C$ is a nonempty proper prefix of $B$. By I.H. if $m$ is the number of left and $n$ the number of right brackets in $C$, then $m > n$. But the number of left brackets of “$(-C)$” is $m + 1$. Since $m + 1 > n$, we are done.

(d) “$(-B)$”. By 1.2.3, $B$ has, say, $k$ left and $k$ right brackets. We are okay, since $k + 1 > k$.\footnote{The I.H. was not needed in this step.}

(3) $A$ has the form $(B \circ C)$—where “$\circ$” is any of $\land, \lor, \to, \equiv$. The I.H. applies on $B$ and $C$. Well, let’s check the nonempty proper prefixes of $(B \circ C)$. These are (quotes not included, of course):

(i) “$. Okay, by inspection.
(ii) "(D)" where D is a nonempty proper prefix of B. By 1.H., if m is the number of left and n the number of right brackets in D, then m > n. But the number of left brackets of "(D)" is m + 1. Okay.

(iii) "(B)". By 1.2.3, B has, say, k left and k right brackets. We are okay, since k + 1 > k.

(iv) "(B ◦ )". The accounting exercise is exactly as in (iii). Okay.

(v) "(B ◦ D)", where D is a nonempty proper prefix of C. By 1.2.3. B has, say, k left and k right brackets. By 1.H., D has, say, m left and r right brackets, where m > r. Thus "(B ◦ D)" has 1 + k + m left and k + r right brackets. Okay.

(vi) "(B ◦ C)". Easy.

The following tells us that once a formula has been written down correctly, there is a unique way to understand the order in which connectives apply.

1.2.5 Theorem. (Unique Readability) For any formula A, its immediate predecessors are uniquely determined.

Proof. Obviously, if A is atomic, then we are okay (nothing to prove, for such instances of A have no i.p.). Moreover, no A can be seen (written) as both atomic and nonatomic. The former do not start with a bracket; the latter do (cf. 1.2.6).

Suppose that A is not atomic. Is it possible to build this string as a formula in more than one way?

Can A have two different sets of i.p., as listed below? (Below, when I say "we are okay" I mean that the answer is "no", as the theorem claims.)

1. (¬C) and (¬D)? Well, if so, C is the same string as D (why?); so in this case we are okay.

2. (¬C) and (D ◦ F), where ◦ is any of ∧, ∨, →, ↔? Well, no (which means we are okay in this case too). Why "no"? For if (¬C) and (D ◦ F) are identical strings then they are, supposedly, two ways to read A, remember?), then "¬" must be the same symbol as the first symbol of D. Now the first symbol of D is one of "(" or an atomic symbol. None matches "¬".

3. (C ◦ D) and (F ◦ G), where ◦ and ◦ are any of ∧, ∨, →, ↔ (possibly the same symbol) and either C and F are different strings, or D and G are different strings, or both? Well, no!

(i) If C and E are different, then, say, C is a proper prefix of E (of course, C is nonempty as well (why?) By 1.2.4, C has more left brackets than right ones, but—being also a formula—it has the same number of left and right brackets.

28 So we cannot be so hopelessly confused as to think at one time that A has no i.p. and at another time that it does.
brackets (by 1.2.3). Impossible! The other case, $E$ being a proper prefix of $C$, instead, is equally impossible.

(ii) If $C'$ and $E$ match, then $\circ$ and $\circ$ match. This forces $D$ and $G$ to be the same string, since the strings $(C' \circ D)$ and $(E \circ G)$ are the same—okay again.

Having answered “no” in all cases, we are done.

1.2.6 Exercise. Prove that the first symbol of any formula $A$ is one of

(1) a variable
(2) $\top$
(3) $\bot$
(4) a left bracket

*Hint.* Induction on formulae, or directly from an analysis of formula-calculations (1.1.3).

1.2.7 Exercise. In footnote 20 of p. 15 we defined the concept of subformula, saying: “A subformula of a formula $A$ is a substring of $A$ that is also a formula.”

This definition does not offer itself toward showing rigorously that, e.g., “if all the occurrences of a subformula $B$ of $A$ are replaced by the same Boolean variable, say $p$, then the string so obtained is a formula.”

Do the following:

(1) Try to contradict me (I said, “This definition does not offer itself toward showing rigorously that, e.g., . . . ”)

(2) Regardless of how you did in (1), give an inductive definition of the concept subformula.

(3) Now use (2) to prove by induction on $A$ that “if all the occurrences of a subformula $B$ of $A$ are replaced by the same Boolean variable, say $p$, then the string so obtained is a formula.”

1.3 INDUCTIVE DEFINITIONS ON FORMULAE

Now that we know (by 1.2.5) that we can decompose a formula uniquely into its constituent parts, we are comfortable with defining functions (more generally “concepts”) on formulae by induction—or recursion—on formula complexity, or as we rather say, “by induction . . . or recursion . . . on formulae.”

This recursion will define the concept as follows:

- (Basis) If $A$ is atomic, we define the concept (or function) directly, depending on what we are trying to achieve.

- If $A$ is $(\neg B)$, then we “call” the definition recursively to define the concept for $B$. Depending on the nature of the concept we then, taking the presence of $\neg$ into account, extend the concept to the entire $A$.

---

29Some people prefer to use the term induction for proofs, and recursion for constructions. Others do not mind using the term induction for either.
• If $A$ is $(B \circ C)$, then we "call" the definition recursively for $B$ and $C$. Depending on the nature of the concept, taking the presence of $\circ \in \{\land, \lor, \equiv, \Rightarrow\}$ into account, we extend the concept to the entire $A$.

I will merely state that the above process is feasible because of 1.2.5, which allows us to have uniquely determined "components" of $A$, its i.p., on which we perform the "recursive calls".

A rigorous proof that indeed the process works, that we can effect recursive definitions on sets such as $\text{WFF}$, which themselves have been inductively defined (1.1.7), is beyond our aims. This subject is fully considered in [54].

1.3.1 Example. We consider here a simple example that shows what can happen if we attempt a recursive definition on a set of formulae that were defined in a manner that the uniqueness of i.p. was not guaranteed.

This time we define simple arithmetic formulae, without variables. As an alphabet we take

$$\{1, 2, 3, +, \times\}$$

Inductively, we define the set "arithmetic formulae", $\text{AR}$:

$\text{AR}$ is the smallest possible set of strings over the alphabet (1) that contains the strings of unit length, 1, 2 and 3, and, moreover, if the strings $X$ and $Y$ are in $\text{AR}$, then so are $X + Y$ and $X \times Y$.

The strings 1, 2, and 3 are the atomic formulae of $\text{AR}$.

The concept i.p. is defined on the formulae of $\text{AR}$ in the obvious manner: The atomic formulae do not have any i.p., and $X + Y$ and $X \times Y$ have, each, $X$ and $Y$ as i.p.

$1 + 2 \times 3$ is an example of a formula in $\text{AR}$ that does not have a unique i.p.

Indeed, according to the definition, we have two sets of i.p. here: $\{1, 2 \times 3\}$ and $\{1 + 2, 3\}$.

Both i.p. sets are correct. Remember that any agreement on the priority of the connectives—and we entered into no such agreement—is not part of the rigorous definition of formula syntax for $\text{AR}$; thus, let us not assume that any such agreement is implied here!

But why do we fear the multiplicity of i.p. sets for $1 + 2 \times 3$?

Let us attempt to define an "evaluation" function, inductively, on the set $\text{AR}$. We will call it $E Vincent$. Here is the "natural" definition $E Vincent$:

$$E Vincent(1) = 1$$
$$E Vincent(2) = 2$$
$$E Vincent(3) = 3$$
$$E Vincent(X + Y) = E Vincent(X) + E Vincent(Y)$$
$$E Vincent(X \times Y) = E Vincent(X) \times E Vincent(Y)$$
Now, what is $EV(1 + 2 \times 3)$?

To answer this, we need to decompose $1 + 2 \times 3$ into a set of i.p. so that we can next do our recursive calls of $EV$ (see the two last cases in the definition of $EV$). Unfortunately, we obtain two distinct answers!

First we compute according to the decomposition $\{1, 2 \times 3\}$:

$$EV(1 + 2 \times 3) = EV(1) + EV(2 \times 3)$$
$$= EV(1) + (EV(2) \times EV(3))$$
$$= 1 + (2 \times 3)$$
$$= 7$$

Next, let us do so according to the other decomposition, $\{1 + 2, 3\}$:

$$EV(1 + 2 \times 3) = EV(1 + 2) \times EV(3)$$
$$= (EV(1) + EV(2)) \times EV(3)$$
$$= (1 + 2) \times 3$$
$$= 9$$

“Natural”, or “only”? I said earlier: “Here is the “natural” definition $EV$.”

But is there any other? Yes, infinitely many! We must get used to the idea that once we define the syntax of a set of strings—of a formal language, as we say in the theory of computation, the language here being AR—we do not have anything more than the syntax, i.e., the knowledge of the “shape” of such strings. All strings in AR are “meaningless”, and their semantics (or interpretation) is totally up to us. The variety of such interpretations at our disposal is infinite.

I wanted the above example to be immediately relevant to our existing knowledge, to be “natural”. That is why I gave the meaning to all the meaningless symbols of alphabet (1) that anyone would likely expect.

However, a “meaningless” symbol such as “1” may stand for infinitely many different objects of mathematics. Staying in algebra, I will mention the following (infinitely many) interpretations: The symbol may be interpreted, as here, to be the number “one”, but also as the unit “$2 \times 2$” matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

or the unit “$3 \times 3$” matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

or ...

