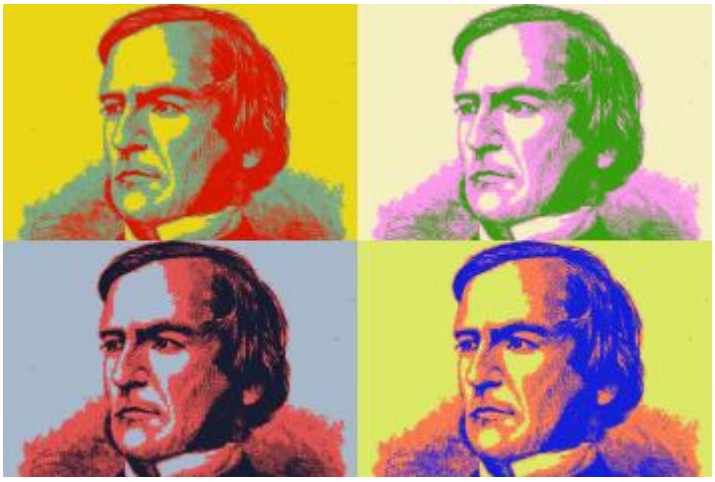
AN INTRODUCTION TO LOGIC

Mark V. Lawson

July 2016



Contents

Preface

Background. These notes were written to accompany my Heriot-Watt University course *F17LP Logic and proof* which I designed and wrote in 2011. The course was in fact instigated by my colleagues in Computer Science and was therefore designed mainly for first year computer science students, but the course is also offered as an option to second year mathematics students.

In writing this course, I was particularly influenced, like many others, by Smullyan's book [13]. Chapters 1 and 3 of these notes cover roughly the same material as the first 65 pages of [13]. I have also incorporated ideas to be found in [12] and [14].

This is very much a *first* introduction to logic and I have been mindful throughout of the nature of the students taking the course, but anyone completing it should have good foundations for further study if desired.

Aims. This is an introduction to first order logic suitable for first and second year mathematicians and computer scientists. There are three components to this course: propositional logic, Boolean algebras and predicate or first-order logic. Logic is the basis of proofs in mathematics — how do we know what we say is true? — and also of computer science — how do I know this program will do what I think it will do? Propositional logic and predicate logic are the two ingredients of first-order logic and are what make proofs work. Propositional logic deals with proofs that can be analysed in terms of the words *and, or, not, implies* whereas predicate logic extends this to encompass the use of the words *there exists* and *for all*. Boolean algebra, on the other hand, is an algebraic system that arises naturally from propositional logic and is the basic mathematical tool used in circuit design in computers.

How much maths? Surprisingly little school-type mathematics is needed to learn and understand logic: this course doesn't involve any calculus, for

example. In fact, most of the maths you studied at school was invented before 1800 and so is (mainly) irrelevant to the needs of computer science (a slight exaggeration, but not by much). The real mathematical prerequisite is an ability to manipulate symbols: in other words, basic algebra. Anyone who can write programs should have this ability already.

Books. There are two kinds of books: those you read and those you work from. For background reading, I recommend [2, 5, 6]. The remaining books are work books. Of those, you might want to start with Zegarelli [17]; don't be put off by the *Dummies* tag, since it isn't a bad introduction to logic. For some of the more obviously mathematical content, Lipschutz and Lipson [8] is useful. The chapters you want, in the second edition are: 1, 2, 3, 4, and 15. However, let me emphasize that we won't be covering everything you will find there. Another maths type book is Hammack [4]. This might be more suitable for the more mathematically inclined student. The book by Smullyan [13] I used when I was writing this course. The book by Teller [15] is a step-up from the Dummies book. There are also two websites worth checking out: [16] is a truth table generator and [7] is a truth tree solver.

Corrections. These are first generation typed notes so I have no doubt that errors have crept in. If you spot them, please email me.

Syllabus. Below is the formal syllabus for the course. At the end of these notes, you will find a specimen exam paper. Exercises may be found throughout the book at the ends of sections.

Introduction

An overview of the sorts of questions we shall be dealing with in this course and, in particular, why mathematics is such an important ingredient in computer science.

1. Propositional logic (PL)

1.1 Informal propositional logic: I shall introduce (PL) in the first instance as a very simple kind of language in which to express various kinds of simple decision-making scenarios. Later we shall see that it can also be viewed as a programming language.

1.2 Syntax of propositional logic: Well-formed formulae (wff), atoms,

compound statements, trees (examples of data structures), parse trees, principal connective, order of precedence rules and brackets.
1.3 Semantics of propositional logic: Definition of the connectives by means of truth-tables, truth assignments, truth-tables in general, contradictions, contingencies and tautologies, satisfiability.
1.4 Logical equivalence: Logical equivalence ($\equiv$), some important examples of logical equivalences, how to say 'exactly one'.
1.5 Two examples: Sudoku viewed through PL glasses.
1.6 Adequate sets of connectives: Truth functions, the connectives **0** and **1**, nand and nor, adequate sets of connectives. **Important theorem**: every truth-function is the truth-table of some wff.
1.7 Normal forms: Negation normal form (NNF), disjunctive normal form (DNF), conjunctive normal form (CNF).
1.8 $P = NP$?: The satisfiability problem: example using sudoku; connections with the question of whether $P = NP$? and its significance. This section is mainly for background knowledge and interest and to explain how (PL) can also be viewed as a programming language.
1.9 Valid arguments: What do we mean by an argument? The use of (PL) in formalizing certain kinds of simple arguments — one of the goals of logic: to mechanize reasoning. The semantic turnstile $\models$ and its properties; classical arguments: modus ponens, modus tollens, disjunctive syllogism, hypothetical syllogism (these terms do not have to be memorized).
1.10 Truth-trees: Derivation of tree rules, the truth-tree algorithm, using truth-trees to solve problems: determining whether a wff is a tautology, determining whether an argument is valid, determining whether a finite set of wff is satisfiable, writing a wff in DNF; the symbol $\vdash$. The following is non-examinable: the soundness and completeness theorem for propositional logic.

2. Boolean algebras

2.1 Definition of Boolean algebras: From propositional logic to Boolean algebras; the Lindenbaum algebra; the axioms defining a Boolean algebra.
2.2 Set theory: Introduction to sets, the power set of a set, intersections, unions, and relative complements, Venn diagrams; the two-element Boolean algebra, simplifying Boolean expressions.

2.3 Binary arithmetic: Writing numbers in base 2; adding numbers in base 2.
2.4 Circuit design: Shannon's work on switching circuits and Boolean algebra, logic gates, from switches to transistors, designing combinatorial circuits; binary arithmetic; how to build an adding machine.
2.5 Transistors: The main goal of this section is to explain why transistors are such important components in a computer.

3. First-order logic (FOL)

3.1 Splitting the atom: names, predicates and relations: Names or constants and variables; 1-place predicates, n-place or n-ary predicates, binary and ternary predicates, the arity of a predicate; atomic formulae; relations of different arities; directed graphs and binary relations.
3.2 Structures: Domains and relations; examples.
3.3 Quantification $\forall$ *and* $\exists$: $\forall$ regarded as an infinite conjunction and $\exists$ regarded as an infinite disjunction; a famous argument on Socrates' mortality analysed.
3.4 Syntax: A first-order language, formulae, parse trees; subtrees of trees, free and bound variables, the scope of a quantifier, closed formula/sentence.
3.5 Semantics: Interpretations, models, logically valid sentences; valid arguments; logical equivalence; satisfiable; contradictions.
3.6 De Morgan's laws for quantifiers: $\neg(\forall x)A \equiv (\exists)\neg A$ and $\neg(\exists x)A \equiv (\forall x)\neg A$.
3.7 Truth-trees for first-order logic: The two additional rules needed to deal with $(\forall x)$ and $(\exists x)$; examples. The following is non-examinable: the deeper theory of truthtrees; systematic truth-trees; Gödel's completeness theorem (no proof).
3.8 The Entscheidungsproblem: Turing, FOL and the computer.

Introduction

1. The world is all that is the case. — Tractatus Logico-Philosophicus, Ludwig Wittgenstein.

Discrete mathematics

Most of the mathematics you learnt at school is pretty irrelevant to computer scientists. I am exaggerating slightly, but not by much. The reason is that most of the mathematics you studied at school is old, very old. In fact, most was invented before 1800. The mathematics that is important in computer science is known as *discrete mathematics*. You will learn this mathematics from scratch at university. The only prerequisites are an ability to manipulate symbols — and if you have ever written a successful computer program you have already demonstrated that ability. Mathematics is *the* tool needed in all the sciences such as physics, engineering and computer science. My aim is not to turn you into mathematicians, but to help you become better computer professionals who understand how to use mathematics in your work. The particular discrete mathematics we shall study in this course is called *first-order logic*. In this section, I want to describe in outline some of the ways that logic is used in CS in the hope that it will whet your appetite to learn more.

The computer

If the Victorian age can be characterized by the steam engine, then ours can be characterized by the computer. Computers are usually viewed as simply products of technology, but computers do not work by engineering alone. They are driven by ideas, and some of these ideas are the subject of this course. One of the reasons the computer is so important is that it is a general purpose machine. In the past, different machines had to be

constructed for different purposes, but one and the same computer — the *hardware* — can be made to do different jobs by changing the instructions or programs it is given — the *software*. I shall now examine in a little more detail these two different aspects of the computer, and I hope to convince you that there are mathematical ideas that lie behind them both. It is these mathematical ideas that will form the basis of this course.

Hardware: the transistor

The hardware of a computer is the part that we tend to notice first. It is also the part that most obviously reflects advances in technology. The single most important component of the hardware of a computer is called a *transistor*. Computers contain millions or even billions of transistors. In 2015, for example, Wikipedia claimed that there are commercially available chips containing $5 \cdot 5$ billion transistors. There is even an empirical result, known as *Moore's law*, which says the number of transistors in integrated circuits doubles roughly every 2 years. Transistors were invented in the 1940s and are constructed from semiconducting materials, such as silicon. The way they work is quite complex and depends on the quantum-mechanical properties of semiconductors, but what they *do* is very simple. The basic function of a transistor is to amplify a signal. A transistor is constructed in such a way that a weak input signal is used to control a large output signal and so achieve amplification of the weak input[1]. An extreme case of this behaviour is where the input is used to turn the large input on or off. That is, where the transistor is used as an electronic switch. It is this function which is the most important use of transistors in computers. To understand this a little better, you have to know that computers operate using *binary logic*. This means that all the information that a computer can handle — text, pictures, sound, whatever — is represented by means of sequences that consist of two values. In a computer, these two values might be high or low voltages but mathematically we think of them as 1 or 0, or T and F, standing for *true* and *false*, respectively[2]. From a mathematical point of view, a transistor can be regarded as a device with one input, one control and one output.

[1] If you have ever used a shower where a small adjustment of the temperature control changes the water from freezing cold to boiling hot, you will have experienced what amplification of a signal can be like.

[2] It might help in what follows to think of 1 to mean that a current is flowing and 0 to mean that a current is not flowing.

- If 0 is input to the control then the switch is closed and the output is equal to the input.
- If 1 is input to the control then the switch is open and the output is 0 irrespective of the input.

The table below shows how a transistor functions.

Input	Control	Output
0	0	0
1	0	1
0	1	0
1	1	0

A transistor on its own doesn't appear to accomplish very much, but more complicated behaviour can be achieved by connecting transistors together into what are called *circuits*. Remarkably, the mathematics needed to understand how to design circuits using transistors is very old and predates the invention of the transistor by over 100 years. In 1854, George Boole (1815–1864), an English mathematician, wrote a book, *An investigation of the laws of thought*, where he tried to show that some of the ways in which we reason could be studied mathematically. This led to what are now called Boolean algebras and was also important in the development of propositional logic and set theory. We shall study all of these topics in this course. The discovery that Boole's work could be used to help in the design of circuits was due to to Claude Shannon (1916–2001) in 1937. Boolean algebras are now the basic mathematical tool needed to design computer circuits. *In this course, we shall define Boolean algebras, study their properties, and use them to design simple circuits. We shall also show that all circuits can be constructed from transistors.*

P = NP? Or how to make a $1,000,000

Imagine a very complex circuit constructed out of the transistors described above. There are n input wires and one output wire. I ask a very simple question about this circuit: is there some combination of inputs (that is, some combination of 0s and 1s on the input wires) so that the circuit outputs 1? This is a version of what is known as the *satisfiability problem (SAT)*. For a single transistor the answer to this question is very easy: the

transistor output is 1 precisely when input = 1 and control = 0. You might think that the answer to my general question is likewise easy. There are 2^n possible combinations of 0s and 1s on the n input wires. Thus we simply try all possible combinations of inputs. Either the output is always 0, in which case we have answered our original question in the negative, or there is some input combination that does yield 1 as an output, in which case we shall eventually find it and the answer to the question is in the affirmative. But there is a snag. The function $n \mapsto 2^n$ is an example of an *exponential function*. Simply put, it gets very big very quickly. Let me illustrate what I mean. Suppose that $n = 90$ so that there are 90 input wires. Suppose also that you can check each combination of 0s and 1s on the input wires in 10^{-9} seconds. Then the total time needed to check all possible combinations is $2^{90} \cdot 10^{-9}$ seconds. A simple calculation using logs (since $10 \approx 2^{3.322}$) shows that $2^{90} \approx 10^{27}$. Thus the total time is 10^{18} seconds. This is just over twice the age of the universe! We deduce that our method for solving SAT is not very practical (to say the least) which raises the question of whether there is a more practical way of solving this problem. So important is this problem that a one million dollar prize (*The Millennium Prize Problems*) has been offered for anyone who either finds a fast algorithm to solve it or proves that no such algorithm exists.

Software: algorithms

An *algorithm* is a method designed to solve a specific problem in a systematic way which does not require any intelligence in its application. This is not a precise definition instead it is intended to convey the key idea. Algorithms embody the mathematical idea of how to solve a problem. Much, but by no means all, of mathematics is about designing algorithms to solve specific problems. Mathematicians have been doing this for about 4,000 years. For example, at school we learn algorithms to add, subtract, multiply and divide numbers. A *program* is an implementation of an algorithm in a computer language (such as Python, Java etc). Programs need to deal with practical issues that mathematicians ignore. For example, the nature of the input, output or error messages. Think of algorithms and programs as two sides of the same coin. They lie at the interface between mathematics and CS. Thus every single program you ever write or use is the implementation of an algorithm. It follows that mathematicians have been writing programs for nearly 4,000 years. It's just that they only got around to inventing the

computer less than 100 years ago. There is a huge amount of experience and concrete methods that computer scientists can draw upon in mathematics to help them write their programs.

From transistors to programs

To a first approximation, then, a computer is a machine consisting of millions of electronic switches. But this raises the question of how such a mechanism could do all the things that a computer can do. The answer is that by setting those electronic switches to different positions inputs can be converted to outputs in different ways. At a fundamental level, this is what the software of the computer does. Think of the computer hardware as being a little like a piano and the software as being the music that can be played on the piano. The software that organizes all those electronic switches is called, of course, a program. Now at this point, you may be wondering how on earth anyone could write a program to make a computer work when they have to worry about what millions of little electronic switches are going to do. The answer is another mathematical idea: we arrange a complex structure into a hierarchy of substructures which perform easy to understand functions. An analogy might help. People are constructed from cells as a computer is constructed from transistors, but it would be a poor doctor who viewed their patient as simply a collection of cells. Instead, we group cells together into organs, such as the heart, brain, liver and so on, and we understand those organs by the functions that they perform. In the same way, the transistors of a computer are organized into circuits that perform certain functions and so on until we reach the top-most layer where we just click on a mouse or poke the screen to achieve certain goals. We shall be interested not in the top-most layer but a little further down the hierarchy where we view a computer as consisting of a memory connected to a CPU (central processing unit). It is at this level that all the wonderful effects at the top-most level are produced, and it is at this level that the programs work to produce all those wonderful effects. Thus programs are written to carry out tasks at this level in a way that is easy for humans to understand using some computer language designed for this purpose, and then are translated into a form that the computer can deal with ultimately leading to a description of which transistors are to be open at any given time. Thus to solve a problem using a computer we first have to find an algorithm that solves the problem (mathematics might help you here); we then write a program in

some convenient computer language; and then we run the computer using this program to accomplish our goals. This all sounds very straightforward but there is a problem. If you write a program that you claim solves a problem how can I — the user — be convinced that your program actually works as advertised? The mere fact that you are convinced you are right is simply not good enough — we all make mistakes and we are all limited in our intellectual abilities. *In fact, the burden of proof that you are right is on you, the one writing the program, and not on me, the user of your program.* This raises the question of how we can *prove* that an algorithm works. Mathematicians figured out how to do this, at least in principle, well over 2,300 years ago. *Logic is a language and a collection of methods that enables us to write down proofs.* This is what this course is about. It does not, however, tell you how to find proofs. To go back to my musical analogy, logic tells you what notes are available and what notes go well together but doesn't tell you how to compose — which requires talent, practice, creativity etc.

Some key questions about programs

1. How do we prove our programs work? Think of the programs that enable planes to be flown using fly-by-wire or are embedded in medical equipment. We need a guarantee that a program does what it is supposed to do.
2. How do we design good programming languages?
3. How efficient are our programs?
4. Are there intrinsic limitations to what can be computed (yes: see the work of Alan Turing)?
5. How can we emulate the way the brain works? We would like to build devices that simulate intelligence.