But “unit” of some sort or another is not an intrinsic meaning of “1”. The symbol could stand for the number 0, or 42, or for some other mathematical object.
Similarly, the meaningless symbol “+” can be interpreted as “plus” on numbers, but also as “plus” on \(n \times n\) matrices (for various \(n\)), and also as concatenation of strings, union of so-called regular expressions, etc. Similar comments hold for all the other symbols of the alphabet (1).

There will be two main examples of recursive definitions in this section. The first follows the definition of the concept of state and is the inductive definition of value of a formula in a state. This leads to Boolean formula semantics.

The other will be the inductive definition of substitution of formulæ into a variable, an operation that is central in the use of the “Leibniz rule” in proofs.

But first, let us introduce states and Boolean semantics.

As we said early on (p. 6), Boolean logic is a subset of the (full) logic on first order languages and is, mainly, a pedagogical tool, since it is “easy” and therefore its study painlessly trains us and prepares us for the study of predicate logic.

Does it have any other use? Yes. We can imagine that the Boolean variables of propositional logic are “abstractions” of statements in mathematics, computer science, philosophy, etc. By the term abstraction of a statement I mean the assignment of a name—a Boolean variable—to it, purposely forgetting the intrinsic semantic content (i.e., what it says) of the original statement.

As an example, we can decide the logical correctness or not of the statement

\[ x = \mathbb{N}_0 \rightarrow x = \mathbb{N}_0 \lor y > \mathbb{N}_1 \]  

within Boolean logic by the method of abstraction, not bothering with the fact that most of the symbols above—i.e., \(x, y, =, >, \mathbb{N}_0, \mathbb{N}_1\)—are not even in the alphabet \(\forall\) of propositional logic!

Indeed, we abstract the elementary statements\(^{31}\) “\(x = \mathbb{N}_0\)” and “\(y > \mathbb{N}_1\)”, naming them, say, \(p\) and \(q\). Since they are two distinct statements, I used two distinct Boolean variables. Thus (*) becomes

\[ p \rightarrow p \lor q \]  

and, as we will soon see, it holds independently of what hides under the names \(p\) and \(q\).

Two useful observations are motivated from this example on abstraction and are noteworthy:

- The “object” of propositional logic is not the study of the elementary “statements” (or “propositions”), that is, of the Boolean variables and their intrinsic “semantic content”. After all, variables have no intrinsic semantic content. Once we name through such a variable an “elementary”—i.e., connective-free—substatement, we turn around and forget the meaning of the original!

To put it positively, the “object” of study is, exclusively, the Boolean connectives and their behavior.

---

\(^{21}\) E.g., advanced texts such as [45, 53] introduce predicate calculus directly and do not cover propositional logic.

\(^{31}\) What makes them “elementary” is that they do not involve Boolean connectives.
The statements that we can abstract are not restricted to those mathematical ones (or other) that are variable-free, like "3 + 9 = 7 > 101". E.g., the elementary statement "x = 8", above depends on the variable x and therefore whether it is intrinsically true or false is indeterminate (depending on the value of x). But this did not hinder our abstraction in any manner!

Here is why: In the process of abstraction—to which we come back in detail in 4.1.25—the Boolean variables that we use as names do not inherit the semantic content of the statements they name. Thus we do not care whether the truth or falsehood of what a Boolean variable names can be determined or not!

The only thing that matters is the Boolean structure of the original statement, that is, how the original statement is put together where the connectives act as "glue". In the previous example, all we needed to know was that the statement had the Boolean structure (**).

The semantics of Boolean formulae is defined—in the metatheory of propositional logic—through a process that allows us to calculate whether a formula is true or false, and this under certain conditions.

Our aim below is to make precise what we mean by "conditions", and to give the process according to which we calculate the "truth value" of a formula under any conditions.

As you probably know from programming courses, only two values are possible for a formula in "classical" Aristotelian logic as well as in its descendant, mathematical logic, which we study here. These two values are true and false—which we collectively call truth values.

Thus we need a set of two distinct objects, which we will find outside the alphabet \( \mathcal{V} \), in logic's metatheory. We freeze, i.e., reserve, these two values in this volume and they will serve, to the last page, as our truth values.

Our choice is the set \( \{ t, f \} \) of truth values. We will pronounce \( t \) as true and \( f \) as false.

Some programming languages, but even books on logic, use different sets of truth values, such as \( \{ 0, 1 \} \). At the end of the day, neither how we write them down nor how we pronounce the truth values matters, as long as we have exactly two distinct ones.

**1.3.2 Definition.** A state \( \nu^{32} \) is a function that assigns the value \( t \) or \( f \) to each Boolean variable, while it assigns necessarily the value \( f \) to the constant \( \bot \) and necessarily the value \( t \) to the constant \( \top \).

We pronounce \( t \) and \( f \) "false" and "true" respectively. On the chalkboard one usually denotes them by \( \bar{f} \) and \( \bar{t} \) respectively.

If, say, the value \( t \) is assigned to \( q'' \), then we write \( \nu(q'') = t \).

\(^{32} \nu \) for value. Alternative letter is "s" for state.
1.3.3 Remark. (1) A state $v$ is one of the infinitely many possible "conditions" where we are interested in finding the truth value of a formula and where it is possible to compute such a value.

(2) A function $v$ is, of course, a table of input/output values such as

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp$</td>
<td>$f$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$t$</td>
</tr>
<tr>
<td>$p$</td>
<td>$t$</td>
</tr>
<tr>
<td>$q$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

where no two rows contain the same input. Disobeying this condition would result in ambiguity, assigning to a variable both values $f$ and $t$.

By definition, a state is an infinite table, so we cannot fit it on a page. Mathematically, it is an infinite set of input/output ordered pairs.

Since the truth values $f$ and $t$ lie outside the alphabet $\mathcal{V}$ (p. 9) of our logic, they are symbols that, despite the similarity of their pronunciation with that of the names of the "meaningless" formal $\perp$ and $\top$ respectively, are different from the latter.

In particular, neither the metasymbol $f$ nor the metasymbol $t$ may appear in a formula!

But why the fuss of assigning the values $f$ and $t$ to the (formal) variables and constants? How does this process give a "meaning" to the variables and constants?

Why do we need the $f$ and $t$? Where do they come from? Aren't these two symbols, well, just symbols? If so, what do we gain by their introduction and why are the $\perp, \top$ "meaningless" while the $f$ and $t$ are "meaningful"? What is their understood meaning?

These are good questions! Here are some answers:

- The symbols $\perp, \top$ (but also $\neg, \lor, \land, \equiv, \rightarrow$) are "meaningless" in the sense that we know nothing of them besides what the axioms will lead us to know. Their behavior and properties are determined only by the axioms and the rules of writing proofs in Boolean logic.

On the other hand, the symbols $f$ and $t$ (as well as the counterparts of $\neg, \lor, \land, \equiv, \rightarrow$, namely, the $F_{\neg}, F_{\lor}$ etc.--see the table in 1.3.4) are directly given via tables in the metalanguage, as part of the elementary "Boolean algebra" that we learn in computer programming. It turns out the properties of the latter faithfully track the properties of the former, and thus in a natural way provide a "concrete" interpretation of the former.

- An analogy may shed more light on the above discussion: Think of axiomatic Euclidean geometry. There we learn the properties of, and interrelationships between, the "meaningless" concepts of point, line and plane by rigorous proofs that are based on the axioms and proof-writing rules, but on nothing
else. Yet, there is another, “naïve”, kind of geometry, analytic geometry. In this geometry, a “point” is not something abstract whose properties we wait to learn from axioms; rather, it is a concrete, well-understood object of mathematics: an ordered pair of real numbers, \((x, y)\). Similarly, a (planar) “line” is an algebraic expression on two variables \(x\) and \(y\) and real coefficients \(a, b, c\): \(ax + by = c\). One finds that properties of the abstract (or “meaningless”) points and lines of the axiomatic version are faithfully tracked by those of the corresponding concepts of the “naïve” versions. In this sense, the points and lines of the latter provide a “concrete” interpretation of the points and lines of the former.

- But why interpret? Because even if we are determined to write proofs solely based on axioms and rigid rules of logic, it aids our motivation and ability to formulate such proofs if we have a “concrete” counterpart in mind. For example, another interpretation of axiomatic geometry is that of geometric drawings, “figures” as we say. You will recall from your high school years how helpful these figures were in your formulation of proofs in geometry. The geometer M. Pasch once wrote a totally figureless monograph on axiomatic geometry, presumably to emphasize that the figures we draw in geometry are only intuitive visual aids that are theoretically redundant. Yet, it seems that the human brain does well with some sort of assistance, be it visual or some other well-understood concrete representation of abstract objects, toward understanding and formulating abstract arguments.

- Analogously with the above, we are often motivated and aided by our knowledge of informal Boolean algebra—that is, the set \(\{f, t\}\) and the various operations on it as introduced in 1.3.4 below—as we construct proofs in axiomatic logic.

- Interpretations of abstract concepts by concrete ones provide a powerful tool toward building counterexamples: The “faithful” nature of such interpretations means that if I can prove a statement involving abstract concepts, let us call it \(A\), then its concrete counterpart, let us call it \(A'\), is also verifiable.

Turning this around I get a very useful observation: If I believe that \(A\) is not provable, I can offer indisputable evidence for this provided I can show, in the concrete domain, that \(A'\) is false. This comment will make more sense later, in 3.1.5.