All the questions I have raised can be handled using logic. Logic is the subject that studies how we reason. It is convenient when learning logic to divide it into two parts.

Propositional logic (PL). This is the logic of elementary decision making. It is very simple and easy to use, there are some interesting and important applications, but it is not very powerful.

Predicate or first-order logic (FOL). This is an extension of PL that includes variables, quantifiers and the apparatus needed to describe much more complex reasoning. It is very powerful, being the basis of all logic studied in CS and suitable for describing all of mathematics, but it is therefore necessarily more complex.

PROLOG

Computer programs are written using artificial or formal languages called *programming languages*. There are two big classes of such languages: *imperative* and *declarative*. In an imperative language, you have to specify in detail how the solution to a problem is to be accomplished. In other words, you have to tell the computer *how to* solve the problem. In a declarative language, you tell the computer *what* you want to solve and leave the details of how to accomplish this to the software. Declarative languages sound like magic but some of them, such as PROLOG, are based on ideas drawn from first-order logic. This language has applications in AI (artificial intelligence).

Logic: a little history

Although the ideas of proof go back a long time, the modern theory of logic is really a product of the latter part of the nineteenth century and the early decades of the twentieth. It is particularly associated with the following names: Boole, Cantor, Church, Frege, Gentzen, Gödel, Hilbert, Russell, Turing and von Neumann. I recommend you look them up on Wikipedia to find out what contributions they made. The names of Turing and von Neumann are also associated with the early development of the computer. In fact, Alan Turing is often called the father of the computer and after WWII, Turing in Britain and von Neumann in the States were both involved in building the first computers. This connection is not accidental. In the early decades of the twentieth century, mathematicians and philosophers began to analyse in great detail how people reason and prove things. It was only a small step from that to wondering whether their insights could be embodied in machines. An account of some of their work can be found in the graphic novel [2]. Logic is a major tool in computer science often described as the 'calculus of computer science'. It is also important in mathematics and in philosophy although in this course, we shall not have anything to say about the philosophical side of logic.

Let me just digress a bit to say more about Turing. Alan Turing was one of the great mathematicians of the twentieth century, and perhaps unusually for a mathematician his life and work has affected the lives of people all over the world. Turing's reputation as the father of computer science rests first of all on a paper he wrote in 1937: A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, **42** (1937), 230–265. In this paper, Turing describes what we now call in his honour a *universal Turing machine*. This is a mathematical blueprint for building a computer. Such a machine runs what we would now call programs. But his description is mathematical and independent of technology. You can access copies of this paper via the links to be found on the Wikipedia article on the *Entscheidungsproblem* (see below). A problem can be solved algorithmically exactly when it can be solved by a program running on a Turing machine. Remarkably, Turing showed in his paper that there were problems that cannot be solved by computer — not because they weren't powerful enough but because they were intrinsically unable to solve them. Thus the limitations of computers were known before the first one had ever been built. The idea that there are limitations to computers might surprise you but is an important theme of theoretical computer science. Turing didn't set out to invent computers. He set out to solve a famous problem stated in 1928 by the great German mathematician David Hilbert. Because Hilbert was German this problem is always known by its German title the *Entscheidungsproblem* which simply means the *decision problem*. Roughly speaking, and using modern terminology, this problem asks whether it is possible to write a program that will answer all mathematical questions. Turing proved that this was impossible. The essence of mathematics is the notion of proof. What a proof is is described by logic. The Entscheidungsproblem is a question about logic. At the end of this course, you will understand what is meant by this problem and its significance in logic.

Turing's work in his paper was the beginning of a short but extraordinarily varied and influential career. During WWII, he worked in Hut 8 at Bletchley Park, the Government Code and Cypher School, and was central to the breaking of Dolphin, the Nazi Enigma code. He was also principal designer of the bombe, a codebreaking machine. After the war, he designed and developed ACE, the Automatic Computing Engine, an electronic digital computer. Later he moved to Manchester and oversaw the Computing Machine Laboratory. In 1947, he founded the field of Machine Intelligence now

known as Artificial Intelligence. In 1952, he carried out pioneering work in mathematical biology by studying pattern formation in animals. This work involved what we would now call computer simulation. You can read an exegesis of Turing's paper in [11].

Summary of this course

The main goal of this course is to introduce you to *logic*, a subject fundamental to both mathematics and computer science and, indeed, analytic philosophy. For convenience in learning, the logic we shall study is divided into two parts: *propositional logic* and *first-order logic*. We shall start with propositional logic because it is easy to understand and to use but not very powerful, whereas later we shall introduce first-order logic which is more powerful but harder to understand and to use.

Propositional logic is useful in helping us understand computer programs and *Boolean algebra*, which is closely related to propositional logic, is the basis of *digital circuit design*. I shall only touch on circuit design but we will at least look at how to construct a simple calculator. You can read more on circuits and Boolean algebras in [10]. An important question that arises in propositional logic is the *satisfiability problem*. Many problems that on the face of it have nothing to do with logic can be rephrased as satisfiability problems in propositional logic, and I shall describe some examples. This problem is also at the core of one of the great unsolved mathematical questions in theoretical computer science: namely, whether P is equal to NP. I shall talk a little about this problem in this course.

First-order logic is really the basis of the whole of mathematics. It is also important in the design of certain so-called logic programming languages such as *PROLOG*. Logic is also an important tool in *program verification*.

Exercises 1

The exercises below do not require any prior knowledge but introduce ideas that are important in this course

---

1. Here are two puzzles by Raymond Smullyan mathematician and magician. On an island there are two kinds of people: *knights* who always tell the truth and *knaves* who always lie. They are indistinguishable.

(a) You meet three such inhabitants A, B and C. You ask A whether he is a knight or knave. He replies so softly that you cannot make out what he said. You ask B what A said and they say 'he said he is a knave'. At which point C interjects and says 'that's a lie!'. Was C a knight or a knave?

(b) You encounter three inhabitants: A, B and C.
A says 'exactly one of us is a knave'.
B says 'exactly two of us are knaves'.
C says: 'all of us are knaves'.
What type is each?

2. This question is a variation of one that has appeared in the puzzle sections of many magazines. There are five houses, from left to right, each of which is painted a different colour, their inhabitants are called Sarah, Charles, Tina, Sam and Mary, but not necessarily in that order, who own different pets, drink different drinks and drive different cars.

(a) Sarah lives in the red house.

(b) Charles owns the dog.

(c) Coffee is drunk in the green house.

(d) Tina drinks tea.

(e) The green house is immediately to the right (that is: your right) of the white house.

(f) The Oldsmobile driver owns snails.

(g) The Bentley owner lives in the yellow house.

(h) Milk is drunk in the middle house.

(i) Sam lives in the first house.

(j) The person who drives the Chevy lives in the house next to the person with the fox.

(k) The Bentley owner lives in a house next to the house where the horse is kept.

(l) The Lotus owner drinks orange juice.

(m) Mary drives the Porsche.

(n) Sam lives next to the blue house.

There are two questions: who drinks water and who owns the aardvark?

3. *Bulls and Cows* is a code-breaking game for two players: the code-setter and the code-breaker. The code-setter writes down a 4-digit secret number all of whose digits must be different. The code-breaker tries to guess this number. For each guess they make, the code-setter scores their answer: for each digit in the right position score 1 bull (1B), for each correct digit in the wrong position score 1 cow (1C); no digit is scored twice. The goal is to guess the secret number in the smallest number of guesses. For example, if the secret number is 4271 and I guess 1234 then my score will be 1B,2C. Here's an easy problem. The following is a table of guesses and scores. What are the possibilities for the secret number?

1389	$0B, 0C$
1234	$0B, 2C$
1759	$1B, 1C$
1785	$2B, 0C$

4. Consider the following algorithm. The input is a positive whole number n; so $n = 1, 2, 3, \ldots$ If n is even, divide it by 2 to get $\frac{n}{2}$; if n is odd, multiply it by 3 and add 1 to get $3n + 1$. Now repeat this process and only stop if you reach 1. For example, if $n = 6$ we get successively $6, 3, 10, 5, 16, 8, 4, 2, 1$ and the algorithm stops at 1. What happens if $n = 11$? What about $n = 27$? Is it true that whatever whole number you input this procedure always yields 1?

5. *Hofstadter's MU-puzzle.* A *string* is just an ordered sequence of symbols. In this puzzle, you will construct strings using the letters M, I, U where each letter can be used any number of times, or not at all. You are given the string MI which is your only input. You can make new strings only by using the following rules any number of times in succession in any order:

   (I) If you have a string that ends in I then you can add a U on at the end.

   (II) If you have a string Mx where x is a string then you may form Mxx.

   (III) If III occurs in a string then you may make a new string with III replaced by U.

(IV) If UU occurs in a string then you may erase it.

I shall write $x \to y$ to mean that y is the string obtained from the string x by applying one of the above four rules. Here are some examples:

- By rule (I), $MI \to MIU$.
- By rule (II), $MIU \to MIUIU$.
- By rule (III), $UMIIIMU \to UMUMU$.
- By rule (IV), $MUUUII \to MUII$.

The question is: can you make MU?

6. *Sudoku puzzles* have become very popular in recent years. The newspaper that first launched them in the UK went to great pains to explain that they had nothing to do with maths despite involving numbers. Instead, they said, they were logic problems. This of course is nonsense: logic is part of mathematics. What they should have said is that they had nothing to do with *arithmetic*. The goal is to insert digits in the boxes to satisfy two conditions: first, each row and each column must contain all the digits from 1 to 9 exactly once, and second, each 3×3 box must contain the digits 1 to 9 exactly once.

	1		4	2				5
		2		7	1		3	9
							4	
2		7	1					6
				4				
6					7	4		3
	7							
1	2		7	3		5		
3				8	2		7	

Chapter 1

Propositional logic

> *2.012. In logic nothing is accidental: if a thing* can *occur in a state of affairs, the possibility of the state of affairs must be written into the thing itself.* — Tractatus Logico-Philosophicus, Ludwig Wittgenstein.

The main goal of this course is to introduce *first-order logic* which can be traced back to the work of George Boole (1815–1864) and Gottlob Frege (1848–1925). This is divided into two parts: the first, and simpler, part is called *propositional* or *sentential logic*; the second, and more complex part, is called *first-order* or *predicate logic*. We deal with the predicate logic in Chapter 3.

First-order logic is an example of an *artificial language* to be contrasted with *natural languages* like English, Welsh, Estonian or Basque. Natural languages are clearly important since we all live our lives through the medium of our native languagues. But natural languages are often ambiguous and imprecise when we try to use them in technical situations. In such cases, artificial languages are used. For example, programming languages are artificial languages suitable for describing algorithms to be implemented by computer whereas first-order logic is an artificial language used to describe logical reasoning.

The description of a language has two aspects: *syntax* and *semantics*. Syntax, or grammar, tells you what the allowable sequences of symbols are, whereas semantics tells you what they mean. Thus to describe first-order logic I will need to describe both its syntax and its semantics.

1.1 Informal propositional logic

We begin by analysing everyday language. Our goal is to construct a precise, unambiguous language that will enable us to determine truth and falsity precisely. Language consists of sentences but not all sentences will be grist to our mill. Here are some examples.

1. Homer Simpson is prime minister.

2. The earth orbits the sun.

3. To be or not to be?

4. Out damned spot!

Sentences (1) and (2) are different from sentences (3) and (4) in that we can say of sentences (1) and (2) whether they are true or false — in this case, (1) is false and (2) is true — whereas for sentences (3) and (4) it is meaningless to ask whether they are true or false, since (3) is a a question and (4) is an exclamation.

> A sentence that is capable of being either true or false (though we might not know which) is called a *statement*.

In mathematics and computer science, it is enough to study only statements (although if we did that in everyday life we would come across as rather robotic).

Statements come in all shapes and sizes from the banal 'Marmite is brown' to the informative 'the battle of Hastings was in 1066'. But in our work, the only thing that interests us in a statement is whether it is *true (T)* or *false (F)*. In other words, what its *truth value* is and nothing else. We can now give some idea as to what the subject of logic is about.

> Logic is about deducing whether a statement is true or false on the of information given by some collection of statements.

I shall say more about this later.

Some statements can be analysed into combinations of simpler statements using special kinds of words called connectives.

Example 1.1.1. Let p be the statement *It is not raining*. This is related to the statement q given by *It is raining*. We know that the truth value of q will be the opposite of the truth value of p. This is because p is the *negation* of q. This is precisely described by the following *truth table*.

It is raining	It is not raining
T	F
F	T

To avoid pecularities of English grammar, we replace the word 'not' by the slighly less natural phrase 'It is not the case that'. Thus *It is not the case that it is raining* means the same thing as *It is not raining* though if you used that phrase in everyday language you would sound like a lawyer. We go one step further and abbreviate the phrase 'It is not the case that' by **not**. Thus if we denote a statement by p then its negation is **not** p. The above table becomes

p	**not** p
T	F
F	T

This summarizes the behaviour of negation for any statement p. What happens if we negate twice? Then we simply apply the above table twice.

p	**not not** p
T	T
F	F

Thus *It is not the case that it is not raining* should mean the same as *It is raining*. I know it sounds weird and it would be an odd thing to say as anything other than a joke but we are building a language suitable for maths and CS rather than everyday usage. The word **not** is our first example of a *logical* or *propositional connective.* It is a *unary* connective because it is only applied to one input.

I shall now introduce some further connectives all of which take exactly two inputs and so are examples of *binary* connectives.

Example 1.1.2. Under what circumstances is the statement *It is raining and it is cold* true? Well, I had better be both wet and cold. However, in everyday English the word 'and' often means 'and then'. The statement *I got up and had a shower* does not mean the same as *I had a shower and got*

up. The latter sentence might be a joke: perhaps my roof leaked when I was asleep. Our goal is to eradicate the ambiguities of everyday language so we cannot allow these two meanings to coexist. Therefore in logic only the first meaning is the one we use. To make it clear that I am using the word 'and' in a special sense, I shall write it in bold **and**.

Given two statements p and q, we can describe the truth values of the *compound statement* p **and** q in terms of the truth values assumed by p and q by means of a *truth table*.

p	q	p **and** q
T	T	T
T	F	F
F	T	F
F	F	F

This table tells us that the statement *Homer Simpson is prime minister* **and** *the earth orbits the sun* is false. In everyday life, we might struggle to know how to interpret such a sentence — if someone turned to you on the bus and said it, I think it would be alarming rather than false. Let me underline that the *only* meaning we attribute to the word **and** is the one described by the above truth table. Thus contrary to everyday life the statements *I got up* **and** *I had a shower* and *I had a shower* **and** *I got up* mean the same thing.

Example 1.1.3. The word *or* in English is a bit more troublesome. Imagine the following set-up. You have built a voice-activated robot that can recognize shapes and colours. It is placed in front of a white square, a black square and a black circle. You tell the robot to choose a black shape or a square. It chooses the black square. Is that good or bad? The problem is that the word *or* can mean *inclusive or* in which case the choice is good or it can mean *exclusive or* in which case the choice is bad. Both meanings are useful so rather than choose one over the other we use two different words to cover the two different meanings. This is an example of *disambiguation*.

We use the word **or** to mean *inclusive or*.

p	q	p **or** q
T	T	T
T	F	T
F	T	T
F	F	F

Thus p **or** q is true when *at least one* of p and q is true. We use the word **xor** to mean *exclusive or*.

p	q	p **xor** q
T	T	F
T	F	T
F	T	T
F	F	F

Thus p **xor** q is true when *exactly one* of p and q is true. Although we haven't got far into our study of logic, this is already a valuable exercise. If you use the word *or* you should always decide what you really mean.

Our next propositional connective will be familiar to maths students but less so to CS students. This is *is equivalent to* or *if and only if* that we write as **iff**. Its meaning is simple.

p	q	p **iff** q
T	T	T
T	F	F
F	T	F
F	F	T

Observe that **not**(p **iff** q) means the same thing as p **xor** q. This is our first result in logic. By the way, I have added brackets to the first statement to make it clear that we are negating the whole statement p **iff** q and not merely p.

Our final connective is the nightmare one *if* p *then* q which we shall write as p **implies** q. In everyday language, we think of the word *implies* as establishing a connection between p and q. But we want to be able to write p **implies** q whatever the sentences p and q are and say what the truth value of p **implies** q is in terms of the truth values of p and q. To understand the truth table for this connective remember that in logic we want to deduce truths from truths. We therefore never want to derive something false from something true. It turns out that this minimum condition is actually sufficient for what we want to do. (See Example 1.4.14 for an attempt to make

this definition plausible.)

p	q	p **implies** q
T	T	T
T	F	F
F	T	T
F	F	T

This does have some bizzare consequences. The statement *Homer simpson is prime minister* **implies** *the sun orbits the earth* is in fact true. This sort of thing can be offputting when first encountered and can seem to undermine what we are trying to achieve. But remember, we are using everyday words in very special ways.

> As long as we translate between English and logic choosing the correct words to reflect the meaning we intend to convey then everything will be fine. In fact, this example provides strong motivation for using symbols rather than the bold forms of the English words. That way we will not be misled.