- *Hmm*. It appears that this discussion builds too strong a case for the import of the “naïve” or “concrete” approach, even though early on (p. 4) I said that the abstract (synactic) approach will be favored in this book. I quote:

> We will learn that correctly written proofs are finite and “checkable” means toward discovering mathematical “truths”. We will also learn via a lot of practice how to write a large variety of proofs that verify all sorts of useful truths of mathematics.

> The above task, writing proofs—or “programming in logic” if you will—is our main aim.
Yet, we noted that the concrete methods track the abstract (axiomatic) approach faithfully. They provide motivation and aid the construction of proofs. They provide definitive evidence regarding the falsehood of mathematical statements. Then why bother with the axiomatic approach? When it comes to logic, would it not be best to work exclusively with Boolean algebra instead?

There are a number of reasons why the axiomatic approach has attained a prominent status, even in the undergraduate curricula:

(a) There is more to logic (predicate logic of Part II) that cannot be tracked by Boolean algebra. For predicate logic the concrete counterparts are in general infinitary, i.e., deal with infinite sets and operations with infinitely many arguments, such as searching an infinite set to determine if an object belongs to it.

By contrast, syntactic proofs of the axiomatic method continue to be finite processes. Where the advantage lies is clear!

(b) The axiomatic method was introduced as a mind-focusing device: Focus on what matters, via axioms and rules, and discard all that is extraneous to our assumptions. The approach literally saved mathematics from the paradoxes that the purely concrete, or naïve, set theory of Cantor introduced.

(c) The axiomatic method makes logic—and any theory that we build upon logic, e.g., modern set theory, Peano number theory—a mathematical object, just as a programming language is a mathematical object. This allows us to use mathematical tools to study logic—and any mathematical theories built upon it—as to its power, limitations, freedom from contradiction, etc.

1.3.4 Definition. (Truth Tables) There are five operations or functions, the Boolean functions, that take as inputs only values from the set \{f, t\} and produce as outputs only values in the same set. The symbols we choose for these functions, one symbol for each Boolean connective, are

\[ F_\& (x), F_\lor (x, y), F_\rightarrow (x, y), F_\neg (x, y), F_\equiv (x, y) \]

and their behavior is fully described by the following table, known as a truth table.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>( F_&amp; (x) )</th>
<th>( F_\lor (x, y) )</th>
<th>( F_\rightarrow (x, y) )</th>
<th>( F_\neg (x, y) )</th>
<th>( F_\equiv (x, y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

The following definition extends a state \( \nu \) so that it can give a Boolean value, hence meaning, to all formulae. Note that originally \( \nu \) gave a value only to atomic formulae.
In essence, the definition gives meaning to the Boolean connectives, as it is clear from the fact that there is a case for each connective of how to compute the value.

Pedantry requires that the extension of \( v \) below be denoted by a different symbol, say \( \bar{v} \), since, after all, the extension is a much bigger table. While the original had a row only for every atomic formula, the extension has a row for every formula. But now that you know about this quibble, we feel safe to use the same symbol, "\( v \)", for both.

1.3.5 Definition. (Value of a Formula in a State \( v \)) Below I use the metavariable "\( p \)", so that the "first" equation actually represents infinitely many, one for each variable in the alphabet \( \mathcal{V} \):

\[
\begin{align*}
v(p) & = \text{whatever we originally assigned to } p; \text{ } t \text{ or } f \\
v(\top) & = t \\
v(\bot) & = f \\
v(\neg A) & = F_\neg v(A) \\
v(A \land B) & = F_\land v(A), v(B) \\
v(A \lor B) & = F_\lor v(A), v(B) \\
v(A \rightarrow B) & = F_\rightarrow v(A), v(B) \\
v(A = B) & = F_= v(A), v(B)
\end{align*}
\]

The symbol "\( = \)" above is the "equals" sign of the metatheory, and means that the left-hand side and right-hand side values are the same (equal). It is not a formal symbol for (at least) two reasons:

1. Our \( \mathcal{V} \) does not include "\( = \)"
2. The definition above compares informal (metatheoretical) values (\( t \) and \( f \)).

Why the above definition works is clear at the intuitive level: Lack of ambiguity in decomposing a nonatomic formula \( G \), i.e., uniquely, as one of \( (\neg A), (A \land B), (A \lor B), (A \rightarrow B), (A = B) \) allow us to know how to compute a unique answer. The why at the technical level is beyond our reach (the demanding reader can find a proof in [54]).

The convenience of truth tables can be extended to rephrase the recursive equations in the above definition (from the 4th equation onward). For example, the 5th equation is represented in table form as

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( A \land B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( f )</td>
<td>( f )</td>
</tr>
<tr>
<td>( f )</td>
<td>( t )</td>
<td>( f )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
<td>( f )</td>
</tr>
<tr>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
</tbody>
</table>
The way we read the above is that for all possible values of the not necessarily atomic formulae \( A \) and \( B \) we have listed the correct value of \( A \land B \). We do not care how \( A \) and \( B \) actually obtained their values. Indeed, we do not care how \( A \) and \( B \) are built: they can be as complex as they like.

Here is an example of a definition of a “concept” regarding formulae, by induction on formulae.

1.3.6 Definition. (Occurrence of a Variable) We define “\( p \) occurs in \( A \)” and “\( p \) does not occur in \( A \)” simultaneously:

Occ1. (Atomic) \( p \) occurs in \( p \). It does not occur in any of \( q, \top, \bot \)—where \( q \) is a variable distinct from \( p \).

Occ2. \( p \) occurs in \( \neg A \) iff it occurs in \( A \).

Occ3. \( p \) occurs in \( (A \circ B) \)—where \( \circ \) is one of \( \land, \lor, \rightarrow, \equiv \)—iff it occurs in \( A \) or \( B \) or both.\(^{33}\)

\[ \square \]

1.3.7 Remark. We wanted to be user-friendly (which often means “slippery”) in the first instance, and said that the above defines a “concept”: “\( \text{Occurs}/\text{Does not occur} \).” In reality, all such “concepts” that we may define by recursion on formulae are just functions.

For example, this “concept” can be captured by the function “\( \text{occurs}(p, A) \)”, where \( \text{occurs}(p, A) = 0 \) means “\( p \) occurs in \( A \)”, and \( \text{occurs}(p, A) = 1 \) means “\( p \) does not occur in \( A \).”\[ \square \]

1.3.8 Remark. (Finite “Appropriate” States) A state \( v \) is by definition an infinite table. Intuitively, the value of a formula \( A \) in any state \( v \) should depend only on the values of the variables that occur in \( A \) and on no others. Thus, for any one \( A \), the state could be truncated into a finite table “appropriate” just for \( A \)—defined on all the variables of \( A \) but undefined elsewhere—without altering the Boolean value of \( A \). Such a table would have one row for each variable that occurs in \( A \), plus the two rows for \( \bot \) and \( \top \)—which in the end can be omitted as they offer no surprises.

Our intuition is correct. Here is a proof by induction on \( A \) of the relevant statement:

If \( v \) and \( v' \) are two states that agree on the variables of \( A \), then \( v(A) = v'(A) \), where “\( = \)” is metamathematical equality on the set \( \{ \bot, \top \} \).

The proof, as it must, goes back and forth among Definitions 1.3.6 and 1.3.5 while, tacitly, it is also mindful at all times of how formulae are formed, going from less to more complex ones (Definitions 1.1.5 and 1.1.10).

- Basis. If \( A \) is atomic, then either
  1. It is a constant, and hence \( v(A) = v'(A) \) by Definition 1.3.5, equations two and three.

or

\(^{33}\)Needless to say, by the “iff”, it does not occur exactly when we have both: It does not occur in \( A \) and it does not occur in \( B \).
(2) It is $p$. By assumption, $v(p) = v'(p)$. Okay!

Complex formulae have two main shapes:

*Case where $A$ is $\neg C$: * The I.H.$^{34}$ applies to $C$. We use it as follows:

1. Since (1.3.6) $C$ and $\neg C$ have precisely the same variable occurrences, it is that $v$ and $v'$ agree on the variables of $C$.

2. By I.H. we get $v(C) = v'(C)$, and so $v(\neg C) = v'(-C)$ by 1.3.5.

*Case where $A$ is $C \circ D$: * By Definition 1.3.6 (Occ3), each of $C$ and $D$ have all their variables also occur in $C \circ D$; therefore, by assumption, $v$ and $v'$ agree on all the variables that occur in $C$ and $D$. By I.H. $v(C) = v'(C)$ and $v(D) = v'(D)$. By 1.3.5, $v(C \circ D) = v'(C \circ D)$. This concludes the proof. □

1.3.9 **Definition.** (Tautologies) Boolean logic is primarily interested in those formulae that are true (t) in all possible states. Such formulae are called tautologies and because of their "shape", i.e., the way they are put together from atomic formulae, brackets, and connectives, are always true. We use the shorthand notation $\equiv_{\text{true}} A$ to indicate that $A$ is a tautology.

In view of 1.3.8, when checking a formula $A$ for tautology status, we need to check it only on all finite states appropriate for $A$. If and only if we find that its value in all those states is $t$, then it is a tautology. Thus there is a finite process to do this, however, one that at the present state of the art is ridiculously inefficient: To so check an $A$ that has $n$ Boolean variables (occurring in it) we need a truth table of $2^n$ rows. Current research on the "P = NP?" question of Cook ([5]) converges toward the opinion that it is highly unlikely that we will ever have a way to check tautologyhood, deterministically, in a manner appreciably more efficient than constructing a truth table.

This is one additional reason why we are interested in discovering tautologies in a different way, nondeterministically, one that allows shortcuts in the calculation by allowing the prover to guess the correct next step from a set of candidates whenever such a choice is offered, thus avoiding having to check all possible avenues that offer themselves. These guesses, when possible, are informed by experience and human intuition and ingenuity and shorten the process of tautology verification. Enter (syntactic) "proofs" of the next section.

1.3.10 **Example.** (Some tautologies) $\top$ and $p \rightarrow p$ are tautologies. The latter follows from $v(p \rightarrow p) = F,(v(p), v(p)))$ and the truth table on p. 29.

How about $p \rightarrow q \rightarrow p$? First, remember that this is sloppy for $(p \rightarrow (q \rightarrow p))$. Thus the last connective to act is the leftmost.

$^{34}$ The I.H. is invariably "assume the claim for all formulae less complex than $A$". As such, it deserves no explicit mention.

$^{35}$ It is not known whether there is a fast nondeterministic algorithm that verifies every tautology. But even if we discover one such, it is unlikely in view of what I said above that we can eliminate the nondeterminism without significant speed loss.
Here's the general technique: The table below has two components. The state-part consists of the first two columns. Each row of these columns is a finite state appropriate for the formula.

The value of the formula in each state is found beneath the last-to-act connective in the process of 1.1.3. To compute this value, we use the values \( v(p) \) and that of \( q \rightarrow p \). The values of the latter are aligned under the relevant connective, the "\( \rightarrow \)" of the subformula \( q \rightarrow p \). In general, in the value part we align the values that we compute under the connective that acted last in the subformula we are evaluating.

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
<th>p \rightarrow q \rightarrow p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

The number headings (1)–(3) above indicate the order in which the columns are built. By the way, we have just verified that \( \models \text{mut} \ p \rightarrow q \rightarrow p \).

What about \( A \rightarrow B \rightarrow A \) where \( A, B \) are arbitrary formulae? Can we settle this question without knowing the particular ways that \( A, B \) are put together from variables, connectives, and brackets?

Yes! No matter how they are put together, in any state \( e \) we have that each of \( A, B \) attain one of two values \( t \) or \( f \). Thus the exact same table as the above, but this time using \( A, B \) and \( A \rightarrow B \rightarrow A \) as column headings.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A \rightarrow B \rightarrow A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

settles the question: We have \( \models \text{mut} \ A \rightarrow B \rightarrow A \).

We must be sure to read the above table correctly. We are not saying that we are assigning values to \( A \) and \( B \) (we can assign values only to variables and constants). We are saying that the possible pairs of values of \( A \) and \( B \)—in that order—no matter what state we are in, will be among the four listed in the table. Then using Definition 1.3.5 we fill in the last two columns of the table.

There are three more concepts related to tautologies that we want to introduce, but first some notation:
We use the metavariables \( p, q, r, q_0, \ldots \) for variables, and \( A, B, E, Q \) for formulae. What shall we use for sets of formulae?

**Convention:** We denote sets of formulae by certain capital Greek letters, as a rule, by those that **cannot** be confused with Latin letters. Thus \( \Gamma, \Delta, \Sigma, \Theta \) will always stand for sets of formulae (such sets may have zero, one, two, three, one million, or infinitely many members). Of course, we use these letters to denote such sets if either

(i) We do not care what are the members of a set of formulae

(ii) We do care, but we are going to refer to that set over and over again in an argument; thus rather than, say, writing \( \{ \bot, p, p \rightarrow q, p \rightarrow \neg q \} \) over and over again, we may give it a name saying, "Let \( \Sigma \) stand for \( \{ \bot, p, p \rightarrow q, p \rightarrow \neg q \} \)."

By the way, we must **not** confuse the set \( \{ A \} \) with its member \( A \). These are different. Think in terms of types (as in programming languages): \( \{ A \} \) is an object of type set while \( A \) is an object of type formula.

1.3.11 Definition. A formula \( A \) is **satisfiable** iff there is at least one state \( \nu \) where \( \nu(A) = t \). A set of formulae \( \Gamma \) is satisfiable iff there is at least one state \( \nu \) where for every formula \( A \) in \( \Gamma \), \( \nu(A) = t \). We say that \( \nu \) satisfies \( \Gamma \).

We say that \( \Gamma \) **tautologically implies** \( A \) and write \( \Gamma \models \text{true} A \) iff for every state \( \nu \) that satisfies \( \Gamma \) we must have \( \nu(A) = t \). We call \( \Gamma \) the **hypotheses** (plural, in general) or **premises** of the implication, while \( A \) is the **conclusion**.

We say that a formula \( A \) is **unsatisfiable** or a **contradiction** iff for every state \( \nu \), we have \( \nu(A) = f \). We say that a set \( \Gamma \) is unsatisfiable iff for every state \( \nu \) there is at least one \( A \) in \( \Gamma \) such that \( \nu(A) = f \). 

By **convention**, logicians write the simpler \( A \models \text{true} B \) for the correct \( \{ A \} \models \text{true} B \), and more generally, prefer to write \( A_1, \ldots, A_n \models \text{true} B \) rather than the correct (but pedantic) \( \{ A_1, \ldots, A_n \} \models \text{true} B \).

Note that intuitively, a tautology \( A \) is true, no questions asked, and **must** be accepted as such. On the other hand, the **conclusion** of a tautological implication is only **relatively true**, relative to the premises, that is: If we accept the premises as true, then we **must** also accept the conclusion as true.

This **relativity of truth** is at the heart of mathematics. For example, if we accept Euclid's "5th postulate"\(^{37}\) as true, then we **must** accept that the sum of the angles of **any** triangle equals 180 degrees. This is a relative **truth**,\(^{18}\) since Euclid's 5th postulate is not an absolute truth. Accepting any one of its possible negations\(^{39}\) leads to a totally different (relative) truth regarding the sum of angles in a triangle.

\(^{36}\) We may also say, "Let \( \Sigma = \{ \bot, p, p \rightarrow q, p \rightarrow \neg q \} \)."

\(^{37}\) It states that through a point that lies outside a given line it is always possible to draw a unique line parallel to the given one.

\(^{38}\) Strictly speaking, this example lies beyond Boolean logic, since we need predicate logic in order to "speak" and do Euclidean geometry. Nevertheless it illustrates the phenomenon of relativity of truth in a familiar branch of mathematics.

\(^{39}\) One possible negation is that you can draw two parallels (Lobachevsky). Another is that you can draw no parallels at all (Riemann).
1.3.12 Example. We can in principle verify $A_1, \ldots, A_n \vdash_{\text{mut}} B$ via a truth table

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\ldots$</th>
<th>$A_1$</th>
<th>$\ldots$</th>
<th>$A_n$</th>
<th>$\ldots$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a state</td>
<td>all t</td>
<td>must be t too</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We look only at those rows that have t everywhere between the two “$\mid$” vertical dividers. We must ensure that these rows have a t under $B$ as well. By the way, the $A_i$ and $B$ being, in general, complicated formulae, we need the “$p, q, \ldots$” columns to the left of the leftmost $|$ in order to compute the values of the $A_i$ and $B$ in all states.

In general, when checking $A_1, \ldots, A_n \vdash_{\text{mut}} B$, one cannot avoid building the whole truth table (or doing some equivalently laborious task) for the following reasons:

1. As we have said, at the present (and foreseeable) state of the art there is no way to check whether $A$ is a tautology any more efficiently than it takes to build the whole truth table, in terms of the variables $p, q, \ldots$, of $A$.

2. If we had substantially faster algorithms for tautological implication, we could use it to also establish tautologyhood fast, since $\vdash_{\text{mut}} A \iff \bot \vdash_{\text{mut}} A$.

However, in many cases of practical interest we can do better since we need to only check the rows that have exclusively t’s between the two $\mid$ dividers, and ignore all the other rows. Sometimes we can identify these rows without building the whole table.

For example, we can quickly see that $A, B \vdash_{\text{mut}} A \land B$ since whenever $v(A) = v(B) = t$ we have $v(A \land B) = t$ (cf. 1.3.5). We did not compute the other three rows. A more substantial example is

$$A \lor B, \neg A \lor C \vdash_{\text{mut}} B \lor C$$

If this is done by the full table method we need 8 rows to compute the values of $A \lor B, \neg A \lor C, B \lor C$ for all possible ordered triples of values $v(A), v(B), v(C)$.

But instead, let us be clever: Okay; we merely need to compute the value $v(B \lor C)$ on the condition that

$$v(A \lor B) = v(\neg A \lor C) = t$$ \hspace{1cm} (1)

We analyze (1) according to two cases:

(i) $v(A) = t$: By (1) and 1.3.5, $v(B) = t$. But then $v(B \lor C) = t$ by 1.3.5.

(ii) $v(A) = f$: By (1) and 1.3.5, $v(C) = t$. But then $v(B \lor C) = t$ by 1.3.5. \hfill $\Box$

1.3.13 Example. Here is a really important example: $\bot \vdash_{\text{mut}} A$. Well, I need to ensure that for every $v$, where $v(\bot) = t$, I also get $v(A) = t$.

Since no $v$ satisfies $v(\bot) = t$—i.e., there are no rows to check—I am done.

We have seen such situations before. The statement is vacuously true. Like before, one can explain this by saying, "The only way to refute $\bot \vdash_{\text{mut}} A$ is to find a $v$ where $v(\bot) = t$ but $v(A) = f$. But such a task must fail as no $v$ satisfies $v(\bot) = t$." \hfill $\Box$
Here is a simple but nice exercise that you must try:

1.3.14 Exercise. (1) Show that \( \vdash_{\text{int}} A \iff \vdash_{\text{int}} \bar{A} \).

(2) Show that \( A_1, \ldots, A_n \vdash_{\text{int}} B \iff \vdash_{\text{int}} A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B \).