Propositional or Boolean connectives

This course	**Zegarelli**	**English**	**Technical**
$\neg p$	$\sim p$	not p	negation
$p \wedge q$	$p \,\&\, q$	p and q	conjunction
$p \vee q$	$p \vee q$	p or q or both	disjunction
$p \to q$	$p \to q$	if p then q	conditional
$p \leftrightarrow q$	$p \leftrightarrow q$	p if and only if q	biconditional
$p \oplus q$	NA	p xor q	exclusive disjunction

Our symbol for **and** is essentially a capital 'A' with the cross-bar missing, and our symbol for **or** is the first letter of the Latin word *vel* which meant 'or'.

> A statement that cannot be analysed further using the propositional connectives is called an *atomic statement* or simply an *atom*. Otherwise a statement is said to be *compound*. The truth value of a compound statement can be determined once the truth values of the atoms are known by applying the truth tables of the propositional connectives defined above.

Example 1.1.4. Determine the truth values of the following statements.

1. (There is a Rhino under the table)$\vee\neg$(there is a Rhino under the table). [Always true]
2. $(1 + 1 = 3) \to (2 + 2 = 5)$. [True]
3. (Mickey Mouse is the President of the USA) $\leftrightarrow$ (pigs can fly). [Amazingly, true]

What we have done so far is informal. I have just highlighted some features of everyday language. What we shall do next is formalize. I shall describe to you an artificial language called PL motivated by what we have found in this section. I shall first describe its syntax and then its semantics. Of course, I haven't shown you yet what we can actually do with this artificial language. That I shall do later.

I should also add that the propositional connectives I have introduced are not the only ones, but they are the most useful. I shall show you later that in fact we can be much more economical in our choice of connectives without sacrificing expressive power.

1.2 Syntax of propositional logic

We are given a collection of symbols called *atomic statements* or *atoms*. I'll usually denote these with lower case letters $p, q, r, \ldots$ or their decorated variants $p_1, p_2, p_3, \ldots$. A *well-formed formula* or *wff* is constructed in the following way:

(WFF1). All atoms are wff.

(WFF2). If A and B are wff then so too are $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \oplus B)$, $(A \to B)$ and $(A \leftrightarrow B)$.

(WFF3). All wff are constructed by repeated application of the rules (WFF1) and (WFF2) a finite number of times.

A wff which is not an atom is said to be a *compound statement*.

Example 1.2.1. We show that

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

is a wff.

1. p, q and r are wff by (WFF1).
2. $(p \vee q)$ is a wff by (1) and (WFF2).
3. $((p \vee q) \wedge r)$ is a wff by (1), (2) and (WFF2).
4. $(\neg((p \vee q) \wedge r))$ is a wff by (3) and (WFF2), as required

Notational convention. To make reading wff easier, I shall omit the outer brackets and also the brackets associated with $\neg$.

Examples 1.2.2.

1. $\neg p \vee q$ means $((\neg p) \vee q)$.
2. $\neg p \to (q \vee r)$ means $((\neg p) \to (q \vee r))$ and is different from $\neg((p \to q) \vee r)$.

I tend to bracket fairly heavily but many books on logic use fewer brackets and arrange the connectives in a hierarchy:

$$\neg, \wedge, \vee, \to, \leftrightarrow$$

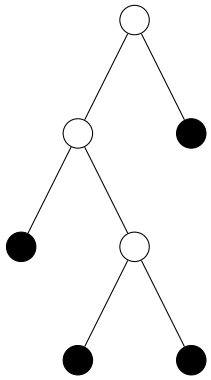
from 'stickiest' to 'least sticky'. It pays to check what conventions an author is using.

The collection of wff forms an example of a *formal language*. This consists of an underlying *alphabet* which in this case is

$$p, q, r, \ldots, p_1, p_2, p_3, \ldots \neg, \wedge, \vee, \oplus, \to, \leftrightarrow, (,)$$

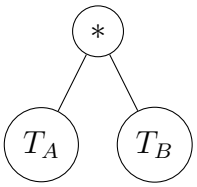
We are interested in strings over this alphabet (meaning ordered sequences of symbols from the alphabet) and finally there is a *context-free grammar*, also known as *Backus-Knaur form (BNF)*, which tells us which strings are wff: this is essentially given by our definition of wff above.

There is a graphical way of representing wff that involves trees. A *tree* is a data-structure consisting of circles called *nodes* or *vertices* joined by lines called *edges* such that there are no closed paths of distinct lines. In addition, the vertices are organized hierarchically. One vertex is singled out and called the *root* and is placed at the top. The vertices are arranged in levels so that vertices at the same level cannot be joined by an edge. The vertices at the bottom are called *leaves*. The picture below is an example of a tree with the leaves being indicated by the filled circles. The root is the vertex at the top.

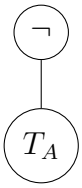


A *parse tree of a wff* is constructed as follows. The parse tree of an atom p is the tree

Now let A and B be wff. Suppose that A has parse tree T_A and B has parse tree T_B. Let $*$ denote any of the binary propositional connectives. Then $A * B$ has the parse tree



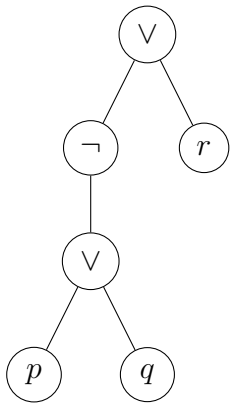
This is accomplished by joining the roots of T_A and T_B to a new root labelled by $*$. The parse tree for $\neg A$ is



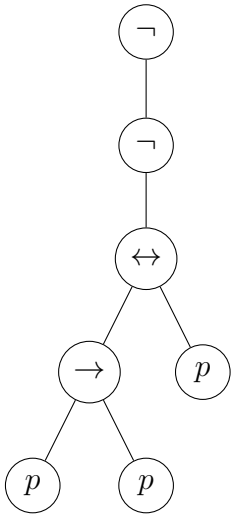
This is accomplished by joining the root of T_A to a new root labelled $\neg$. Parse trees are a way of representing wff without using brackets though we pay the price of having to work in two dimensions rather than one.

> We shall see in Chapter 2 that parse trees are in fact useful in circuit design.

Example 1.2.3. The parse tree for $\neg(p \vee q) \vee r$ is



Example 1.2.4. The parse tree for $\neg\neg((p \to p) \leftrightarrow p)$ is



1.3 Semantics of propositional logic

An atomic statement is assumed to have one of two *truth values*: *true (T)* or *false (F)*. We now consider the truth values of those compound statements that contain exactly one of the Boolean connectives. The following *truth tables* **define** the meaning of the Boolean connectives.

p	$\neg p$
T	F
F	T

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Then there is the one that everyone gets wrong

p	q	$p \rightarrow q$
T	T	T
T	**F**	**F**
F	T	T
F	F	T

The meanings of the logical connectives above are *suggested by* their meanings in everyday language, but are not the same as them. Think of our definitions as technical definitions for technical purposes only.

> *It is vitally important in what follows that you learn the above truth tables by heart.*

Truth tables can also be used to work out the truth values of compound statements. Let A be a compound statement consisting of atoms $p_1, \ldots, p_n$. A specific *truth assignment* to $p_1, \ldots, p_n$ leads to a truth value being assigned to A itself by using the definitions above.

Example 1.3.1. Let $A = (p \vee q) \rightarrow (r \leftrightarrow \neg s)$. A truth assignment is given by the following table

p	q	r	s
T	F	F	T

If we insert these values into our wff we get

$$(T \vee F) \rightarrow (F \leftrightarrow \neg T).$$

We use our truth tables above to evaluate this expression in stages

$$T \rightarrow (F \leftrightarrow F), \quad T \rightarrow T, \quad T.$$

Lemma 1.3.2. *If the compound proposition A consists of n atoms then there are 2^n possible truth assignments.*

We may draw up a table, also called a *truth table*, whose rows consist of all possible truth assignments along with the corresponding truth value of A.

We shall use the following pattern of assignments of truth values:

$\ldots$	T	T	T
$\ldots$	T	T	F
$\ldots$	T	F	T
$\ldots$	T	F	F
$\ldots$	F	T	T
$\ldots$	F	T	F
$\ldots$	F	F	T
$\ldots$	F	F	F
$\ldots$	$\ldots$	$\ldots$	$\ldots$

Examples 1.3.3. Here are some examples of truth tables

1. The truth table for $A = \neg(p \to (p \vee q))$.

p	q	$p \vee q$	$p \to (p \vee q)$	A
T	T	T	T	F
T	F	T	T	F
F	T	T	T	F
F	F	F	T	F

2. The truth table for $B = (p \wedge (p \to q)) \to q$.

p	q	$p \to q$	$p \wedge (p \to q)$	B
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

3. The truth table for $C = (p \vee q) \wedge \neg r$.

p	q	r	$p \vee q$	$\neg r$	C
T	T	T	T	F	F
T	T	F	T	T	T
T	F	T	T	F	F
T	F	F	T	T	T
F	T	T	T	F	F
F	T	F	T	T	T
F	F	T	F	F	F
F	F	F	F	T	F

4. Given the wff $(p \wedge \neg q) \wedge r$ we could draw up a truth table but in this case we can easily figure out how it behaves. It is true if and only if p is true, $\neg q$ is true and r is true. Thus the following is a truth assignment that makes the wff true

p	q	r
T	F	T

and the wff is false for all other truth assignments. We shall generalize this example later.

Important definitions

- An atom or the negation of an atom is called a *literal*.
- We say that a wff A built up from the atomic propositions $p_1, \ldots, p_n$ is *satisfiable* if there is some assignment of truth values to the atoms in A which gives A the truth value true.
- If $A_1, \ldots, A_n$ are wff we say they are *(jointly) satisfiable* if there is a single truth assignment that makes all of $A_1, \ldots, A_n$ true. It is left as en exercise to show that $A_1, \ldots, A_n$ are jointly satisfiable if and only if $A_1 \wedge \ldots \wedge A_n$ is satisfiable.
- If a wff is always true we say that it is a *tautology*. If A is a tautology we shall write

$$\vDash A.$$

The symbol $\vDash$ is called the *semantic turnstile*.

- If a wff is always false we say it is a *contradiction*. If A is a contradiction we shall write

  $$A \vDash .$$

  Observe that contradictions are on the left or sinister side of the semantic turnstile.

- If a wff is sometimes true and sometimes false we refer to it as a *contingency*.

- A truth assignment that makes a wff true is said to *satisfy* the wff otherwise it is said to *falsify* the wff.

A very important problem in PL can now be stated.

The satisfiability problem (SAT)
Given a wff decide whether there is some truth assignment to the atoms that makes the wff take the value true.

I shall discuss this problem in more detail later and explain why it is so important.

The following examples illustrate an idea that we shall develop in the next section.

Example 1.3.4. Compare the true tables of $p \to q$ and $\neg p \vee q$.

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

p	q	$\neg p$	$\neg p \vee q$
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

They are clearly the same.

Example 1.3.5. Compare the true tables of $p \leftrightarrow q$ and $(p \to q) \wedge (q \to p)$.

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

p	q	$p \to q$	$q \to p$	$(p \to q) \wedge (q \to p)$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

They are clearly the same.

Example 1.3.6. Compare the truth tables of $p \oplus q$ and $(p \vee q) \wedge \neg(p \wedge q)$.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

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

They are clearly the same.

> *It is important to remember that all questions in PL can be settled, at least in principle, by using truth tables.*

1.4 Logical equivalence

It can happen that two different-looking statements A and B can have the same truth table. This means they have the same meaning. We saw examples of this in Examples 1.3.4, 1.3.5 and 1.3.6. In that case, we say that A is *logically equivalent to* B written $A \equiv B$. It is important to remember that $\equiv$ is not a logical connective. It is a *relation* between wff.

Examples 1.4.1.

1. $p \to q \equiv \neg p \vee q$.
2. $p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$.
3. $p \oplus q \equiv (p \vee q) \wedge \neg(p \wedge q)$.

Observe that A and B do not need to have the same atoms but the truth tables must be constructed using all the atoms that occur in either A or B. Here is an example.

Example 1.4.2. We prove that $p \equiv p \wedge (q \vee \neg q)$. We construct two truth tables with atoms p and q in both cases.

p	q	p
T	T	T
T	F	T
F	T	F
F	F	F

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

The two truth tables are the same and so the two wff are logically equivalent.

The following result is the first indication of the important role that tautologies play in propositional logic. You can also take this as the definition of logical equivalence.

Proposition 1.4.3. *Let A and B be statements. Then $A \equiv B$ if and only if $A \leftrightarrow B$ is a tautology if and only if $\vDash A \leftrightarrow B$.*

Proof. We use the fact that $X \leftrightarrow Y$ is true when X and Y have the same truth value.

Let the atoms that occur in either A or B be $p_1, \ldots, p_n$.

Let $A \equiv B$ and suppose that $A \leftrightarrow B$ were not a tautology. Then there is some assignment of truth values to the atoms $p_1, \ldots, p_n$ such that A and B have different truth values. But this would imply that there was a row of the truth table of A that was different from the corresponding row of B. This contradicts the fact that A and B have the same truth tables. It follows that $A \leftrightarrow B$ is a tautology.

Let $A \leftrightarrow B$ be a tautology and suppose that A and B have truth tables that differ. This implies that there is a row of the truth table of A that is different from the corresponding row of B. Then there is some assignment of truth values to the atoms $p_1, \ldots, p_n$ such that A and B have different truth values. But this would imply that $A \leftrightarrow B$ is not a tautology. □

Example 1.4.4. Prove that $\vDash p \leftrightarrow (p \wedge (q \vee \neg q))$. This implies that $p \equiv p \wedge (q \vee \neg q)$.

p	q	$p \wedge (q \vee \neg q)$	$p \leftrightarrow (p \wedge (q \vee \neg q))$
T	T	T	T
T	F	T	T
F	T	F	T
F	F	F	T

The following theorem lists some important logical equivalences that you will be asked to prove in Exercises 2.

Theorem 1.4.5.

1. *$\neg\neg p \equiv p$. Double negation.*
2. *$p \wedge p \equiv p$ and $p \vee p \equiv p$. Idempotence.*

3. $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ *and* $(p \vee q) \vee r \equiv p \vee (q \vee r)$. *Associativity.*

4. $p \wedge q \equiv q \wedge p$ *and* $p \vee q \equiv q \vee p$. *Commutativity.*

5. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ *and* $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$. *Distributivity.*

6. $\neg(p \wedge q) \equiv \neg p \vee \neg q$ *and* $\neg(p \vee q) \equiv \neg p \wedge \neg q$. *De Morgan's laws.*

7. $p \vee (p \wedge q) \equiv p$ *and* $p \wedge (p \vee q) \equiv p$. *Absorption.*

There are some interesting patterns in the above results that involve the interplay between $\wedge$ and $\vee$:

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

and

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

There are some useful consequences of the above theorem.

- The fact that $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ means that we can write simply $p \wedge q \wedge r$ without ambiguity because the two ways of bracketing this expression lead to the same truth table. It can be shown that as a result we can write expressions like $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ (and so on) without brackets because it can be proved that however we bracket such an expression leads to the same truth table. What we have said for $\wedge$ also applies to $\vee$.

- The fact that $p \wedge q \equiv q \wedge p$ implies that the order in which we carry out a sequence of conjunctions does not matter. What we have said for $\wedge$ also applies to $\vee$.

- The fact $p \wedge p \equiv p$ means that we can eliminate repeats in conjunctions of one and the same atom. What we have said for $\wedge$ also applies to $\vee$.

Example 1.4.6. By the results above

$$p \wedge q \wedge p \wedge q \wedge p \equiv p \wedge q.$$

It is important not to overgeneralize the above results as the following two examples show.

Example 1.4.7. Observe that $p \to q \not\equiv q \to p$ since the truth assignment

p	q
T	F

makes the LHS equal to F but the RHS equal to T.

Example 1.4.8. Observe that $(p \to q) \to r \not\equiv p \to (q \to r)$ since the truth assignment

p	q	r
F	F	F

makes the LHS equal to F but the RHS equal to T.

Our next example is an application of some of our results.

Example 1.4.9. We have defined a binary propositional connective $\oplus$ such that $p \oplus q$ is true when exactly one of p or q is true. Our goal now is to extend this to three atoms. Define $\text{xor}(p, q, r)$ to be true when exactly one of p, q or r is true, and false in all other cases. We can describe this connective in terms of the ones already defined. I claim that

$$\text{xor}(p, q, r) = (p \vee q \vee r) \wedge \neg(p \wedge q) \wedge \neg(p \wedge r) \wedge \neg(q \wedge r).$$

This can easily be verified by constructing the truth table of the RHS. Put

$$A = (p \vee q \vee r) \wedge \neg(p \wedge q) \wedge \neg(p \wedge r) \wedge \neg(q \wedge r).$$

p	q	r	$p \vee q \vee r$	$\neg(p \wedge q)$	$\neg(p \wedge r)$	$\neg(q \wedge r)$	A
T	T	T	T	F	F	F	F
T	T	F	T	F	T	T	F
T	F	T	T	T	F	T	F
T	F	F	T	T	T	T	**T**
F	T	T	T	T	T	F	F
F	**T**	F	T	T	T	T	**T**
F	F	**T**	T	T	T	T	**T**
F	F	F	F	T	T	T	F

The following properties of logical equivalence will be important when we come to show how Boolean algebras are related to PL in Chapter 2.