(3) Show that \( \Gamma \vdash_{\text{int}} B \) iff \( \Gamma \cup \{-B\} \) is unsatisfiable. We note here that for two sets \( \Gamma \) and \( \Sigma \) the notation \( \Gamma \cup \Sigma \)—called the union of \( \Gamma \) and \( \Sigma \)—denotes the set that contains every member of \( \Gamma \) and every member of \( \Sigma \).

We conclude this section with the definition of substitution in a Boolean expression, and with the proof of two important properties of substitution.

Intuitively, the symbol \( \"A[p := B]\" \) is shorthand that means—i.e., expands into—the string we get if we replace all occurrences of the variable \( p \) in \( A \) by the formula \( B \). We may think of this operation as defining a function from formulae to strings:


In order to read the following definition of substitution correctly, we emphasize that the "operation" \( [p := B] \) takes place in the metatheory and has the highest priority against all other "formal operations", i.e., the Boolean connectives \( \neg, \land, \lor, \rightarrow, \equiv \). For example, \( \neg A[p := B] \) means \( \neg \{ A[p := B] \} \), where the symbols \( \"\{, \}\" \) are here meta-brackets inserted to indicate the order of application of "operations". Naturally, because of its placement, this operation is left-associative, so that \( A[p := B][q := C] \) means \( \{ A[p := B] \}[q := C] \).

1.3.15 Definition. (Substitution in Formulae) In what follows, \("=\" \) denotes equality in the metatheory, here between strings. The definition states the obvious: (1) It handles the basic cases in the trivial manner, and (2) when \( A \) is actually built from i.p. (cf. 1.1.10), it says that we substitute into each i.p. first, and then apply the connective.

As we usually do in definitions we are careful not to use formula abbreviations. All brackets are present. Note the use of metavariables.

\[
A[p := B] = \begin{cases} 
B & \text{if } A = p \\
A & \text{if } A = q \text{ (where } p \neq q), \text{ or} \\
& A = T, \text{ or } A = \bot \\
\{ C[p := B] \} & \text{if } A = \neg C \\
\{ C[p := B], D[p := B] \} & \text{if } A = (C \circ D) 
\end{cases}
\]

where \( \circ \) is one of \( \land, \lor, \rightarrow, \equiv \).

We state and prove two easy and hardly unexpected properties of substitution:

1.3.16 Proposition. For any formulae \( A \) and \( B \) and variable \( p \), \( A[p := B] \) is a (well-formed!) formula.\(^{40}\)

---

\(^{40}\) A *proposition* is a theorem—here, metatheorem—that did not quite make it to be called that. You see, people reserve the term theorem, or metatheorem, for the "important" or earth-shattering stuff that
Proof. Induction on $A$, keeping an eye on Definition 1.3.15.

**Basis.** If $A$ is atomic, then we get either $A$ or $B$ formula; okay in either case.

Complex formulae have two main shapes:

- $A$ is $(-C)$: The I.H. applies to $C$, thus $C[p := B]$ is a formula. Then so is $\neg C[p := B]$ by 1.1.7.

- $A$ is $(C \rightarrow D)$: The I.H. applies to $C$ and $D$, thus $C[p := B]$ and $D[p := B]$ both are formulae. Then so is $(C[p := B] \rightarrow D[p := B])$, again by 1.1.7. \(\square\)

1.3.17 Proposition. If $p$ does not occur in $A$, then $A[p := B] = A$, where for convenience we once more use "=" as metamathematical equality of strings.

Proof. Again, by induction on formulae $A$:

**Basis.** $A$ can only be one of $q (q \neq p)$, $\top$, or $\bot$, by Definition 1.3.6. Then, by Definition 1.3.15 (2nd case), $A[p := B] = A$. Okay, so far.

Complex formulae have two main shapes:

- $A$ is $(-C)$: Does $p$ occur in $C$? No, by assumption (it does not occur in $A$) and definition of occurrence (1.3.6). Now, the I.H. applies to $C$, thus $C[p := B] = C$. We are done by 1.3.15, case 3.

- $A$ is $(C \rightarrow D)$: By 1.3.6 (Occ3) $p$ occurs neither in $C$ nor in $D$. The I.H. applies to both these subformulae; thus $C[p := B] = C$ and $D[p := B] = D$, and we are done by 1.3.15, case 4. \(\square\)

1.4 PROOFS AND THEOREMS

We are ready to develop a calculus that we may use to write down theorems. We will learn to "calculate theorems" just as we have learned to "calculate" ("parse") formulae.

Boolean logic is a (crude) vehicle through which we formulate and establish mathematical truth. This truth is captured absolutely (tautologies) or relatively to certain *premises* (tautological implications). Thus, when we do Boolean logic, our main task is to discover and verify tautologies, and more generally, to discover and verify tautological implications.

We have already remarked that the presently known mechanical ways to check for tautology status as well as to verify tautological implications are hopelessly inefficient, and that there is every indication that an efficient tautology (or tautological implication) checker will never be discovered.

One turns to utilizing human ingenuity and experience—in other words, utilizing (educated) *guessing*—toward effecting serious shortcuts in the process of certifying tautologies and tautological implications. This guessing (or "nondeterministic")

we prove. All else that we prove are just *propositions* (or *lemmata*—singular: *lemma*) if they have just "auxiliary status", just like FORTRAN subroutines; or they are *corollaries*, if they follow trivially—more or less—from earlier results.
process of certifying tautologies and tautological implications is **syntactic** rather than truth table driven\(^4\) (semantic) and is called **theorem proving**.

The theorems that we will learn to prove with this new syntactic technique will be either absolute truths (tautologies) or truths that are relative (conclusions of tautological implications) to certain assumptions that we have accepted.

Our major concern as we are founding this syntactic proof calculus will be to ensure that whatever tools we utilize are capable of certifying all absolute and relative truths, and only those. That is, these tools will never "certify" a falsehood as a "theorem". Our degree of success in implementing these requirements will be assessed later (Section 3.1 will assess the promise for only those, while 3.2 will assess the one for all).

As I said before, I kind of like the terms calculate theorems and theorem calculus as they remind us that proving theorems is a precise syntactic (synonym for formal, i.e., depending only on form) algorithmic process. This observation is the origin of the alternative name of **equational logic** of \([11]\): calculational logic. However, most logicians and mathematicians would proclaim that they "proved" (rather than "calculated") a theorem and would rather call a theorem-calculus a **proof**.

First off, a theorem-calculation or **proof** is a finite sequence of formulae, entirely analogous to formula-calculations. Each formula occurring in a proof will be called a theorem.

The specifications of formula-calculations are the following two:

1. A formula-calculation must start with the **simplest possible kind** of formula: one that is "primitive", i.e., atomic.
2. Every **operation that extends** the formula-calculation must preserve the property "formula": either it is the trivial operation of writing down an atomic formula, or it is an operation that acts on previously written formulae and produces a formula.

**Entirely analogously**, as it flows from the preceding discussion in this section, the specifications of theorem-calculations are the following two:

1. A theorem calculation must start with the writing down of a formula that is among the simplest possible theorems— a "primitive theorem" for which the validation process is simply to write it down! We call such a formula an **axiom**.
2. Every **operation that extends** the theorem-calculation must preserve the property "theorem"; thus, it is either the trivial act of writing down another axiom, or it is applied to already-written theorems, resulting into a new theorem. Since a theorem calculation must **certify truth**, these nontrivial operations, or rules, must preserve truth. Technically, whenever they are applied to formulae \(A_1, \ldots, A_n\) and yield a formula \(B\) as a result, it is necessary that they obey \(A_1, \ldots, A_n \vdash \text{that} \ B\).

We have two types of axioms: The **logical axioms** are certain well-chosen\(^5\) absolute truths; therefore, they are certain tautologies. The other type we will call **special axioms**, but also **assumptions or hypotheses**. These are not fixed outright, but

---

\(^4\) Later there will be a weakening of this somewhat dogmatic statement.

\(^5\) The qualifier well-chosen will be revisited later.
may change from discussion to discussion. They are not deliberately chosen to be absolute truths, but are formulae that we "accept as true" (cf. discussion on relativity on p. 34) simply because we are interested to explore what sort of (tautological) conclusions we may draw from them.

In intuitive terms—since the quest for theorems is the quest for "mathematical truths", absolute or relative—the axioms are our initial truths. Our rules of reasoning will allow us to derive further truths from, and relative to, the axioms.

The nontrivial operations that lengthen proofs (cf. (2) above) are called rules of inference. To achieve the purely syntactic character of proofs, the rules of inference are applied in a manner that the input/output relation is purely syntactic. For example, one of the two primitive rules, soon to be introduced, applies to any formulae of the forms $A$ and $A \equiv B$ and "outputs" $B$. The rule does not care about which specific formulae $A$ or $B$ stand for, nor about the semantics of any of $A$, $B$, $A \equiv B$. The only thing that the rule cares about is that it "sees" as input an equivalence on one hand, and the first formula of this equivalence on the other. It immediately "knows" that it must "output" the second formula of the equivalence.

In order to describe the rules of inference, we need formula schemata (or, simply, schemata).

A schema is a string in the metalanguage (i.e., outside logic) over the augmented alphabet that along with $\mathcal{V}$ (p. 9) includes the symbols "[", ":="", and "]", and all the syntactic variables for formulae and Boolean variables.

The syntactic structure of a schema is, by definition, such that if we replace all the syntactic variables that occur in it by any specific formulae and variables, as appropriate, then the result names a formula of WFF.

1.4.1 Definition. (Schema Instance) An instance of a schema is the formula we obtain if we replace all its metavariables with specific objects (formulae/Boolean variables) as appropriate.

Here are some examples of schemata:

1. $A$: It is a formula-metavariable; if we replace the letter "$A$" by some formula, well, we get that formula as the result!

2. $(A \equiv B)$: This schema has two formula-metavariables, $A$ and $B$. Whatever formulae we may replace $A$ and $B$ with, we get a formula by 1.1.7.

3. $A[p := B]$: This schema has two formula-metavariables, $A$ and $B$, and a Boolean metavariable, $p$. Whatever formulae we replace $A$ and $B$ with, and whichever Boolean variable replaces $p$, we get a formula by 1.3.15.

---

43 For this reason, I suppose, some people call them temporary assumptions. However, temporary is not a technical term. For example, Euclid's 5th postulate has no expiry date. Yet it is not an absolute truth.

44 The qualification "chosen" is picked purposefully: I do not want to rule out as special axioms any that, either by accident or on purpose, have been chosen to be tautologies. Analogously, when we defined the notation $\Sigma \equiv_{\text{ex}} A$, quite correctly we did not forbid the case where some formulae of $\Sigma$ may be tautologies.

45 We will encounter many derived rules. These are not "given" or postulated up front, but we prove their validity.

46 Schema, plural schemata (and, incorrectly but often, schemas), is Greek for form and figure.
We often write the rules of inference as “fractions”, like

\[
\frac{P_1, P_2, \ldots, P_n}{Q}
\]

where all of \( P_1, \ldots, P_n, Q \) are formula schemata. We call the “numerator” the premise (case \( n = 1 \)) or premises (case \( n > 1 \)) and the “denominator” the conclusion or result of the rule. Instead of premises we also say hypotheses or assumptions.

The \( P_i \) and \( Q \) are syntactically related so that one can mechanically check this input/output relation by simply looking at the form of the \( P_i \) and \( Q \).

We have already noted an example of this mechanical applicability of a rule such as

\[
\frac{A, A \equiv B}{B}
\]

The input/output relation of a rule need not be “functional”; that is, the result of a rule need not be uniquely determined by the hypotheses (cf. Leibniz rule below).

It is obvious why a rule is expressed in terms of schemata rather than specific formulae. Schemata allow a rule to be applicable to infinitely many formulae. If conclusion and premises were specific formulae, then there would be just one case where the rule would be applicable, which would hardly qualify it to be called a rule, a term that creates the expectation of applicability to a vast number of cases. Here is an analogy: The input/output relation on numbers “in:3 / out:9” is not a rule, but “in:x / out:x^2” is.

How a rule like \((R)\) above is applied will be clear in Definition 1.4.5.

Let us finally introduce the two primitive rules of Boolean logic that we will adopt in this volume.

1.4.2 Definition. (Rules of Inference) The following two are our primitive or primary rules of inference, given with the help of the syntactic variables \( A, B, p, C \):

\[
\text{Inf1}
\]

\[
\frac{A \equiv B}{C[p := A] \equiv C[p := B]}
\]

\text{(Leibniz)}

\[
\text{Inf2}
\]

\[
\frac{A, A \equiv B}{B}
\]

\text{(Equanimity)}

An instance of a rule of inference is obtained by replacing all the letters \( A, B, C \) and \( p \) by specific formulae and a specific variable respectively.

The rule names conform to those in [17].

1.4.3 Remark. (1) Why primary? Do we also have “secondary rules”? Yes. We will soon learn that we can apply additional rules in a theorem-calculation, which are not mentioned in the definition of proof below (1.4.5) because they are not theoretically
necessary toward defining proof and theorem. Such additional rules are not to be arbitrarily added to our toolbox without question. Instead, we will show before adding them, via a rigorous mathematical argument, that we are allowed to use them. This "allowed" means that there is nothing we can prove using these additional rules that we cannot prove without them. We call such additional rules derived rules, or secondary rules.

Compare, once again, with programming languages. Some general-purpose procedural languages were designed not to contain the instruction goto because it was considered "harmful" by some influential computer scientists in the 1970s (cf. [10], which, arguably, started it all)—but not by all (cf. [28]). Nevertheless, one can prove that goto can be simulated by the originally given instructions. It is a "derived" kind of instruction in those goto-less languages.

Harmful or not, one will agree that adding one more tool does add to convenience, in general.

(2) Other than the "restriction" that the \( A, B, C \) and \( p \) are metavariables of the agreed-upon kind, there is no other restriction on the letters. In particular, we have no assumption on whether \( p \) actually occurs in \( C \) of the Leibniz rule. Either way, it is all right.

(3) We have already discussed the mechanical nature of \textbf{Inf2}. That also \textbf{Inf1} is mechanically applicable is clear: Once we have written \( A \equiv B \), we can pick any formula \( C \) whatsoever and any variable \( p \) and construct the output, first effecting two substitutions and then connecting the results with the connective \( \equiv \) in the indicated order.

Note that the Leibniz rule is not functional: Infinitely many different outputs are possible for a given input \( A \equiv B \).

We now turn to our choice of axioms.

\textbf{Schemata: again.} We noted already that the axioms will be "initial truths", and as such they will be selected tautologies.

Suppose then, for the sake of discussion, that one of the tautologies of choice to attain axiom status is \( p \equiv p \).

But how about \( q \equiv q \)? Or how about \( p \lor p' \equiv p \lor p' \) and, indeed, how about \( (p \lor p' \equiv p \lor p') \equiv (p \lor p' \equiv p \lor p') \)?

All these have the same form, namely \( A \equiv A \), where \( A \) stands for an arbitrary formula. Naturally, if we want to include \( p \equiv p \), then we will want to include all those tautologies that have the same "shape", \( A \equiv A \), as well, since surely they all state the same (absolute) principle: "Every statement is equivalent to itself." If this is a "truth" worth postulating, then it would be absurd to do so just for one of its special cases, just for the variable \( p \).

There are two main ways to achieve this generality:

(1) The "modern" and rather obvious way: Rather than saying, "include \( p \equiv p \) and \( q \equiv q \) and \( p \lor p' \equiv p \lor p' \) and \( (p \lor p' \equiv p \lor p') \equiv (p \lor p' \equiv p \lor p') \) and ...", we say, "include the schema \( A \equiv A \)."

This means include all instances of \( A \equiv A \) (cf. 4.4.1).
(2) The "old" way: "Include just $p \equiv p$; however, add a new primary rule of inference called substitution." This rule

$$
\frac{A}{A[p := B]} \tag{Sub}
$$

when applied to the formula "$p \equiv p$" (that is, take $A$ to be $p \equiv p$ and $p$ to be $p$) will be able to generate, by successive applications, all possible tautologies of the form $B \equiv B$.

There is a catch: Once you add rule (Sub) as a primary rule, you have to awkwardly hedge when writing proofs. The rule cannot be used in a theorem-calculation unless you know that the variable $p$ does not occur in any formulæ that are special axioms. This restriction in turn makes a very useful tool that we will soon obtain and learn to use—the deduction theorem—hard to state and even harder to apply.

Thus, the old way is rightfully abandoned. In particular, we will not have any use for (Sub).\(^4\)

We next present the list of logical axioms for Boolean logic. The list is infinite, but because of the use of schemata it can be presented by a finite table. That is, there are only finitely many different formulæ of tautologies that we need to take as our starting point. These are largely the ones in [17] with a few adjustments.

What the axioms do is to codify the most basic properties of the connectives. The following list both presents (in partially parenthesized notation\(^4\)) and names the axioms.

1.4.4 Definition. (Logical Axioms of Boolean Logic) In what follows, $A, B, C$ denote arbitrary formulæ:

- **Properties of $\equiv$**
  - **Associativity of $\equiv$**
    \[ ((A \equiv B) \equiv C) \equiv (A \equiv (B \equiv C)) \]  \(1\)
  - **Symmetry of $\equiv$**
    \[ (A \equiv B) \equiv (B \equiv A) \]  \(2\)

- **Properties of $\top, \bot$**
  - $\top$ vs. $\bot$
    \[ \top \equiv \bot \equiv \bot \]  \(3\)

- **Properties of $\neg$**
  - **Introduction of $\neg$**
    \[ \neg A \equiv A \equiv \bot \]  \(4\)

- **Properties of $\lor$**

\(^4\)The current edition of [17] uses the old approach in Boolean logic (their Chapter 3) but switches to the modern approach for predicate logic (Chapters 8 and 9). One may assume that by the time the authors decided that the approach with schemata is better it was too late in terms of publication deadlines to rewrite Chapter 3 with schemata.