Proposition 1.4.10. *Let A, B and C be wff.*

1. $A \equiv A$.
2. *If* $A \equiv B$ *then* $B \equiv A$.
3. *If* $A \equiv B$ *and* $B \equiv C$ *then* $A \equiv C$.
4. *If* $A \equiv B$ *then* $\neg A \equiv \neg B$.
5. *If* $A \equiv B$ *and* $C \equiv D$ *then* $A \wedge C \equiv B \wedge D$.
6. *If* $A \equiv B$ *and* $C \equiv D$ *then* $A \vee C \equiv B \vee D$.

Proof. By way of an example, I shall prove (6). We are given that $A \equiv B$ and $C \equiv D$ and we have to prove that $A \vee C \equiv B \vee D$. That is we need to prove that from $\vDash A \leftrightarrow B$ and $\vDash C \leftrightarrow D$ we can deduce $\vDash (A \vee C) \leftrightarrow (B \vee D)$. Suppose that $(A \vee C) \leftrightarrow (B \vee D)$ is not a tautology. Then there is some truth assignment to the atoms that makes $A \vee C$ true and $B \vee D$ false or vice versa. I shall just deal with the first case here. Suppose that $A \vee C$ is true and $B \vee D$ is false. Then both B and D are false and at least one of A and C is true. If A is true then this contradicts $A \equiv B$, and if C is true then this contradicts $C \equiv D$. It follows that $A \vee C \equiv B \vee D$, as required. □

Logical equivalence can be used to *simplify* complicated compound statements as follows. Let A be a compound statement which contains occurrences of the wff X. Suppose that $X \equiv Y$ where Y is simpler than X. Let A' be the same as A except that some or all occurrences of X are replaced by Y. Then $A' \equiv A$ but A' is simpler than A.

Example 1.4.11. Let

$$A = p \wedge (q \vee \neg q) \wedge q \wedge (r \vee \neg r) \wedge r \wedge (p \vee \neg p).$$

But

$$p \wedge (q \vee \neg q) \equiv p \text{ and } q \wedge (r \vee \neg r) \equiv q \text{ and } r \wedge (p \vee \neg p) \equiv r$$

and so

$$A \equiv p \wedge (q \wedge r).$$

Examples 1.4.12. Here are some examples of using known logical equivalences to show that two wff are logically equivalent.

1. We show that $p \to q \equiv \neg q \to \neg p$.

$$\begin{aligned} \neg q \to \neg p &\equiv \neg\neg q \vee \neg p \text{ by Example 4.1(1)} \\ &\equiv q \vee \neg p \text{ by double negation} \\ &\equiv \neg p \vee q \text{ by commutativity} \\ &\equiv p \to q \text{ by Example 4.1(1).} \end{aligned}$$

2. We show that $(p \to q) \to q \equiv p \vee q$.

$$\begin{aligned} (p \to q) \to q &\equiv \neg(p \to q) \vee q \text{ by Example 4.1(1)} \\ &\equiv \neg(\neg p \vee q) \vee q \text{ by Example 4.1(1)} \\ &\equiv (\neg\neg p \wedge \neg q) \vee q \text{ by de Morgan} \\ &\equiv (p \wedge \neg q) \vee q \text{ by double negation} \\ &\equiv (p \vee q) \wedge (\neg q \vee q) \text{ by distributivity} \\ &\equiv p \vee q \text{ since } \vDash \neg q \vee q. \end{aligned}$$

3. We show that $p \to (q \to r) \equiv (p \wedge q) \to r$.

$$\begin{aligned} p \to (q \to r) &\equiv \neg p \vee (q \to r) \text{ by Example 4.1(1)} \\ &\equiv \neg p \vee (\neg q \vee r) \text{ by Example 4.1(1)} \\ &\equiv \neg(p \wedge q) \vee r \text{ by associativity and de Morgan} \\ &\equiv (p \wedge q) \to r \text{ by Example 4.1(1).} \end{aligned}$$

4. We show that $p \to (q \to r) \equiv q \to (p \to r)$.

$$\begin{aligned} p \to (q \to r) &\equiv \neg p \vee (q \to r) \text{ by Example 4.1(1)} \\ &\equiv \neg p \vee (\neg q \vee r) \text{ by Example 4.1(1)} \\ &\equiv (\neg p \vee \neg q) \vee r \text{ by associativity} \\ &\equiv (\neg q \vee \neg p) \vee r \text{ by commutativity} \\ &\equiv \neg q \vee (\neg p \vee r) \text{ by associativity} \\ &\equiv \neg q \vee (p \to r) \text{ by Example 4.1(1)} \\ &\equiv q \to (p \to r) \text{ by Example 4.1(1).} \end{aligned}$$

5. We show that $(p \to q) \wedge (p \to r) \equiv p \to (q \wedge r)$.

$$\begin{aligned}(p \to q) \wedge (p \to r) &\equiv (\neg p \vee q) \wedge (\neg p \vee r) \text{ by Example 4.1(1)} \\ &\equiv \neg p \vee (q \wedge r) \text{ by distributivity} \\ &\equiv p \to (q \wedge r) \text{ by Example 4.1(1).}\end{aligned}$$

The next example is a little different.

Example 1.4.13. We shall prove that $\vDash p \to (q \to p)$ by using logical equivalences.

$$\begin{aligned}p \to (q \to p) &\equiv \neg p \vee (\neg q \vee p) \text{ by Example 4.1(1)} \\ &\equiv (\neg p \vee p) \vee \neg q \text{ by associativity and commutativity} \\ &\equiv T \text{ since } \vDash \neg p \vee p.\end{aligned}$$

Finally, here is an attempt to explain the rationale behind the definition of $\to$.

Example 1.4.14. I shall try to show how the truth table of $\to$ is forced upon us if we make some reasonable assumptions.

p	q	$p \to q$
T	T	u
T	F	v
F	T	w
F	F	x

We pretend that we do not yet know the values of u, v, w, x. We now make the following assumptions.

1. The truth table for $\leftrightarrow$ is known.
2. Whatever the truth table of $\to$ is we should have that
$$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p).$$
3. The truth table for $\to$ is different from that of $\leftrightarrow$.
4. $T \to F$ must be false.

In the light of the above assumptions, we have the following.

p	q	$p \to q$	$q \to p$	$(p \to q) \wedge (q \to p)$
T	T	u	u	$u = T$
T	F	v	w	$v \wedge w = F$
F	T	w	v	$w \wedge v = F$
F	F	x	x	$x = T$

Thus $u = T$ and $x = T$. We cannot have $w = v = F$ because then $\to$ would have the same truth table as $\leftrightarrow$. It follows that v and w must have opposite truth values. But then $v = F$ and so $w = T$.

Exercises 2

These cover Sections 1.1, 1.2, 1.3 and 1.4.

---

1. Construct parse trees for the following wff.

   (a) $(\neg p \vee q) \leftrightarrow (q \to p)$.

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

   (c) $(p \to \neg p) \leftrightarrow \neg p$.

   (d) $\neg(p \to \neg p)$.

   (e) $(p \to (q \to r)) \leftrightarrow ((p \wedge q) \to r)$.

2. Determine which of the following wff are satisfiable. For those which are, find all the assignments of truth values to the atoms which make the wff true.

   (a) $(p \wedge \neg q) \to \neg r$.

   (b) $(p \vee q) \to ((p \wedge q) \vee q)$.

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

3. Determine which of the following wff are tautologies by using truth tables.

(a) $(\neg p \vee q) \leftrightarrow (q \rightarrow p)$.

(b) $p \rightarrow ((q \rightarrow r) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)))$.

(c) $(p \rightarrow \neg p) \leftrightarrow \neg p$.

(d) $\neg(p \rightarrow \neg p)$.

(e) $(p \rightarrow (q \rightarrow r)) \leftrightarrow ((p \wedge q) \rightarrow r)$.

4. Prove the following logical equivalences using truth tables.

(a) $\neg\neg p \equiv p$.

(b) $p \wedge p \equiv p$ and $p \vee p \equiv p$. Idempotence.

(c) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ and $(p \vee q) \vee r \equiv p \vee (q \vee r)$. Associativity.

(d) $p \wedge q \equiv q \wedge p$ and $p \vee q \equiv q \vee p$. Commutativity.

(e) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$. Distributivity.

(f) $\neg(p \wedge q) \equiv \neg p \vee \neg q$ and $\neg(p \vee q) \equiv \neg p \wedge \neg q$. De Morgan's laws.

5. Let F stand for any wff which is a contradiction and T stand for any wff which is a tautology. Prove the following.

(a) $p \vee \neg p \equiv T$.

(b) $p \wedge \neg p \equiv F$.

(c) $p \vee F \equiv p$.

(d) $p \vee T \equiv T$.

(e) $p \wedge F \equiv F$.

(f) $p \wedge T \equiv p$.

6. Prove the following by using known logical equivalences (rather than using truth tables).

(a) $(p \rightarrow q) \wedge (p \vee q) \equiv q$.

(b) $(p \wedge q) \rightarrow r \equiv (p \rightarrow r) \vee (q \rightarrow r)$.

(c) $p \rightarrow (q \vee r) \equiv (p \rightarrow q) \vee (p \rightarrow r)$.

7. We defined only 5 binary connectives, but there are in fact 16 possible ones. The tables below show all of them.

p	q	$\circ_1$	$\circ_2$	$\circ_3$	$\circ_4$	$\circ_5$	$\circ_6$	$\circ_7$	$\circ_8$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

p	q	$\circ_9$	$\circ_{10}$	$\circ_{11}$	$\circ_{12}$	$\circ_{13}$	$\circ_{14}$	$\circ_{15}$	$\circ_{16}$
T	T	F	F	F	F	F	F	F	F
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

   (a) Express each of the connectives from 1 to 8 in terms of $\neg$, $\rightarrow$, p, q and brackets only.

   (b) Express each of the connectives from 9 to 16 in terms of $\neg$, $\wedge$, p, q and brackets only.

1.5 Two examples: PL as a 'programming language'

PL can seem like a toy and some of our examples don't help this impression, but in fact it has serious applications independently of its being the foundation of the more general first-order logic that we shall study in Chapter 3. In this section, we shall analyze a couple of examples of simplified Sudoku-type problem in terms of PL. These illustrate the ideas needed to analyse full Sudoku in terms of PL. In fact, many important problems in mathematics and computer science can be regarded as instances of the satisfiability problem. We shall say more about this in Section 1.8.

Example 1

To understand new ideas always start with the simplest examples. So, here is a childishly simple Sudoku puzzle. Consider the following grid:

where the small squares are called *cells*. The puzzle consists in filling the cells with numbers according to the following two constraints.

(C1) Each cell contains exactly one of the numbers 1 or 2.

(C2) If two cells occur in the same row then the numbers they contain must be different.

There are obviously two solutions to this puzzle

1	2	and	2	1

I shall now show how this puzzle can be encoded by a wff of PL. Please note that I shall solve it in a way that generalizes so I do not claim that the solution in this case is the simplest. We first have to decide what the atoms are. To define them we shall label the cells as follows

c_{11}	c_{12}

We need four atoms that are defined as follows.

- p is the statement that cell c_{11} contains the number 1.
- q is the statement that cell c_{11} contains the number 2.
- r is the statement that cell c_{12} contains the number 1.
- s is the statement that cell c_{12} contains the number 2.

For example, if p is true then the grid looks like this

1	?

where the ? indicates that we don't care what is there. Consider now the following wff.

$$A = (p \oplus q) \wedge (r \oplus s) \wedge (p \oplus r) \wedge (q \oplus s).$$

I now describe what each of the parts of this wff are doing.

- $p \oplus q$ is true precisely when cell c_{11} contains a 1 or a 2 but not both.
- $r \oplus s$ is true precisely when cell c_{12} contains a 1 or a 2 but not both.

- $p \oplus r$ is true precisely when the number 1 occurs in exactly one of the cells c_{11} and c_{12}.
- $q \oplus s$ is true precisely when the number 2 occurs in exactly one of the cells c_{11} and c_{12}.

Here is the important consequence of all this:

> **It follows that A is satisfiable precisely when the puzzle can be solved. In addition, each satisfying truth assignment can be used to read off a solution to the original puzzle.**

Here is the truth table for A.

p	q	r	s	A
T	T	T	T	F
T	T	T	F	F
T	T	F	T	F
T	T	F	F	F
T	F	T	T	F
T	F	T	F	F
T	F	F	T	T
T	F	F	F	F
F	T	T	T	F
F	T	T	F	T
F	T	F	T	F
F	T	F	F	F
F	F	T	T	F
F	F	T	F	F
F	F	F	T	F
F	F	F	F	F

We observe first that the wff A is satisfiable and so the original problem can be solved. Second, here are the two satisfying truth assignments.

p	q	r	s
T	F	F	T
F	T	T	F

The first truth assignment tells us that $c_{11} = 1$ and $c_{12} = 2$, whereas the second truth assignment tells us that $c_{11} = 2$ and $c_{12} = 1$. These are, of course, the two solutions we saw earlier.

Example 2

We now describe a slightly more complex example and generalize what we did above. To do this, I introduce some notation. Define

$$\bigvee_{i=1}^{n} A_i = A_1 \vee \ldots \vee A_n$$

and

$$\bigwedge_{i=1}^{n} A_i = A_1 \wedge \ldots \wedge A_n.$$

Consider the following slightly larger grid:

3		
		2

where again the small squares are called *cells*. Some cells contain numbers at the beginning and these must not be changed. Our task is to fill the remaining cells with numbers according to the following constraints.

(C1) Each cell contains exactly one of the numbers 1 or 2 or 3.

(C2) If two cells occur in the same row then the numbers they contain must be different.

(C3) If two cells occur in the same column then the numbers they contain must be different.

It is very easy to solve this problem satisfying these constraints to obtain

3	2	1
1	3	2
2	1	3

I shall now show how this problem can be represented in PL and how its solution is a special case of the satisfiability problem. First of all, I shall label the cells in the grid as follows:

c_{11}	c_{12}	c_{13}
c_{21}	c_{22}	c_{23}
c_{31}	c_{32}	c_{33}

The label c_{ij} refers to the cell in row i and column j. PL requires atomic statements. To model this problem we shall need 27 atomic statements c_{ijk} where $1 \leq i \leq 3$ and $1 \leq j \leq 3$ and $1 \leq k \leq 3$. The atomic statement c_{ijk} is defined as follows

$$c_{ijk} = \text{the cell in row } i \text{ and column } j \text{ contains the number } k.$$

For example, the atomic statement c_{113} is true when the grid is as follows:

3	?	?
?	?	?
?	?	?

where the ?s mean that we don't know what is in that cell. In the above case, the atomic statements c_{111} and c_{112} are both false.

We shall now construct a wff A from the above 27 atoms such that A is satisfiable if and only if the above problem can be solved and such that a satisfying truth assignment can be used to read off a solution. I shall construct A in stages.

- Define $I = c_{113} \wedge c_{232}$. This wff is true precisely when the grid looks like this

3	?	?
?	?	2
?	?	?

- Each cell must contain exactly one of the numbers $1, 2, 3$. For each $1 \leq i \leq 3$ and $1 \leq j \leq 3$ the wff

$$\text{xor}(c_{ij1}, c_{ij2}, c_{ij3})$$

is true when the cell in row i and column j contains exactly one of the numbers $1, 2, 3$. Put B equal to the conjunction of all of these wff. Thus

$$B = \bigwedge_{i=1}^{i=3} \bigwedge_{j=1}^{j=3} \mathrm{xor}(c_{ij1}, c_{ij2}, c_{ij3}).$$

Then B is true precisely when each cell of the grid contains exactly one of the numbers $1, 2, 3$.

- In each row, each of the numbers $1, 2, 3$ must occur exactly once. For each $1 \leq i \leq 3$, define

$$R_i = \mathrm{xor}(c_{i11}, c_{i21}, c_{i31}) \wedge \mathrm{xor}(c_{i12}, c_{i22}, c_{i32}) \wedge \mathrm{xor}(c_{i13}, c_{i23}, c_{i33}).$$

Then R_i is true when each of the numbers $1, 2, 3$ occurs exactly once in the cells in row i. Define $R = \bigwedge_{i=1}^{i=3} R_i$.

- In each column, each of the numbers $1, 2, 3$ must occur exactly once. For each $1 \leq j \leq 3$, define

$$C_j = \mathrm{xor}(c_{1j1}, c_{2j1}, c_{3j1}) \wedge \mathrm{xor}(c_{1j2}, c_{2j2}, c_{3j2}) \wedge \mathrm{xor}(c_{1j3}, c_{2j3}, c_{3j3}).$$

Then C_j is true when each of the numbers $1, 2, 3$ occurs exactly once in the cells in column j. Define $C = \bigwedge_{j=1}^{i=3} C_i$.

- Put $A = I \wedge B \wedge R \wedge C$. Then by construction A is satisfiable precisely when the original problem is satisfiable and a satisfying truth assignment to the atoms can be used to read off a solution as follows. Precisely the following atoms are true:

$$c_{113}, c_{122}, c_{131}, c_{211}, c_{223}, c_{232}, c_{312}, c_{321}, c_{333}$$

and all the remainder are false.

It is now easy in principle to generalize our two examples above and show that a full-scale Sudoku puzzle can be solved in the same way. This consists of a grid with 9×9 cells and each cell can contain exactly one of the numbers $1, 2, \ldots, 9$ satisfying certain constraints. It follows that to describe this puzzle by means of a wff in PL we shall need $9 \times 9 \times 9 = 729$ atoms. This is left as an exercise.