\(^4\)Recall that brackets associate from right to left.
Associativity of ∨  \[(A ∨ B) ∨ C ≡ A ∨ (B ∨ C)\]  \hspace{1cm} (5)
Symmetry of ∨  \[A ∨ B ≡ B ∨ A\]  \hspace{1cm} (6)
Idempotency of ∨  \[A ∨ A ≡ A\]  \hspace{1cm} (7)
Distributivity of ∨ Over ≡  \[A ∨ (B ≡ C) ≡ A ∨ B ≡ A ∨ C\]  \hspace{1cm} (8)
Excluded Middle  \[A ∨ ¬A\]  \hspace{1cm} (9)

Properties of ∧

Golden Rule  \[A ∧ B ≡ A ≡ B ≡ A ∨ B\]  \hspace{1cm} (10)

Properties of →

Implication  \[A → B ≡ A ∨ B ≡ B\]  \hspace{1cm} (11)

We will reserve the capital Greek letter "lambda", Λ, to denote the set of all logical axioms. This set is, of course, infinite. □

The axioms of Λ, here, however, formulated in their schemata edition, are those that are customary in the "equational" (or "calculational") approach to doing logic, as presented, e.g., in [17] and [32], but with some minor differences—besides the essential one that [17] does not use schemata. For example, [17] uses 1 ≡ A ≡ A instead of (3). Moreover, our choice for (4) is natural, and hence easy to remember: It intuitively says "negating A is tantamount to saying that it is false" (A ≡ ⊥). But [17] adopts instead a different axiom that eventually implies our (4): I quote it, but in schema form, "¬(A = B) ≡ ¬A = B". This being intuitively less clear is also less memorable than the rest.

We are ready to calculate! (Compare with Definition 1.1.3 and Remark 1.1.6(2).)

1.4.5 Definition. (Theorem-Calculations—or Proofs) Let $\Gamma$ be an arbitrary, given set of formulae.

A theorem-calculation (or proof) from $\Gamma$ is any finite (ordered) sequence of formulae that we may write respecting the following two requirements:

In any stage we may write down

Pr1 Any member of Λ or $\Gamma$

Pr2 Any formula that appears in the denominator of an instance of a rule Infl–Inf2

as long as all the formulae in the numerator of the same instance of the (same) rule have already been written down at an earlier stage

We may call a proof from $\Gamma$ by the alternative name $\Gamma$-proof. □

1.4.6 Remark. (1) By definition, a $\Gamma$-proof is a purely form-manipulation construction without any reference to semantic concepts, such as t, f, etc.

(2) We will call $\Gamma$ the set of special axioms (A contains the "general" axioms). Special axioms are also called hypotheses or assumptions. Clearly, while A is reserved and "frozen" in the first part of this book, $\Gamma$ can vary from subject to subject.
(3) Any member of \( \Gamma \) that is not also in \( \Lambda \) we will call a nonlogical axiom.\(^{49}\)

(4) Since any theorem-calculation from some \( \Gamma \) is a finite sequence of formulae, only a finite part of \( \Gamma \) and \( \Lambda \) may appear in such a calculation. This is entirely analogous with what happens in a formula-calculation: Each uses only a finite number of formal variables even though we have an infinite supply of those. \( \square \)

1.4.7 Definition. (Theorems) Any formula \( A \) that appears in a \( \Gamma \)-proof is called a \( \Gamma \)-theorem. We write \( \Gamma \vdash A \) to indicate this. If \( \Gamma \) is empty (\( \Gamma = \emptyset \))—i.e., we have no special assumptions—then we simply write \( \vdash A \) and call \( A \) just "a theorem".

Caution! We may also do this out of laziness and call a \( \Gamma \)-theorem just "a theorem" if the context makes clear which \( \Gamma \neq \emptyset \) we have in mind.

We say that \( A \) is an absolute, or logical theorem whenever \( \Gamma \) is empty. \( \square \)

(1) Clearly, the symbol \( \vdash \) of the metatheory formulates the predicate "is a theorem".

(2) Definition 1.4.7 says that a formula is a theorem on one and only one condition: It occurs in some \( \Gamma \)-proof. We say that such a proof proves \( A \) from \( \Gamma \), or from the hypotheses (of) \( \Gamma \).

In the common parlance of mathematicians we may also say that "\( \Gamma \) derives \( A \)."

(3) Note how in the symbol \( \Gamma \vdash A \) we take \( \Lambda \) for granted and do not mention it to the left of \( \vdash \).

1.4.8 Remark. Thus, a \( \Gamma \)-proof of a formula \( A \) is a sequence of formulae

\[ B_1, \ldots, B_n; A, C_1, \ldots, C_m \]

obeying the requirements stated in 1.4.5. It is trivial that if we discard the "tail" part

"\( C_1, \ldots, C_m \)" of the sequence, then

\[ B_1, \ldots, B_n; A \quad (1) \]

is still a proof. The reason is that every formula in a proof is either legitimized outright—without reference to any other formulae—or is legitimized by reference to formulæ to its left. Thus (1) also proves \( A \), since \( A \) occurs in it. This technicality allows us to stop a proof as soon as we write down the formula that we wanted to prove. \( \square \)

So, 1.4.7 tells us what kind of theorems we have:

1. Anything in \( \Lambda \cup \Gamma \).\(^{50}\)

2. For any formula \( C \) and variable \( p \), the formula \( C[p := A] \equiv C[p := B] \), provided \( A \equiv B \) was written down already, and therefore is a (\( \Gamma \)-) theorem.

\(^{49}\)That is, it does not speak about logic itself: logic does not read this axiom in order to function properly.

\(^{50}\)We explained the notation "\( \Lambda \cup \Gamma \)" in 1.3.14 on p. 36, item (3).
3. \( B \) (any \( B \)), provided \( A \equiv B \) and \( A \) were written down already and therefore are both \((\Gamma-)\) theorems.

Hey! The above is a recursive definition of \((\Gamma-)\) theorems, and is worth recording (compare with Definition 1.1.7).

1.4.9 Definition. (Theorems, Inductively) A formula \( E \) is a \( \Gamma \)-theorem iff \( E \) fulfills one of Th1–Th3 below:

\( \textbf{Th1} \) \( E \) is in \( A \cup \Gamma \).

\( \textbf{Th2} \) For some formula \( C \) and variable \( p \), \( E \) is \( C[p := A] \equiv C[p := B] \), and (we know that) \( A \equiv B \) is a \((\Gamma-)\) theorem.

\( \textbf{Th3} \) (We know that) \( A \equiv E \) and \( A \) are \((\Gamma-)\) theorems.

\[ \square \]

1.4.10 Remark. (Theorem vs. Metatheorem) In the expression \( \Gamma \vdash A \), \( "A" \) is the theorem, \( \textit{not} \ "\Gamma \vdash A" \). After all, a theorem by definition has to be a \textit{single formula} that appears somewhere in some proof (1.4.7).

So what is \( \"\Gamma \sim A\" \) then? It is a \textit{metatheorem}. It is a statement that we are making \textit{about} our logic, \textit{about} what the logic can do, \textit{if} we take all the formulae in \( \Gamma \) as assumptions. It says, \"there is a proof, which from assumptions \( \Gamma \) proves \( A \).\" But how does one establish the validity of such a meta-result as quoted immediately above?

By \textit{proving} within Boolean logic — according to Definition 1.4.5, that is — the theorem \( A \) using assumptions \( \Gamma \).

This action does two things at once: it \textit{proves} \( A \) (from \( \Gamma \)) and it also \textit{metaproves} the statement \( \Gamma \vdash A \).

\textit{Nevertheless, people are mostly shy of such distinctions and it has become acceptable practice to say (by abuse of language) things like "I proved \( \Gamma \vdash A" \ all the time. See also the start-up comment in the next chapter.}\[ \square \]

1.4.11 Exercise. So that you can check your understanding of the concepts \textit{proof} and \textit{theorem}, show (i.e., present proofs) that

(1) \( A \vdash A \), for any \( A \).

(2) A more general form of (1): If \( A \) is a member of \( \Sigma \)— also written \( "A \in \Sigma" \) — then \( \Sigma \vdash A \).

(3) \( \vdash B \), for any axiom \( B \).\[ \square \]

1.4.12 Remark. (Hilbert Proofs) A \( \Gamma \)-proof is also called a \textit{Hilbert proof} after the name of the great mathematician David Hilbert, who essentially was the first serious proponent of the idea to \textit{use logic to do mathematics}. Hilbert also helped to found modern logic in an axiomatic and rigorous setting, and defined the concept of \textit{proof}, essentially as above.\[ \textsuperscript{31} \]

\[ \textsuperscript{31} \] There are some \textit{inesential} differences, Hilbert used a different set of logical axioms, and a single rule of inference.
In practice we write a Hilbert proof vertically on the page, i.e., one formula on top of the other, numbering every formula. It is imperative that we provide annotations to explain what we are doing at every step and why. The numbering assists us in referring to previous formulæ.

All proofs, whether they are in the Hilbert style or in the equational style (the latter style will be introduced shortly), must be annotated.

1.4.13 Example. (Some Very Simple Annotated Hilbert Proofs)

(a) We will verify that \( A, A \vdash B \vdash B \) for any formulæ \( A \) and \( B \). Thus we need to write a formal proof of \( B \) using \( A \) and \( A \vdash B \) as hypotheses (ct. 1.4.10). The part "for any formulæ \( A \) and \( B \)" makes this result applicable to infinitely many instances, one for each specific choice of \( A \) and \( B \). It is therefore a metatheorem schema. We could have said instead, "prove the schema \( A, A \vdash B \vdash B \)"; a formulation where the part "for any formulæ \( A \) and \( B \)" is redundant.

The same comment applies to any theorems (and metatheorems) that we will prove where we have letters to stand for arbitrary formulæ.

Let us establish (a) now. Make sure you memorize the style! This is how you will write your own Hilbert proofs. Every line must be numbered and annotated!

\[
\begin{align*}
(1) & \quad A & \quad \text{(hypothesis)} \\
(2) & \quad A \vdash B & \quad \text{(hypothesis)} \\
(3) & \quad B & \quad \{(1) \text{ and } (2) \text{ and Equanimity)}
\end{align*}
\]

Worth stating. It is clear from Definition 1.4.5 that assumptions can be scrambled. So we have also established \( A \vdash B, A \vdash B \) by the very same proof.