Remark 1.5.1. Don't be perturbed by the number of atoms in the above examples nor by the amount of labour needed to write down the wff A. The point is that the problem can be faithfully represented by a wff in PL and that the solution of the problem is achieved via a satisfying assignment of the atoms. We shall place this example in a more general context we we discuss '$P = NP$?' later on. In many ways, PL is like an *assembly language* and it is perfectly adapted to studying a particular class of problems that are widespread and important.

1.6 Adequate sets of connectives

We defined our version of PL using the following six connectives

$$\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus.$$

This is not a unique choice: for example, many books on logic do not include $\oplus$. In this section, we shall explore what is actually neded to define PL.

It is easy to show that

$$p \oplus q \equiv \neg(p \leftrightarrow q).$$

We have already proved that

$$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$$

and also that

$$p \rightarrow q \equiv \neg p \vee q.$$

We have therefore proved the following.

Proposition 1.6.1. *Every wff in PL is logically equivalent to one that uses only the logical connectives $\neg$, $\vee$ and $\wedge$.*

Although it would be more efficient to use only the above three logical connectives, it would be less user-friendly. Significantly, those books that do not take $\oplus$ as a basic logical connective are not sacrificing any expressive power.

At this point, we introduce some terminology. We say that a set of logical connectives is *adequate* if every wff is logically equivalent to a wff that uses

only logical connectives from that set. In these terms, we proved above that the connectives $\neg$, $\vee$, $\wedge$ form an adequate set.

We can if we want be even more miserly in the number of logical connectives we use. The following two logical equivalences can be proved using double negation and de Morgan.

- $p \vee q \equiv \neg(\neg p \wedge \neg q)$.
- $p \wedge q \equiv \neg(\neg p \vee \neg q)$.

From these we can deduce the following.

Proposition 1.6.2.

1. *The connectives $\neg$ and $\wedge$ together form an adequate set.*
2. *The connectives $\neg$ and $\vee$ together form an adequate set.*

Example 1.6.3. We show that the following wff are equivalent to wff using only the connectives $\neg$ and $\wedge$.

1. $p \vee q \equiv \neg(\neg p \wedge \neg q)$.
2. $p \to q \equiv \neg p \vee q \equiv \neg(\neg\neg p \wedge \neg q) \equiv \neg(p \wedge \neg q)$.
3. $p \leftrightarrow q \equiv (p \to q) \wedge (q \to p) \equiv \neg(p \wedge \neg q) \wedge \neg(q \wedge \neg p)$.

At this point, you might wonder if we can go one better. Indeed, we can but we have to define some new binary connectives. Define

$$p \downarrow q = \neg(p \vee q)$$

called *nor*. Define

$$p \mid q = \neg(p \wedge q)$$

called *nand*.

Proposition 1.6.4. *The binary connectives $\downarrow$ and $\mid$ on their own are adequate.*

Proof. We prove first that $\downarrow$ is adequate on its own. Observe that

$$\neg p \equiv \neg(p \vee p) \equiv p \downarrow p$$

and

$$p \wedge q \equiv \neg\neg p \wedge \neg\neg q \equiv \neg(\neg p \vee \neg q) \equiv (\neg p) \downarrow (\neg q) \equiv (p \downarrow p) \downarrow (q \downarrow q).$$

But since $\neg$ and $\wedge$ form an adequate set of connectives we can now construct everything using $\downarrow$ alone.

We now prove that $|$ is adequate on its own. Observe that

$$\neg p \equiv \neg(p \wedge p) \equiv p \mid p$$

and

$$p \vee q \equiv \neg\neg p \vee \neg\neg q \equiv \neg(\neg p \wedge \neg q) \equiv (\neg p) \mid (\neg q) \equiv (p \mid p) \mid (q \mid q).$$

□

It can be proved that these are the only binary connectives which are adequate on their own. It would be possible to develop the whole of PL using for example just **nor**, and some mathematicians have done just that. But it renders PL truly non-user-friendly.

Example 1.6.5. We find a wff logically equivalent to $p \to q$ that uses only nors.

$$\begin{aligned} p \to q &\equiv \neg p \vee q \\ &\equiv \neg\neg(\neg p \vee q) \\ &\equiv \neg(\neg(\neg p \vee q)) \\ &\equiv \neg(\neg p \downarrow q) \\ &\equiv \neg((p \downarrow p) \downarrow q) \\ &\equiv ((p \downarrow p) \downarrow q) \downarrow ((p \downarrow p) \downarrow q.) \end{aligned}$$

Truth functions

We now have the apparatus we need to prove a result that is significant in circuit design. Let A be a wff with n atoms $p_1, \ldots, p_n$. The truth table for A has 2^n rows, where each row represents a specific assignment of truth

values to each of $p_1, \ldots, p_n$. The final column of the truth table contains the truth value assumed by A for each of these 2^n truth assignments.

A *truth function* is *any* table with rows all 2^n possible truth values and an output column which only takes the values T or F. There is no implication in the definition of a truth function that it need be the truth table *of* a wff. The following is an example of a truth function.

T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

We now come to our first major result.

Theorem 1.6.6 (Realization of truth functions)**.** *For each truth function with 2^n rows there is a wff with n atoms whose truth table is the given truth function.*

Proof. We shall prove this result in three steps constructing a wff A.

Step 1. Suppose that the truth function always outputs F. Define

$$A = (p_1 \wedge \neg p_1) \wedge \ldots \wedge p_n.$$

Then A has a truth table with 2^n rows that always outputs F.

Step 2. Suppose that the truth function outputs T *exactly once*. Let $v_1, \ldots, v_n$, where $v_i = T$ or F, be the assignment of truth values which yields the output T. Define a wff A as follows. It is a conjunction of exactly one of p_1 or $\neg p_1$, of p_2 or $\neg p_2$, ..., of p_n or $\neg p_n$ where p_i is chosen if $v_i = T$ and $\neg p_i$ is chosen if $v_i = F$. I shall call A a *basic conjunction* corresponding to the pattern of truth values $v_1, \ldots, v_n$. The truth table of A is the given truth function.

Step 3. Suppose that we are given now an arbitrary truth function not covered in steps 1 and 2 above. We construct a wff A whose truth table is the given truth function by taking a disjunction of all the basic conjunctions constructed from each row of the truth function that outputs T. □

The proof of the above theorem is best explained by means of an example.

Example 1.6.7. We construct a wff that has as truth table the following truth function.

T	T	T	$\mathbf{T}\,(1)$
T	T	F	F
T	F	T	F
T	F	F	$\mathbf{T}\,(2)$
F	T	T	$\mathbf{T}\,(3)$
F	T	F	F
F	F	T	F
F	F	F	F

We need only consider the rows that output T, which I have highlighted. I have also included a reference number that I shall use below. The basic conjunction corresponding to row (1) is

$$p \wedge q \wedge r.$$

The basic conjunction corresponding to row (2) is

$$p \wedge \neg q \wedge \neg r.$$

The basic conjunction corresponding to row (3) is

$$\neg p \wedge q \wedge r.$$

The disjunction of these basic conjunctions is

$$A = (p \wedge q \wedge r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge q \wedge r).$$

You should check that the truth table of A is the truth function given above.

1.7 Normal forms

A *normal form* is a particular way of writing a wff. We begin with a normal form that is a stepping stone to two others that are more important.

Negation normal form (NNF)

A wff is in *negation normal form (NNF)* if it is constructed using only $\wedge$, $\vee$ and literals. Recall that a *literal* is either an atom or the negation of an atom.

Proposition 1.7.1. *Every wff is logically equivalent to a wff in NNF.*

Proof. Let A be a wff in PL. First, replace any occurrences of $x \oplus y$ by $\neg(x \leftrightarrow y)$. Second, replace any occurrences of $x \leftrightarrow y$ by $(x \rightarrow y) \wedge (y \rightarrow x)$. Third, replace all occurrences of $x \rightarrow y$ by $\neg x \vee y$. Fourth, use de Morgan's laws to push all occurrences of negation through brackets. Finally, use double negation to ensure that only literals occur. □

Example 1.7.2. We convert $\neg(p \rightarrow (p \wedge q))$ into NNF using the method outlined in the proof of the above result.

$$\begin{aligned} \neg(p \rightarrow (p \wedge q)) &\equiv \neg(\neg p \vee (p \wedge q)) \\ &\equiv \neg\neg p \wedge \neg(p \wedge q) \\ &\equiv \neg\neg p \wedge (\neg p \vee \neg q) \\ &\equiv p \wedge (\neg p \vee \neg q). \end{aligned}$$

Disjunctive normal form (DNF)

We now come to the first of the two important normal forms. A wff that can be written as a disjunction of one of more terms each of which is a conjunction of one or more literals is said to be in *disjunctive normal form (DNF)*. Thus a wff in DNF has the following schematic shape

$$(\wedge \text{ literals}) \vee \ldots \vee (\wedge \text{ literals}).$$

Examples 1.7.3. Some special cases of DNF are worth highlighting because they often cause confusion.

1. A single atom p is in DNF.
2. A term such as $(p \wedge q \wedge \neg r)$ is in DNF.
3. The expression $p \vee q$ is in DNF. You should think of it as $(p) \vee (q)$.

Proposition 1.7.4. *Every wff is logically equivalent to one in DNF.*

Proof. Let A be a wff. Construct the truth table for A. Now apply Theorem 1.6.6. The wff that results is in DNF and logically equivalent to A. □

The method of proof used above can be used as a method for constructing DNF though it is a little laborious. Another method is to use logical equivalences. Let A be a wff. First convert A to NNF and then if necessary use the distributive laws to convert to a wff which is in DNF.

Example 1.7.5. We show how to convert $\neg(p \to (p \wedge q))$ into DNF using a sequence of logical equivalences. The first step is to replace $\to$. We use the fact that $x \to y \equiv \neg x \vee y$. This gives us $\neg(\neg p \vee (p \wedge q))$. Now use de Morgan's laws to push negation inside the brackets. This yields $\neg\neg p \wedge \neg(p \wedge q)$ and then $\neg\neg p \wedge (\neg p \vee \neg q)$. We now apply double negation to get $p \wedge (\neg p \vee \neg q)$. This is in NNF. Finally, we apply one of the distributive laws to get the $\vee$ out of the brackets. This yields $(p \wedge \neg p) \vee (p \wedge \neg q)$. This wff is in DNF and

$$\neg(p \to (p \wedge q)) \equiv (p \wedge \neg p) \vee (p \wedge \neg q).$$

Conjunctive normal form (CNF)

There is another normal form that plays an important role in the logic programming language PROLOG. A wff is in *conjunctive normal form (CNF)* if it is a conjunction of one or more terms each of which is a disjunction of one or more literals. It therefore looks a bit like the reverse of (DNF).

$$(\vee \text{ literals}) \wedge \ldots \wedge (\vee \text{ literals}).$$

Proposition 1.7.6. *Every wff is logically equivalent to one in CNF.*

Proof. Let A be our wff. Write $\neg A$ in DNF by Proposition 1.7.4. Now negate both sides of the logical equivalence and use double negation where necessary to obtain a wff in CNF. □

Example 1.7.7. A wff A has the following truth table.

T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	T

The truth table for $\neg A$ is

T	T	T	F
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	F

It follows that

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

is the DNF for $\neg A$. Negating both sides we get

$$A \equiv (\neg p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r).$$

This is the CNF for A.

Sometimes using logical equivalences is more efficient.

Example 1.7.8. The wff $(\neg x \wedge \neg y) \vee (\neg x \wedge z)$ is in DNF. It can easily be converted into CNF using one of the distributivity laws.

$$\begin{aligned}(\neg x \wedge \neg y) \vee (\neg x \wedge z) &\equiv ((\neg x \wedge \neg y) \vee \neg x) \wedge ((\neg x \wedge \neg y) \vee z) \\ &\equiv (\neg x \vee \neg x) \wedge (\neg y \vee \neg x) \wedge (\neg x \vee \neg z) \wedge (\neg y \wedge z).\end{aligned}$$

Exercises 3

These cover Sections 1.6 and 1.7.

---

1. Use known logical equivalences to transform each of the following wff first into NNF and then into DNF.

   (a) $(p \to q) \to p$.

   (b) $p \to (q \to p)$.

   (c) $(q \wedge \neg p) \to p$.

(d) $(p \vee q) \wedge r$.

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

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

2. Use known logical equivalences to transform each of the following into CNF.

(a) $(p \rightarrow q) \rightarrow p$.

(b) $p \rightarrow (q \rightarrow p)$.

(c) $(q \wedge \neg p) \rightarrow p$.

(d) $(p \vee q) \wedge r$.

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

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

3. Write $p \leftrightarrow (q \leftrightarrow r)$ in NNF.

4. The following truth table gives the semantics of three truth functions (a), (b) and (c). In each case, find a wff whose truth table is equal to the corresponding truth function.

p	q	r	(a)	(b)	(c)
T	T	T	T	F	T
T	T	F	T	T	F
T	F	T	T	F	T
T	F	F	T	T	F
F	T	T	F	T	T
F	T	F	T	T	F
F	F	T	T	F	T
F	F	F	T	F	F

5. Show that $p \oplus (q \oplus r) \equiv (p \oplus q) \oplus r$.

6. Do $\neg$ and $\rightarrow$ together form an adequate set of connectives?

7. Do $\vee$, $\wedge$ and $\rightarrow$ together form an adequate set of connectives?

1.8 P = NP?

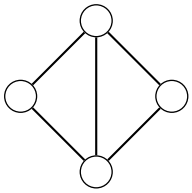
I cannot in this course go in to the details of this question but I can at least sketch out what it means and why it is important. The question whether **P** is equal to **NP** is the first of the seven *Millennium problems* that were posed by the Clay Mathematics Institute in 2000. Anyone who solves one of these problems wins a million dollars. So far only one of these problems has been solved: namely, the *Poincaré Conjecture* by Grigori Perelman who turned down the prize money. The other six problems require advanced mathematics just to understand what they are saying — except one. This is the question of whether **P** is equal to **NP**. It is intelligible to anyone who has taken a course in a subject called complexity theory that could easily be taught to second year maths and CS students, for example. I shall begin this sketch by explaining what we mean by **P** and **NP**.

How long does it take a program to solve a problem? As it stands this is too vague to admit an answer so we need to make it more precise. For concreteness, imagine a program that takes as input a whole number n and produces as output either the result that 'n is prime' or 'n is not prime'. So if you input 10 to the program it would tell you it was not prime but if you input 17 it would tell you that it was prime. Clearly, how long the program takes to solve this question depends on how big the number is that you input. A number with hundreds of digits is clearly going to take a lot longer for the program to work than one with just a few digits. Thus to say how long a program takes to solve a problem has to refer to the length of the input to that program. Now for each input of a fixed length, say m, the program might take varying amounts of time to produce an output. We agree to take the longest amount of time over all inputs of length m as a measure of how long it takes to process any input of length m. Of course, for many inputs of length m the program may be faster, but there will be some input of length m that takes the most time to process. The other issue we have to deal with is what we mean by 'time'. Your fancy MacBook Pro may be a lot faster than my Babbage Imperial. So instead of time we count the number of basic computational steps needed to transform input to output. It turns out that we don't really have to worry too much about what this means but it can be made precise using Turing machines. Thus with each program we can try to calculate its *time complexity profile*. This will be a function $m \mapsto f(m)$ where m is the length of the input and $f(m)$ is the maximum number of steps needed to transform any input of size m into an output. Calculating

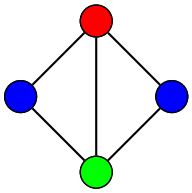
the time complexity profile exactly of even simple programs requires a lot of mathematical analysis, but fortunately we don't need an exact answer merely a good estimate. A program that has the time complexity profile $m \mapsto am$ where a is a number is said to run in *linear time*, if the time complexity profile is $m \mapsto am^2$ it is said to run in *quadratic time*, if the time complexity profile is $m \mapsto am^3$ it is said to run in *cubic time*. More generally, if the time complexity profile is $m \mapsto am^n$ for some n it is said to run in *polynomial time*. All the basic algorithms you learnt at school for adding, subtracting, multiplying and dividing numbers run in polynomial time. We define the class **P** to be all those problems that can be solved in polynomial time. These are essentially nice problems with nice (meaning fast) programs to solve them. What would constitute a nasty problem? This would be one whose time complexity profile looked like $m \mapsto 2^m$. This is nasty because by just increasing the size of the input by 1 doubles the amount of time needed to solve the problem. We now isolate those nasty problems that have a nice feature: that is, that any purported solution can be *checked* quickly, that is in polynomial time. Such problems constitute the class of *non-deterministic polynomial times problems* denoted by **NP**. A really nasty problem would be one where it is even difficult just to check a purported solution.

Clearly, **P** is contained in **NP** but there is no earthly reason why they should be equal. The problem is that currently (2016) no one has been able to prove that they are not equal. In 1971, Stephen Cook came up with an idea for resolving this question that shed new light on the nature of algorithms. His idea was that inside **NP** there should be a nastiest problem with an important, and ironically nice, property: if that problem could be shown to be in **P** then everything in **NP** would also have to be in **P** thus showing that $\mathbf{P} = \mathbf{NP}$. On the other hand, if it could be shown that this problem wasn't in **P** then we would have shown that $\mathbf{P} \neq \mathbf{NP}$. Thus Cook's problem would provide a sort of litmus-test for equality. Such a problem is called **NP**-*complete*. Given that we don't even know all the problems in **NP** Cook's idea might have sounded simply too good to be true. But in fact, he was able to prove a specific problem to be **NP**-complete: (fanfare) that problem is SAT — the *satisfiability problem in PL*. This probably all sounds very mysterious but in fact once you start studying complexity theory the mystery disappears. The reason that it is possible to prove that SAT is **NP**-complete boils down to the fact that PL is a sufficiently rich language to describe the behaviour of computers. Cook's result, known as *Cook's theorem*, is remarkable enough and explains the central importance of the satisfiability

in theoretical computers science. But there is more (no!). Thousands and thousands of problems have been shown to be **NP**-complete and so equivalent to SAT[1]. The *travelling salesman problem* is one well known example. I shall describe another interesting one here: *is a graph k-colourable?* A *graph* consists of *vertices* which are represented by circles and *edges* which are lines joining the circles (here always joining different circles). Vertices joined by an edge are said to be *adjacent*. The following is an example of a graph.



By a *colouring* of a graph we mean an assignment of colours to the vertices so that adjacent vertices have different colours. By a *k-colouring* of a graph we mean a colouring that uses at most k colours. Depending on the graph and k that may or may not be possible. The following is a 3-colouring of the above graph.



However, it is not possible to find a 2-colouring of this graph because there is a triangle of vertcies. The crucial point is this: if you do claim to have found a k-colouring of a graph I can easily check whether you are right. Thus the problem of k-colouring a graph is in **NP**. On the other hand, finding a k-colouring can involve a lot of work, including much back-tracking. It can be proved that this problem is **NP**-complete and so equivalent to SAT.

[1]The standard reference to all this is [3].

1.9 Valid arguments

Monty Python, part of the argument sketch

M = Man looking for an argument
A = Arguer

M. An argument isn't just contradiction.

A. It can be.

M. No it can't. An argument is a connected series of statements intended to establish a proposition.

A. No it isn't.

M. Yes it is! It's not just contradiction.

A. Look, if I argue with you, I must take up a contrary position.

M. Yes, but that's not just saying 'No it isn't.'

A. Yes it is!

M. No it isn't!

A. Yes it is!

M. Argument is an intellectual process. Contradiction is just the automatic gainsaying of any statement the other person makes.

(short pause)

A. No it isn't.

We have so far viewed PL as a low-level description language. This is its significance in the question as to whether **P** equals **NP** or not. We shall now touch on the other important application of PL which was actually the reason why PL was introduced in the first place. This is PL as a language for describing correct reasoning. Here is the idea.

Let $A_1, \ldots, A_n$ and B be wff. We say that B is a *consequence* of $A_1, \ldots, A_n$ if whenever all of $A_1, \ldots, A_n$ are true then B must be true as well. Equivalently, it is impossible for $A_1, \ldots, A_n$ to be true and B to be false. If B is a consequence of $A_1, \ldots, A_n$ we write

$$A_1, \ldots, A_n \vDash B$$

and we say this is a *valid argument*. This definition encapsulates many examples of logical reasoning. It is the foundation of mathematics and the basis of trying to prove that programs do what we claim they do. We shall see later that there are examples of logical reasoning that cannot be captured by PL and this will lead us to the generalization of PL called *first-order logic* or FOL.

Examples 1.9.1. Here are some examples of valid arguments.

1. $p, p \to q \vDash q$. We show that this is a valid argument. Here is the truth table for $p \to q$.

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

   We are only interested in the cases where both p and $p \to q$ are true.

p	q	$p \to q$
T	T	**T**

   We see that if p and $p \to q$ are true then q must be true.

2. $p \to q, \neg q \vDash \neg p$. We show this is a valid argument.

p	q	$\neg p$	$\neg q$	$p \to q$
T	T	F	F	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

   We are only interested in the cases where both $p \to q$ and $\neg q$ are true.

p	q	$\neg p$	$\neg q$	$p \to q$
F	F	T	**T**	**T**

   We see that if $p \to q$ and $\neg q$ are true then $\neg p$ must be true.

3. $p \vee q, \neg p \vDash q$. We show this is a valid argument.

p	q	$\neg p$	$p \vee q$
T	T	F	T
T	F	F	T
F	T	T	T
F	F	T	F

We are only interested in the cases where both $p \vee q$ and $\neg p$ are true.

p	q	$\neg p$	$p \vee q$
F	T	**T**	**T**

We see that if $p \vee q$ and $\neg p$ are true then q must be true.

4. $p \to q, q \to r \vDash p \to r$. We show this is a valid argument.

p	q	r	$p \to q$	$q \to r$	$p \to r$
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	F	T	T
T	F	F	F	T	F
F	T	T	T	T	T
F	T	F	T	F	T
F	F	T	T	T	T
F	F	F	T	T	T

We are only interested in the cases where both $p \to q$ and $q \to r$ are true.

p	q	r	$p \to q$	$q \to r$	$p \to r$
T	T	T	T	T	T
F	T	T	T	T	T
F	F	T	T	T	T
F	F	F	T	T	T

We see that in every case $p \to r$ is true. Thus the argument is valid.

We may always use truth tables in the way above to decide whether an argument is valid or not but it is quite laborious. The following result, however, enables us to do this in a very straightforward way.

Proposition 1.9.2. *We have that*

$$A_1, \ldots, A_n \vDash B$$

precisely when

$$\vDash (A_1 \wedge \ldots \wedge A_n) \rightarrow B.$$

Proof. The result is proved in two steps.

1. We prove that $A_1, \ldots, A_n \vDash B$ precisely when $A_1 \wedge \ldots \wedge A_n \vDash B$. This means that have have exactly one wff on the LHS of the semantic turnstile. Suppose that $A_1, \ldots, A_n \vDash B$ is a valid argument and that it is not the case that $A_1 \wedge \ldots \wedge A_n \vDash B$ is a valid argument. Then there is some assignment of truth values to the atoms that makes $A_1 \wedge \ldots \wedge A_n$ true and B false. But this means that each of $A_1, \ldots, A_n$ is true and B is false that contradicts that we are given that $A_1, \ldots, A_n \vDash B$ is a valid argument. Suppose that $A_1 \wedge \ldots \wedge A_n \vDash B$ is a valid argument but that $A_1, \ldots, A_n \vDash B$ is not a valid argument. Then some assignment of truth values to the atoms makes $A_1, \ldots, A_n$ true and B false. This means that $A_1 \wedge \ldots \wedge A_n$ is true and B is false. But this contradicts that we are given that $A_1 \wedge \ldots \wedge A_n \vDash B$ is a valid argument.

2. We prove that $A \vDash B$ precisely when $\vDash A \rightarrow B$. Suppose that $A \vDash B$ is a valid argument and that $A \rightarrow B$ is not a tautology. Then there is some assignment of the truth values to the atoms that makes A true and B false. But this contradicts that $A \vDash B$ is a valid argument. Suppose that $A \rightarrow B$ is a tautology and $A \vDash B$ is not a valid argument. Then there is some assignment of truth values to the atoms that makes A true and B false. But then the truth values of $A \rightarrow B$ is false from the way implication is defined and this is a contradiction.

□

Example 1.9.3. We show that $p, p \rightarrow q \vDash q$ is a valid argument by showing that $\vDash (p \wedge (p \rightarrow q)) \rightarrow q$ is a tautology.

p	q	$p \rightarrow q$	$p \wedge (p \rightarrow q)$	$(p \wedge (p \rightarrow q)) \rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Exercises 4

These cover all sections so far.

1. The binary connective *nor* is defined by $p \downarrow q = \neg(p \vee q)$. Show that $p \to q$ is logically equivalent to a wff in which the only binary connective that appears is nor.

2. Show that $\to$ and F together form an adequate set by showing now both $\neg p$ and $p \wedge q$ can be constructed from them. You should interpret F as being any contradiction.

3. Let $A = ((p \wedge q) \to r) \wedge (\neg(p \wedge q) \to r)$.

   (a) Draw the truth table for A.

   (b) Construct DNF using (a).

   (c) Draw the truth table for $\neg A$.

   (d) Construct DNF for $\neg A$ using (c).

   (e) Construct CNF for A using (d).

4. Let $A = ((p \wedge q) \to r) \wedge (\neg(p \wedge q) \to r)$.

   (a) Write A in NNF.

   (b) Use known logical equivalences applied to (a) to get DNF.

   (c) Use logical equivalences applied to (b) to get CNF.

   (d) Simplify A as much as possible using known logical equivalences.

5. Determine which of the following really are valid arguments.

   (a) $p \to q \vDash \neg q \vee \neg p$.

   (b) $p \to q, \neg q \to p \vDash q$.

   (c) $p \to q, r \to s, p \vee r \vDash q \vee s$.

   (d) $p \to q, r \to s, \neg q \vee \neg s \vDash \neg p \vee \neg r$.

6. Let $*$ be any binary connective that is supposed to be adequate on its own. [By Question 7 of Exercises 2, there are 16 possible values of $*$.]

   (a) Explain why $T * T = F$.

   (b) Explain why $F * F = T$.

   (c) Explain why $T * F = F * T$.

   Deduce that the only possible values for $*$ are $\downarrow$ and $|$.

7. Use [16] to construct a truth table for the following wff.

$$A(p,q,r,s) = (p \vee q \vee r \vee s) \wedge \neg(p \wedge q) \wedge \neg(p \wedge r) \wedge \neg(p \wedge s) \\ \wedge \neg(q \wedge r) \wedge \neg(q \wedge s) \wedge \neg(r \wedge s).$$

   Describe in words the meaning of $A(p,q,r,s)$.

8. This question deals with the following 2×2 array, whose cell numbers are as indicated (the array is actually empty)

1	2
3	4

   For $1 \leq i \leq 4$ and $1 \leq j \leq 2$ define the atom

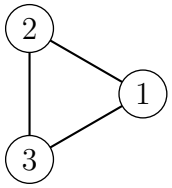
$$p_{i,j} = \text{'The cell } i \text{ contains the digit } j\text{'}$$

   Consider the following wff

$$\begin{aligned} P \;&=\; (p_{1,1} \oplus p_{1,2}) \wedge (p_{2,1} \oplus p_{2,2}) \wedge (p_{3,1} \oplus p_{3,2}) \wedge (p_{4,1} \oplus p_{4,2}) \\ &\wedge\; (p_{1,1} \vee p_{2,1}) \wedge (p_{3,1} \vee p_{4,1}) \wedge (p_{1,2} \vee p_{2,2}) \wedge (p_{3,2} \vee p_{4,2}) \\ &\wedge\; (p_{1,1} \vee p_{3,1}) \wedge (p_{2,1} \vee p_{4,1}) \wedge (p_{1,2} \vee p_{3,2}) \wedge (p_{2,2} \vee p_{4,2}). \end{aligned}$$

   Determine precisely when P is satisfiable and explain your result in everyday English. [Hint: Don't even think about using truth tables].

9. The picture below shows a *graph* that has *vertices* marked by the numbered circles and *edges* marked by the lines.



We say that two vertices are *adjacent* if they are joined by an edge. The vertices are to be coloured either blue or red under certain constraints that will be described by means of a wff in PL denoted by A. The following is a list of atoms and what they mean.

- p. Vertex 1 is blue.
- q. Vertex 1 is red.
- r. Vertex 2 is blue.
- s. Vertex 2 is red.
- u. Vertex 3 is blue.
- v. Vertex 3 is red.

Here is the wff A constructed from these atoms.

$$\begin{gathered}(p \oplus q) \wedge (r \oplus s) \wedge (u \oplus v) \wedge \\ (q \to (r \wedge u)) \wedge (p \to (s \wedge v)) \wedge \\ (s \to (p \wedge u)) \wedge (r \to (q \wedge v)) \wedge \\ (v \to (p \wedge r)) \wedge (u \to (q \wedge s))\end{gathered}$$

(a) Translate A into English in as pithy and precise a way as possible.

(b) Use [16] to construct a truth table of A.

(c) Interpret this truth table.

1.10 Truth trees

All problems about PL can be answered using truth tables. But there are two problems.

1. The method of truth tables is hard work.
2. The method of truth tables does not generalize to first-order logic.

In this section, we shall describe an algorithm that is often more efficient than truth tables and which can also be generalized to first-order logic. This is the method of truth trees. It will speed up the process of answering the following questions.

- Determining whether a wff is satisfiable or not.
- Determining whether a set of wff is satisfiable or not.
- Determining whether a wff is a tautology or not.
- Determining whether an argument is valid or not.
- Converting a wff into DNF.

It is based on the following ideas.

- Given a wff A that we wish to determine is satisfiable or not we start by assuming A is satisfiable and work backwards.
- We use a data structure, a tree, to keep track efficiently of all possibilities that occur.
- We break A into smaller pieces and so the algorithm is a *divide and conquer* algorithm.
- What is not true is false. Thus we need only keep track of what is true and any other cases will automatically be false.

The starting point is to consider the various possible shapes that a wff A can have. Observe that since $X \oplus Y \equiv \neg(X \leftrightarrow Y)$, I shall not mention $\oplus$ explicitly. There are therefore nine possibilities for A.

1. $X \wedge Y$.
2. $\neg(X \vee Y)$.
3. $\neg(X \rightarrow Y)$.
4. $\neg\neg X$.
5. $X \vee Y$.
6. $\neg(X \wedge Y)$.
7. $X \rightarrow Y$.
8. $X \leftrightarrow Y$.

9. $\neg(X \leftrightarrow Y)$.

We now introduce a graphical way of representing the truth values of A in terms of the truth values for X and Y. We introduce two kinds of graphical rules.

- The *α-rule* is non-branching/and-like. If $A = X * Y$ then this rule looks like

$$\begin{array}{c} X * Y \\ | \\ X \\ Y \end{array}$$

  This means that $X * Y$ is true precisely when both X and Y are true.

- The *β-rule* is branching/or-like. If $A = X * Y$ then this rule looks like

$$\begin{array}{c} X * Y \\ \diagup\ \diagdown \\ X \quad Y \end{array}$$

  This means that $X * Y$ is true precisely when X or Y is true.

We now list the truth tree rules for the nine forms of the wff given above.

α-rules

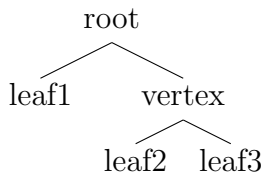
$$\begin{array}{c} X \wedge Y \\ | \\ X \\ Y \end{array} \qquad \begin{array}{c} \neg(X \vee Y) \\ | \\ \neg X \\ \neg Y \end{array} \qquad \begin{array}{c} \neg(X \rightarrow Y) \\ | \\ X \\ \neg Y \end{array} \qquad \begin{array}{c} \neg\neg X \\ | \\ X \end{array}$$

β-rules

$$\begin{array}{c} X \vee Y \\ \diagup\ \diagdown \\ X \quad Y \end{array} \qquad \begin{array}{c} \neg(X \wedge Y) \\ \diagup\ \diagdown \\ \neg X \quad \neg Y \end{array} \qquad \begin{array}{c} X \rightarrow Y \\ \diagup\ \diagdown \\ \neg X \quad Y \end{array} \qquad \begin{array}{c} X \leftrightarrow Y \\ \diagup\ \diagdown \\ X \quad \neg X \\ Y \quad \neg Y \end{array} \qquad \begin{array}{c} \neg(X \leftrightarrow Y) \\ \diagup\ \diagdown \\ X \quad \neg X \\ \neg Y \quad Y \end{array}$$

We shall now combine these small trees to construct a truth tree of a wff A that will enable us to determine when A is satisfiable and how. The

following definition dealing with trees is essential. A *branch* of a tree is a path that starts at the root and ends at a leaf. For example in the tree below



there are three branches. The key idea in what follows can now be stated: *truth flows down the branches and different branches represent alternative possibilities and so should be regarded as being combined by means disjunctions.* I shall describe the full truth tree algorithm once I have worked my way through some illustrative examples.

> Don't confuse *parse trees* which are about syntax with *truth trees* which are about semantics.

Example 1.10.1. Find all truth assignments that make the wff $A = \neg p \to (q \wedge r)$ true using truth trees. The first step is to place A at the root of what will be the truth tree

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

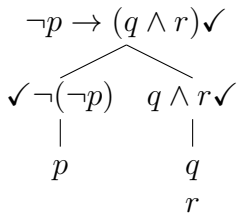
We now use the branching rule for $\to$ to get

$$\begin{array}{c} \neg p \to (q \wedge r)\checkmark \\ \diagup \quad \diagdown \\ \neg(\neg p) \quad q \wedge r \end{array}$$

I have used the symbol $\checkmark$to indicate that I have *used* the occurrence of the wff $\neg p \to (q \wedge r)$. It is best to be systematic and so I shall work from left-to-right. We now apply the truth tree rule for double negation to get

$$\begin{array}{c} \neg p \to (q \wedge r)\checkmark \\ \diagup \quad \diagdown \\ \checkmark\neg(\neg p) \quad q \wedge r \\ | \\ p \end{array}$$

where once again I have used the check symbol $\checkmark$to indicate that that occurrence of a wff has been used. Finally, I apply the truth tree rule for conjunction to get



There are now no further truth tree rules I can apply and so we say that the truth tree is *finished*. We deduce that the root A is true precisely when either p is true or (q and r are both true). In other words, $A \equiv p \vee (q \wedge r)$, which is in DNF. You can easily check that this contains exactly the same information as the rows of the truth table of A that output T. By our idea above we know that all other truth values must be F.