Can we also swap \( A \) and \( B \) in \( A \vdash B \)? It turns out that we can. But the above proof does not address this question: after all, \( B \vdash A \) is never mentioned.

It is a fatal error to say, "but is not \( \vdash \) symmetric? I can see that it is so from the truth table of p. 29:"

No; you see, we have not connected our syntactic proofs with semantics yet. Until such time, the only things that we may assume that we can do must directly follow from 1.4.5 in connection with our axioms and rules of inference.

\[
\begin{align*}
(1) & \quad A \vdash B & \quad \text{(hypothesis)} \\
(2) & \quad C[p := A] \vdash C[p := B] & \quad \{(1) \text{ and Leibniz)}
\end{align*}
\]

\(Hmhm\). Is there a pattern here? Indeed there is! Any rule like \((R)\) of p. 40 leads to the statement of provability

\[
P_1, P_2, \ldots, P_n \vdash Q \quad \text{\[(R')\]}
\]

as the proof

\[
P_1, P_2, \ldots, P_n, Q
\]
establishes. This once we have written a Hilbert proof horizontally and without annotation, in order to expedite the obvious.

That this is a \{P_1, P_2, \ldots, P_n\}-proof of \(Q\) is clear: Referring to Definition 1.4.5 we see that writing \(P_1, P_2, \ldots, P_n\) (indeed doing so in any order if we wish) is legitimate by \(\text{Pr}1\). Then following this by writing \(Q\) is also legitimate by an application of rule \(R\) on the previously written formulæ.

This is why in some of the literature (e.g., [43]) rules of inference are written as in \(R'\) above rather than in “fraction form”. Tradition has it that derived rules of inference are always written in the style \(R'\). After all, such a rule is a (meta)provable principle and says that from certain assumptions we can prove a certain conclusion.

(c) We (meta) prove a derived rule of inference, called transitivity. It is

\[ A \equiv B, B \equiv C \vdash A \equiv C \]  
\((\text{Transitivity})\)

Here it goes:

1. \( A \equiv B \) (hypothesis)
2. \( B \equiv C \) (hypothesis)
3. \((A \equiv B) \equiv (A \equiv C)\) (2) and Leibniz, denom. “\(A \equiv p\)” where \(p\) is fresh
4. \( A \equiv C \) (1) and (3) and equanimity

What’s this about “fresh”? This means that \(p\) does not occur in any of \(A, B, C\). Actually, all I need here is that it just does not occur in \(A\), but it takes less space to say it is fresh in the annotation, so I can fit it in one line!

I want \(p\) not to occur in \(A\) so that when I do “(\(A \equiv p\))[p := B] \equiv (A \equiv p))[p := C]”, I am guaranteed that I get line (3). If \(p\) does occur in \(A\) then the substitutions will change \(A\) and I will not get line (3)! See also 1.3.17.

Pause. But what if I cannot find a fresh \(p\)? Actually, I always can, since I have an infinite supply of variables, while only finitely many appear in \(A, B, C\).

(d) We next prove the theorem (schema) \(A \equiv A\). Note that I mentioned no assumptions: This is an absolute result.

Another way to say this is “mutaprove \(A \equiv A\)”. or if you hate to say “meta”, say instead, “establish that \(\vdash A \equiv A\)”. or “show that \(\vdash A \equiv A\).

1. \( A \lor A \equiv A \) (axiom)
2. \( A \equiv A \) (1) and Leib: \(A[p := A \lor A] \equiv A[p := A] \) where \(p\) is fresh

A few remarks: (i) We may use any logical axiom of the form “\(\ldots \equiv \ldots\)” in place of \(A \lor A \equiv A\) in the above proof. By the way, this is our first proof where we used a logical axiom.

(ii) For logical axioms our annotation will just be “axiom”. For special axioms, our annotation will always be “assumption” or “hypothesis”.
(iii) We do not need to name the axioms (idempotent, etc.) in our annotations, and certainly I do not want you to memorize their numbers in the list! (What if I scramble the list?!) But we must be truthful. Writing, say, "A ≡ B" and annotating "axiom" will not get us anywhere.

(iv) Rules must be named, and we must annotate how they are applied! In what follows we will make a habit of abbreviating rule names. For example, "Leibniz" will be Leib and "Equanimity" will be Eqn. Transitivity will be Trans.

In the next chapter we systematically prove several theorems (in almost all cases schemata) and metatheorems to enrich our toolbox and enhance our familiarity with the methodology. We also introduce the equational style of proof.

1.5 ADDITIONAL EXERCISES

1. Which of the following are Boolean formulae? Why? (Do not use any general principles; just try to give a good reason based on the definition of formula-calculation (1.1.3)).
   - p
   - (p)
   - T
   - ∨
   - p → q
   - (p → q)

2. Are the following string sequences formula-calculation? Why?
   - p, T, (p ∨ ⊥), ⊥
   - p, ⊥, (p ∨ ⊥), T

3. Give a formula-calculation for \( \neg((p \lor q) \rightarrow \bot) \)

4. Prove either by analyzing formula-calculation or by induction on formulac that the string (\( ) \) is not a formula.

5. Prove either by analyzing formula-calculation or by induction on formulæ that the string (\( \neg \bot \)) is not a formula.

6. True or false, and why? "If \( A \) is a formula, then so is (\( A \))."

7. Show by induction on formulæ, or by analyzing formula-calculation, that every Boolean formula must contain at least one Boolean variable or one Boolean constant.
8. Prove that the complexity of a Boolean formula—correctly written as required by 1.1.5—equals the number of its left brackets.

9. (a) Prove that the last symbol of a Boolean formula is never ∧.
(b) Prove that the string ∧ ∨ never occurs as part of a Boolean formula.

The proof of each part must be either by induction on the complexity of formulae, or by analyzing formula-calculations. In part (b), you may use the result of part (a).

10. Which of the following schemata are tautologies? Show all work and remember that to show that a schema is not a tautology we must identify an instance of it that is not a tautology.

I am not using all the brackets required by 1.1.5.

- \((A \rightarrow B) \rightarrow A\) \rightarrow A
- \(A \land B \rightarrow A \lor B\)
- \(A \lor B \rightarrow A \land B\)
- \(A \rightarrow B \equiv \neg B \rightarrow \neg A\)
- \(A \land (B \equiv C) \equiv A \land B \equiv A \land C\)
- \(A \lor (B \equiv C) \equiv A \lor B \equiv A \lor C\)

11. Using truth tables or truth table shortcuts, determine the validity of the following. Show all your work. Again, a schema is not a tautological implication iff some instance of it is not.

- \(p \models \text{taut } p \land q\)
- \(A, B \models \text{taut } A \land B\)
- \(A, A \rightarrow B \models \text{taut } B\)
- \(B, A \rightarrow B \models \text{taut } A\)
- \(p \land q \models \text{taut } p\)

12. Use a truth table, or a shortcut of one, to show that

\[\models \text{taut } (A \land B \land C \rightarrow D) \equiv (A \rightarrow (B \rightarrow (C \rightarrow D)))\]

13. Use a truth table, or a shortcut of one, to show that

\[C, A \rightarrow (B \equiv C) \models \text{taut } A \rightarrow B\]

14. Which of the following sets is satisfiable?

- \{A, A \rightarrow B, A \rightarrow \neg B\}
- \{A \lor B, \neg A \land C, \neg B, \neg C\}
15. Calculate the following (show/explain all work!).

\textbf{NB.} The first bullet below must be done using Definition 1.3.15, step by step. For the rest you are free to rely on the intuitive definition of substitution/replacement. Some of the replacements I ask you to do may be illegal. If so, explain precisely why they are illegal and don't do them!

\begin{itemize}
  \item \( \{ A \lor B, \neg A \lor C, B \lor C \} \)
\end{itemize}

\begin{enumerate}
  \item \( p \lor (q \rightarrow p)[p := r] \)
  \item \( (p \lor q)[p := t] \)
  \item \( (p \lor q)[p := \neg] \)
  \item \( p \lor q \land r[q := A'] \) (where \( A \) is some formula, we don't care which)
  \item \( p \lor (q \land r)[q := A] \) (where \( A \) is some formula, we don't care which)
\end{enumerate}

16. If \( A \models_{\text{mut}} B \) and also \( B \models_{\text{mut}} A \), then we say that \( A \) and \( B \) are \textit{tautologically equivalent}. Prove that every formula is tautologically equivalent to one that contains no constants (\( \bot, \top \)) and moreover, the only connectives in it are \( \neg \) and \( \lor \).

17. Prove that every formula is tautologically equivalent to one that contains no constants (\( \bot, \top \)) and moreover, the only connectives in it are \( \neg \) and \( \land \).

18. Prove that every formula is tautologically equivalent to one that does not contain the constant \( \top \) and moreover, the only connective in it is \( \neg \).

19. Let us introduce a new Boolean connective \( \downarrow \) by "\( A \downarrow B \) means \( \neg( A \lor B ) \)". Prove that every formula is tautologically equivalent to one that contains no constants (\( \bot, \top \)) and moreover, the only connective in it is \( \downarrow \).

20. Let us introduce a new Boolean connective \( \uparrow \) by "\( A \uparrow B \) means \( \neg( A \land B ) \)". Prove that every formula is tautologically equivalent to one that contains no constants (\( \bot, \top \)) and moreover, the only connective in it is \( \uparrow \).