Before giving some more complex examples, let me highlight some important points (all of which can be proved).

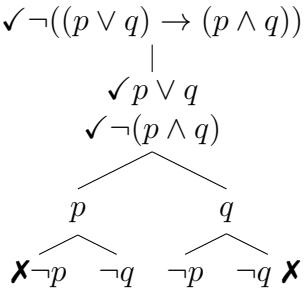
- It is the branches of the truth tree that contain information about the truth table. Each branch contains information about one or more rows of the truth table.
- It follows that all the literals on a branch must be true.
- Thus if an atom and its negation occur on the same branch then there is a contradiction. That branch is then *closed* by placing a ✗ at its leaf. No further growth takes place at a closed leaf.

The next example illustrates an important point about applying β-rules.

Example 1.10.2. Find all the truth assignments that satisfy

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

Here is the truth tree for this wff.



The truth tree is finished and there are two open branches (those branches not marked with a ✗.) The first branch tells us that p and $\neg q$ are both true and the second tells is that $\neg p$ and q are both true. It follows that the wff has the DNF

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

The key point to observe in this example is that when I applied the β-rule to the wff $\neg(p \wedge q)$ *I applied it to all branches that contained that wff.* This is crucially important since *possibilities multiply.*

The above example leads to the following *strategy.*

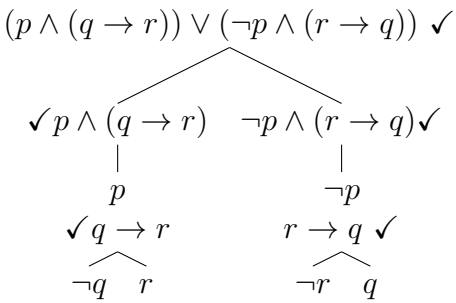
> *Apply α-rules before β-rules since the application of β-rules leads to the tree gaining more branches and subsequent applications of any rule must be appended to every branch.*

The following examples are for illustrative purposes.

Example 1.10.3. Find all satisfying truth assignments to

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

Here is the truth tree of this wff.



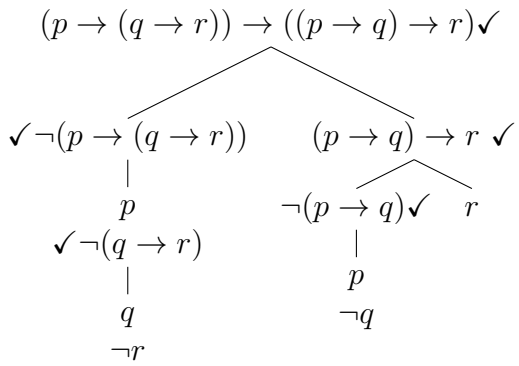
There are four branches and all branches are open. These lead to the DNF

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

Example 1.10.4. Write

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

in DNF. Here is the truth tree of this wff.



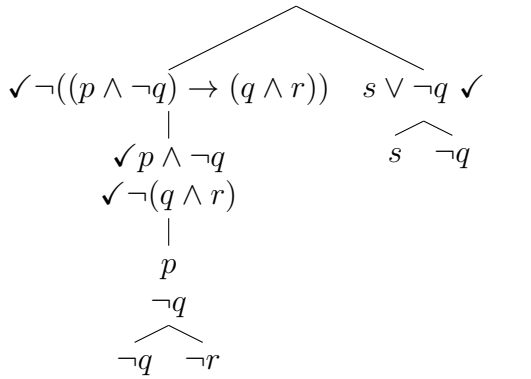
There are three branches all open. These lead to the DNF

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

Example 1.10.5. Write

$$[(p \wedge \neg q) \rightarrow (q \wedge r)] \rightarrow (s \vee \neg q),$$

which contains four atoms, in DNF. Here is the truth tree of this wff.



There are four branches all open. These lead to the DNF

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

We say that a set of wff $A_1, \ldots, A_n$ is *satisfiable* if there is at least one single truth assigment that makes them all true. This is equivalent to saying that $A_1 \wedge \ldots \wedge A_n$ is satisfiable. Thus to show that $A_1, \ldots, A_n$ is satisfiable simply list these wff as the root of a truth tree.

It's important to remember that truth trees are an algorithm for finding those truth assignments that make a wff true. This leads to the following important result.

Proposition 1.10.6. *To show that X is a tautology show that the truth tree for $\neg X$ has the property that every branch closes.*

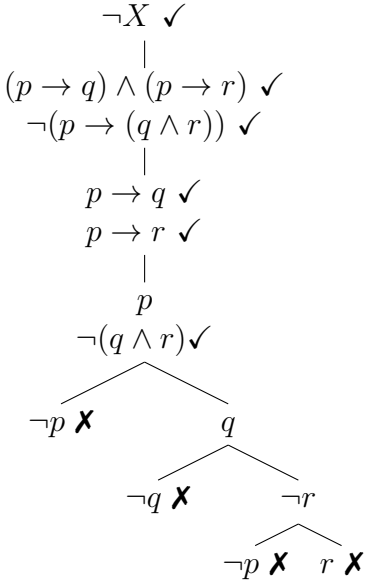
Proof. If the truth tree for $\neg X$ has the property that every branch closes then $\neg X$ is not satisfiable. This means that $\neg X$ is a contradiction. Thus X is a tautology. □

Here are some examples of showing that a wff is a tautology.

Example 1.10.7. Determine whether

$$X = ((p \to q) \wedge (p \to r)) \to (p \to (q \wedge r))$$

is a tautology or not. We begin the truth tree with $\neg X$.

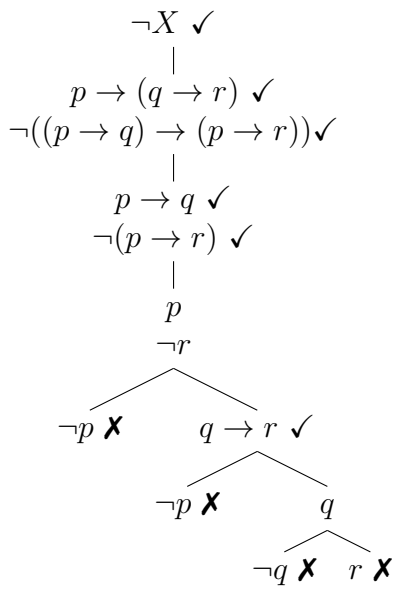


The tree for $\neg X$ closes and so $\neg X$ is a contradiction. Thus X is a tautology.

Example 1.10.8. Determine whether

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

is a tautology or not.

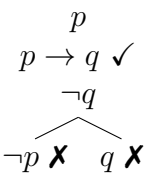


The tree for $\neg X$ closes and so $\neg X$ is a contradiction. Thus X is a tautology.

We can also use truth trees to determine whether an argument is valid or not. To see how, we use the result proved earlier that $A_1, \ldots, A_n \vDash B$ precisely when $\vDash (A_1 \wedge \ldots \wedge A_n) \to B$. Thus to show that an argument is valid using a truth tree, we could place $\neg[(A_1 \wedge \ldots \wedge A_n) \to B]$ at its root. But this is logically equivalent to $A_1 \wedge \ldots \wedge A_n \wedge \neg B$. Thus we need only list $A_1, \ldots, A_n, \neg B$ at the root of our truth tree. If every branch closes the corresponding argument is valid, and if there are open branches then the argument is not logically valid.

Examples 1.10.9. We use truth trees to show that our simple examples of arguments really are valid.

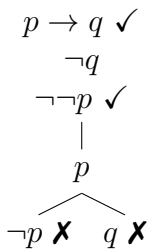
1. $p, p \to q \vDash q$ is a valid argument[2].



   The tree closes and so the argument is valid.

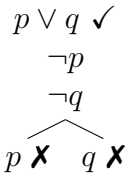
2. $p \to q, \neg q \vDash \neg p$ is a valid argument[3].

[2]Known as *modus ponens.*
[3]Known as *modus tollens.*



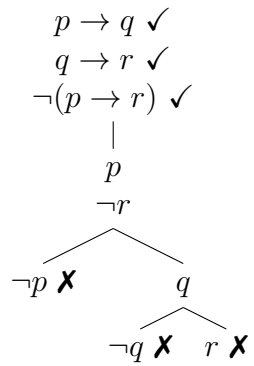
The tree closes and so the argument is valid.

3. $p \vee q, \neg p \vDash q$ is a valid argument[4].



The tree closes and so the argument is valid.

4. $p \to q, q \to r \vDash p \to r$ is a valid argument[5].



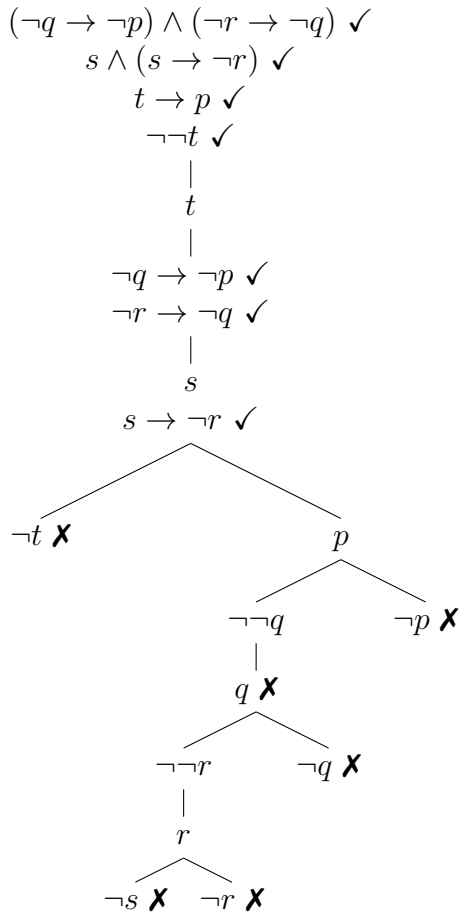
The tree closes and so the argument is valid.

Example 1.10.10. Show that the following

$$(\neg q \to \neg p) \wedge (\neg r \to \neg q), s \wedge (s \to \neg r), t \to p \vDash \neg t$$

is a valid argument.

[4]Known as *disjunctive syllogism.*
[5]Known as *hypothetical syllogism.*



The tree closes and so the argument is valid.

Truth tree algorithm

Input: wff X.

Procedure: Place X at what will be the root of the truth tree. Depending on its shape, apply either an α-rule or a β-rule. Place a ✓against X to indicate that it has been *used*.

Now repeat the following:

- *Close* any branch that contains an atom and its negation by placing a cross ✗ beneath the leaf defining that branch.
- If all branches are closed then *stop* since the tree is now *finished and closed*.
- It not all branches are closed but the tree contains only literals or used wff then *stop* since the tree is now *finished and open*.
- If the tree is not finished then choose an unused wff Y which is not a literal. Now do the following: **for each open branch that contains Y append the effect of applying either the α-rule or β-rule to Y, depending on which is appropriate, to the leaf of that branch.**

Output:

- If the tree is finished and closed then X is a contradiction.
- If the tree is finished and open then X is satisfiable. We may find all the truth assignments that make X true as follows: for each open branch in the finished tree for X assign the value T to all the literals in that branch. If any atoms are missing from this branch then they may be assigned truth values arbitrarily.

Applications of truth trees

1. To prove that X is a *tautology*, show that the finished truth tree for $\neg X$ is closed.

2. To prove that X is a *satisfiable*, show that the finished truth tree for X is open.

3. To prove that $A_1, \ldots, A_n \models B$ is a *valid argument*, place

$$A_1, \ldots, A_n, \neg B$$

at the root of the tree and show that the finished truth tree is closed.

4. To put X into *DNF*, construct the truth-tree for X. Assume that when finished it is open. For each open branch i, construct the conjunction of the literals that appear on that branch C_i. Then form the disjunction of the C_i.

Notation. Suppose that the finished truth tree with root $A_1, \ldots, A_n, \neg B$ closes. Then we write

$$A_1, \ldots, A_n \vdash B.$$

The following theorem says that the truth tree algorithm does exactly what it is supposed to do.

Theorem 1.10.11 (Soundness and completeness)**.**

$$A_1, \ldots, A_n \vdash B \text{ } \textit{if and only if} \text{ } A_1, \ldots, A_n \models B.$$

The symbol $\models$ is about *truth* and the symbol $\vdash$ is about *proof*. The above theorem says that truth can be discovered via proof.

Exercises 5

Section 1.10.

---

1. Determine whether the following arguments are valid or not using truth trees.

(a) $p \to q, r \to s, p \vee r \models q \vee s$.

(b) $p \to q, r \to s, \neg q \vee \neg s \models \neg p \vee \neg r$.

(c) $p \to q, r \to s, p \vee \neg s \models q \vee \neg r$.

2. Show that the following are tautologies using truth trees.

(a) $q \to (p \to q)$.

(b) $[(p \to q) \wedge (q \to r)] \to (p \to r)$.

(c) $[(p \to q) \wedge (p \to r)] \to (p \to (q \wedge r))$.

(d) $[((p \to r) \wedge (q \to r)) \wedge (p \vee q)] \to r$.

3. Determine whether the following sets of wff are satisfiable and where they are find all truth assignments making them true using truth trees.

(a) $A \to \neg(B \wedge C)$, $(D \vee E) \to G$, $G \to \neg(H \vee I)$, $\neg C \wedge E \wedge H$.

(b) $(A \vee B) \to (C \wedge D)$, $(D \vee E) \to G$, $A \vee \neg G$.

(c) $(A \to B) \wedge (C \to D)$, $(B \to D) \wedge (\neg C \to A)$, $(E \to G) \wedge (G \to \neg D)$, $\neg E \to E$.

4. In Chapter 1 of *Winnie-the-Pooh* by A. A. Milne, 1926, Pooh makes the following argument.

(a) There is a buzzing.

(b) If there is a buzzing then somebody is making the buzzing.

(c) If somebody is making the buzzing then somebody is bees.

(d) If somebody is bees then there is honey.

(e) If there is honey then there is honey for Pooh to eat.

(f) Therefore there is honey for Pooh to eat.

Using the letters $p, q, r \ldots$ express this argument in symbolic form and use truth trees to determine whether Pooh's argument is valid.

5. A student of logic reasons as follows

> "If I go to the lecture tomorrow then I (must) get up early and if I go to the pub tonight then I (will) stay up late. If I stay up late and I get up early then I (will only) have five hours sleep. It is inconceivable that (*treat as negation*) I have (only) five hours of sleep. Therefore either I (will) not go to the lecture tomorrow (inclusive) or I (will) not go to the pub tonight."

Assign atoms as follows:

p I go to the lecture tomorrow.

q I get up early.

r I go to the pub tonight.

s I stay up late.

t I have five hours sleep.

[Words in brackets are included for grammatical rather than logical reasons]. Express the student's argument in symbolic form and determine whether the argument is valid or not using truth trees.

6. The following is due to Charles Lutwidge Dodgson (1832–1898), an Oxford mathematician.

   (a) All the dated letters in this room are written on blue paper.

   (b) None of them is in black ink, except those that are written in the third person.

   (c) I have not filed any of those that I can read.

   (d) None of those that are written on one sheet are undated.

   (e) All of those that are not crossed out are in black ink.

   (f) All of those written by Brown begin with 'Dear Sir'.

   (g) All of those that are written on blue paper are filed.

   (h) None of those that are written on more than one sheet are crossed out.

   (i) None of those that begin with 'Dear Sir' are written in the third person.

(j) Therefore: I cannot read any of Brown's letters.

Assign atoms as follows:

p The letter is dated.

q The letter is written on blue paper.

r The letter is written in black ink.

s The letter is written in the third person.

t The letter is filed.

u I can read the letter.

v The letter is written on one sheet.

w The letter is crossed out.

x The letter is written by Brown.

y The letter begins with 'Dear Sir'.

You will will need to do a certain amount of interpretation to convert (a)–(j) into symbolic PL form using these atoms. Hence formalize the above argument and use truth trees to determine its validity.

Exercises 6

Revision of Chapter 1

1.

(a) Construct a parse tree for the following wff

$$A = (p \leftrightarrow q) \rightarrow ((p \wedge r) \leftrightarrow (q \wedge r)).$$

Construct a truth-table for A.

(b) Write $(p \rightarrow q) \rightarrow p$ in conjunctive normal form.

(c) Write $p \leftrightarrow q$ in disjunctive normal form.

(d) Prove that

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

2. You must use truth trees to answer this question.

(a) Prove that

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

is a tautology.

(b) Prove that the argument

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

is valid.

(c) Write the wff below in disjunctive normal form

$$(q \wedge r) \to (p \leftrightarrow (\neg q \vee r)).$$

3.

(a) Construct a parse tree for the following wff

$$A = (p \wedge (q \to r)) \vee (\neg p \wedge (r \to q)).$$

(b) Construct a truth table for A.

(c) Write A in disjunctive normal form.

(d) Write A in conjunctive normal form.

4. You must use truth trees to answer this question.

(a) Prove that

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

is a tautology.

(b) Prove that the argument

$$(p \to q) \wedge (r \to s) \wedge (p \vee \neg s) \models q \vee \neg r.$$

is valid.

(c) Write the wff below in disjunctive normal form

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

Chapter 2

Boolean algebras

> *6.5. When the answer cannot be put into words, neither can the question be put into words. The* riddle *does not exist. If a question can be framed at all, it is also* possible *to answer it.* — Tractatus Logico-Philosophicus, Ludwig Wittgenstein.

This chapter can be read after Chapter 1 or after Chapters 1 and 3. Unlike the other two chapters, this one is not about logic per se but about the algebra that arises from PL. I have included it because the main application is to circuit design: specfically, the kinds of circuits that are needed to build computers. It therefore provides a bridge between logic and the real world.

2.1 Definition of Boolean algebras

I shall begin by listing some logical equivalences in propositional logic which were proved in the exercises. I shall use T to mean any tautology and F to mean any contradiction.

1. $(p \vee q) \vee r \equiv p \vee (q \vee r)$.
2. $p \vee q \equiv q \vee p$.
3. $p \vee F \equiv p$.
4. $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$.
5. $p \wedge q \equiv q \wedge p$.

6. $p \wedge T \equiv p$.
7. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.
8. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.
9. $p \vee \neg p \equiv T$.
10. $p \wedge \neg p \equiv F$.

We now make the following replacements in the above list.

PL	BA
$\equiv$	$=$
$\vee$	$+$
$\wedge$	$\cdot$
$\neg$	$^-$
T	1
F	0

to get exactly the definition of a Boolean algebra. This is now an *algebraic system* rather than a *logical system*. It is adapted to dealing with truth tables and so with circuits, as we shall see.

Formally, a *Boolean algebra* is defined by the following data $(B, +, \cdot, ^-, 0, 1)$ where B is a set, we shall say more about sets later, that carries the structure, $+$ and $\cdot$ are binary operations, meaning that they have two, ordered inputs and one output, $a \mapsto \bar{a}$ is a unary operation, meaning that it has one input and one output, and two special elements of B: namely, 0 and 1. In addition, the following ten axioms are required to hold.

(B1) $(x + y) + z = x + (y + z)$.

(B2) $x + y = y + x$.

(B3) $x + 0 = x$.

(B4) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

(B5) $x \cdot y = y \cdot x$.

(B6) $x \cdot 1 = x$.

(B7) $x \cdot (y + z) = x \cdot y + x \cdot z$.

(B8) $x + (y \cdot z) = (x + y) \cdot (x + z)$.

(B9) $x + \bar{x} = 1$.

(B10) $x \cdot \bar{x} = 0$.

These axioms are organized as follows. The first group of three (B1), (B2), (B3) deals with the properties of $+$ on its own: brackets, order, special element. The second group of three (B4), (B5), (B6) deals with the properties of $\cdot$ on its own: brackets, order, special element. The third group (B7), (B8) deals with how $+$ and $\cdot$ interact, with axiom (B8) being odd looking. The final group (B9), (B10) deals with the properties of $a \mapsto \bar{a}$, called *complementation*. On a point of notation, we shall usually write xy rather than $x \cdot y$.

Remark 2.1.1. Do I have to remember these axioms? No. In the exam paper I will list these axioms if they are needed. You do, however, have to familiarize yourself with them and learn how to use them.

Remark 2.1.2. The Boolean algebra associated with PL is called the *Lindenbaum algebra*. The only issue is how to convert $\equiv$ into $=$. Denote by **Prop** the set of all wff in propositional logic. There are two answers.

> **Answer 1.** Not correct but not far off. The set **Prop** is *essentially* a Boolean algebra. You need to insert the word 'essentially' in order to cover yourself to avoid legal action by outraged Boolean algebraists.

> **Answer 2.** The correct one. Logical equivalence is an equivalence relation on the set **Prop** and is a congruence with respect to the operations $\vee$, $\wedge$ and $\neg$. The quotient of **Prop** by $\equiv$ is then the Boolean algebra. This is the correct answer but needs a lot of unpacking before it can be understood which will not be given here.

The Boolean algebra we shall use in circuit design is the 2-element Boolean algebra $\mathbb{B}$. It is defined as follows. Put $\mathbb{B} = \{0, 1\}$. We define operations $^-$,

$\cdot$, and $+$ by means of the following tables. They are the same as the truth table for $\neg$, $\wedge$ and $\vee$ except that we replace T by 1 and F by 0.

x	$\bar{x}$
1	0
0	1

x	y	$x \cdot y$
1	1	1
1	0	0
0	1	0
0	0	0

x	y	$x + y$
1	1	1
1	0	1
0	1	1
0	0	0

Proposition 2.1.3. *With the definitions above* $(\mathbb{B}, +, \cdot, ^-, 0, 1)$ *really is a Boolean algebra.*

We shall mainly be working with expressions in x, y, z where the variables are *Boolean variables* meaning that $x, y, z \in \mathbb{B}$.

Boolean algebras form an algebra that is similar to but also different from the kind of algebra you learnt at school. Just how different is illustrated by the following results.

Proposition 2.1.4. *Let B be a Boolean algebra and let $a, b \in B$.*

1. $a^2 = a \cdot a = a$. *Idempotence.*
2. $a + a = a$.
3. $a \cdot 0 = 0$.
4. $1 + a = 1$.
5. $a = a + a \cdot b$. *Absorption law.*
6. $a + b = a + \bar{a} \cdot b$. *Absorption law.*

Proof. (1)

$$\begin{aligned} a &= a \cdot 1 \text{ by (B6)} \\ &= a \cdot (a + \bar{a}) \text{ by (B9)} \\ &= a \cdot a + a \cdot \bar{a} \text{ by (B7)} \\ &= a^2 + 0 \text{ by (B10)} \\ &= a^2 \text{ by (B3).} \end{aligned}$$

(2) is proved in a very similar way to (1). Use the fact that the Boolean algebra axioms come in pairs where $\cdot$ and $+$ are interchanged and 0 and 1 are interchanged. This is an aspect of what is called *duality*.

(3)

$$\begin{aligned} a \cdot 0 &= a \cdot (a \cdot \bar{a}) \text{ by (B10)} \\ &= (a \cdot a) \cdot \bar{a} \text{ by (B4)} \\ &= a \cdot \bar{a} \text{ by (1) above} \\ &= 0 \text{ by (B10).} \end{aligned}$$

(4) The dual proof to (3).

(5)

$$\begin{aligned} a + a \cdot b &= a \cdot 1 + a \cdot b \text{ by (B6)} \\ &= a \cdot (1 + b) \text{ by (B7)} \\ &= a \cdot 1 \text{ by (4) above} \\ &= a \text{ by (B6).} \end{aligned}$$

(6)

$$\begin{aligned} a + b &= a + 1b \text{ by (B6)} \\ &= a + (a + \bar{a})b \text{ by (B9)} \\ &= a + ab + \bar{a}b \text{ by (B7) and (B5)} \\ &= a + \bar{a}b \text{ by (5) above} \end{aligned}$$

□

These properties can be used to simplify Boolean expressions.

Example 2.1.5. Simplify

$$x + yz + \bar{x}y + x\bar{y}z.$$

At each stage in our argument we are explicit about what properties we are using. I shall use associativity of addition throughout to avoid too many

brackets.

$$\begin{aligned}
x + yz + \bar{x}y + x\bar{y}z &= (x + \bar{x}y) + yz + x\bar{y}z \text{ by commutativity} \\
&= (x + y) + yz + x\bar{y}z \text{ by absorption} \\
&= x + (y + yz) + x\bar{y}z \text{ by associativity} \\
&= x + y + x\bar{y}z \text{ by absorption} \\
&= x + (y + \bar{y}(xz)) \text{ by commutativity and associativity} \\
&= x + (y + xz) \text{ by absorption} \\
&= (x + xz) + y \text{ by commutativity and associativity} \\
&= x + y \text{ by absorption.}
\end{aligned}$$

The obvious question about calculations such as these is how do you know what to aim for? There is a diagrammatic way of thinking about Boolean algebras that uses sets which will help us answer this question.

2.2 Set theory

A set is a new data type that will also be important when we come to study first-order logic. Set theory was invented by Georg Cantor (1845–1918) in the last quarter of the nineteenth century.

Basic definitions

Set theory begins with two deceptively simple definitions on which everything is based.

1. A *set* is a collection of objects, called *elements*, which we wish to regard as a whole.

2. Two sets are *equal* precisely when they contain the same elements.

It is customary to use capital letters to name sets such as $A, B, C \ldots$ or fancy capital letters such as $\mathbb{N}, \mathbb{Z} \ldots$ with the elements of a set usually being denoted by lower case letters. If x *is an element of* the set A then we write $x \in A$ and if x *is not an element of* the set A then we write $x \notin A$. To indicate that some things are to be regarded as a set rather than just as isolated individuals, we enclose them in 'curly brackets' { and } formally called braces. Thus the set of suits in a pack of cards is $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$.

A set should be regarded as a bag of elements, and so the order of the elements within the set is not important. Thus $\{a, b\} = \{b, a\}$. Perhaps more surprisingly, repetition of elements is ignored. Thus $\{a, b\} = \{a, a, a, b, b, a\}$. Sets can be generalized to what are called *multisets* where repetition is recorded.

The set $\{\}$ is empty and is called the *empty set*. It is given a special symbol $\emptyset$, which is not the Greek letter ϕ or Φ, but is allegedly the first letter of a Danish word meaning 'desolate'. The symbol $\emptyset$ means exactly the same thing as $\{\}$. Observe that $\emptyset \neq \{\emptyset\}$ since the empty set contains no elements whereas the set $\{\emptyset\}$ contains one element.

The number of elements a set contains is called its *cardinality* denoted by $|X|$. A set is *finite* if it only has a finite number of elements, otherwise it is *infinite.* A set with exactly one element is called a *singleton set.*

We can sometimes define infinite sets by using curly brackets but then, because we cannot list all elements in an infinite set, we use '...' to mean 'and so on in the obvious way'. This can also be used to define big finite sets where there is an obvious pattern. However, the most common way of describing a set is to say what properties an element must have to belong to it. By a *property* we mean a sentence containing a variable such as x so that the sentence becomes true or false depending on what we substitute for x. For example, the sentence 'x is an even natural number' is true when x is replaced by 2 and false when x is replaced by 3. If we abbreviate 'x is an even natural number' by $E(x)$ then the set of even natural numbers is the set of all natural numbers n such that $E(n)$ is true. This set is written $\{x\colon\ E(x)\}$ or $\{x \mid E(x)\}$. More generally, if $P(x)$ is any property then $\{x\colon\ P(x)\}$ means 'the set of all things x that satisfy the condition P'. Properties are important in first-order logic. Here are some examples of sets defined in various ways.

Examples 2.2.1.

1. $D =$ { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }, the set of the days of the week. This is a small finite set and so we can conveniently list all its elements.

2. $M =$ { January, February, March, ..., November, December }, the set of the months of the year. This is a finite set but we did not want to write down all the elements explicitly so we wrote '...' instead.

3. $A = \{x\colon\ x$ is a prime natural number$\}$. We here define a set by describing the properties that the elements of the set must have. In this

case $P(x)$ is the statement 'x is a prime natural number' and those natural numbers x are admitted membership to the set when they are indeed prime.

The following notation will be useful when we come to study first-order logic.

Examples 2.2.2.

1. The set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ of all *natural numbers*. Caution is required here since some books eccentrically do not regard 0 as a natural number.

2. The set $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ of all *integers*. The reason $\mathbb{Z}$ is used to designate this set is because 'Z' is the first letter of the word 'Zahl', the German for number.

3. The set $\mathbb{Q}$ of all *rational numbers*. That is those numbers that can be written as quotients of integers with non-zero denominators.

4. The set $\mathbb{R}$ of all *real numbers*. That is all numbers which can be represented by decimals with potentially infinitely many digits after the decimal point.

Given a set A, a new set B can be formed by *choosing* elements from A to put into B. We say that B is a *subset* of A, denoted by $B \subseteq A$. In mathematics, the word 'choose', unlike in polite society, also includes the possibility of *choosing nothing* and the possibility of *choosing everything*. In addition, there does not need to be any rhyme or reason to your choices: you can pick elements 'at random' if you want. If $A \subseteq B$ and $A \neq B$ then we say that A is a *proper subset* of B.

Examples 2.2.3.

1. $\emptyset \subseteq A$ for every set A, where we choose no elements from A.

2. $A \subseteq A$ for every set A, where we choose all the elements from A.

3. $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$. Observe that $\mathbb{Z} \subseteq \mathbb{Q}$ because an integer n is equal to the rational number $\frac{n}{1}$.

4. $\mathbb{E}$, the set of even natural numbers, is a subset of $\mathbb{N}$.

5. $\mathbb{O}$, the set of odd natural numbers, is a subset of $\mathbb{N}$.

6. $A = \{x \colon x \in \mathbb{R} \text{ and } x^2 = 4\}$ which is equal to the set $\{-2, 2\}$. This example demonstrates that you may have to do some work to actually produce specific elements of a set defined by a property.

Remark 2.2.4. Russell's paradox. The idea that sets are defined by properties is a natural one, but there are murky logical depths. It seems obvious that given a property $P(x)$, there is a corresponding set $\{x \colon P(x)\}$ of all those things that have that property. We shall now describe a famous result in the history of mathematics called *Russell's Paradox*, named after Bertrand Russell (1872–1970), which shows that just because something is obvious does not make it true. Define $\mathscr{R} = \{x \colon x \notin x\}$. In other words: the set of all sets that do not contain themselves as an element. For example, $\emptyset \in \mathscr{R}$. We now ask the question: is $\mathscr{R} \in \mathscr{R}$? There are only two possible answers and we investigate them both.

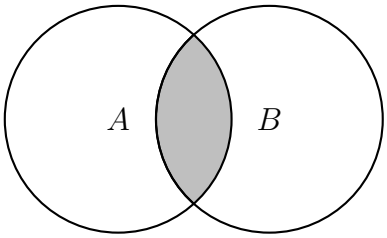
1. Suppose that $\mathscr{R} \in \mathscr{R}$. This means that $\mathscr{R}$ must satisfy the entry requirements to belong to $\mathscr{R}$ which it can only do if $\mathscr{R} \notin \mathscr{R}$.

2. Suppose that $\mathscr{R} \notin \mathscr{R}$. Then it satisfies the entry requirement to belong to $\mathscr{R}$ and so $\mathscr{R} \in \mathscr{R}$.

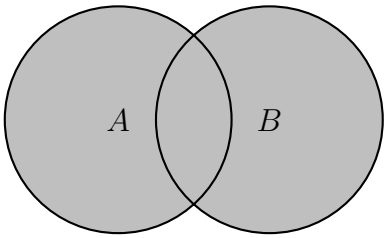
Thus exactly one of $\mathscr{R} \in \mathscr{R}$ and $\mathscr{R} \notin \mathscr{R}$ must be true but assuming one implies the other. We therefore have an honest-to-goodness contradiction. Our only way out is to conclude that, whatever $\mathscr{R}$ might be, it is not a set. This contradicts the obvious statement we began with. If you want to understand how to escape this predicament, you will have to study *set theory*. Disconcerting as this might be, imagine how much more so it was to the mathematician Gottlob Frege (1848–1925). He was working on a book which based the development of mathematics on sets when he received a letter from Russell describing this paradox thereby undermining what Frege was attempting to achieve.

Boolean operations

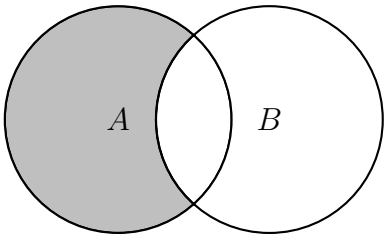
We now define three operations on sets that are based on the PL logical connectives $\wedge$, $\vee$ and $\neg$. They are called *Boolean operations*, named after George Boole (1815–1864). Let A and B be sets. Define a set, called the *intersection* of A and B, denoted by $A \cap B$, whose elements consist of all those elements that belong to A **and** B.



Define a set, called the *union* of A and B, denoted by $A \cup B$, whose elements consist of all those elements that belong to A **or** B.



Define a set, called the *difference* or *relative complement* of A and B, denoted by $A \setminus B$,[1] whose elements consist of all those elements that belong to A **and not** to B.



The diagrams used to illustrate the above definitions are called *Venn diagrams* where a set is represented by a region in the plane.

Example 2.2.5. Let $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6\}$. Determine $A \cap B$, $A \cup B$, $A \setminus B$ and $B \setminus A$.

- $A \cap B$. We have to find the elements that belong to both A and B. We start with the elements in A and work left-to-right: 1 is not an element of B; 2 is not an element of B; 3 and 4 are elements of B. Thus $A \cap B = \{3, 4\}$.

[1] Sometimes denoted by $A - B$.

- $A \cup B$. We join the two sets together $\{1, 2, 3, 4, 3, 4, 5, 6\}$ and then read from left-to-right weeding out repetitions to get $A \cup B = \{1, 2, 3, 4, 5, 6\}$.
- $A \setminus B$. We have to find the elements of A that do not belong to B. Read the elements of A from left-to-right comparing them with the elements of B: 1 does not belong to B; 2 does not belong to B: but 3 and 4 do belong to B. It follows that $A \setminus B = \{1, 2\}$.
- To calculate $B \setminus A$ we have to find the set of elements of B that do not belong to A. This set is equal to $\{5, 6\}$.

Sets A and B are said to be *disjoint* if $A \cap B = \emptyset$.

Properties of Boolean operations

In the theorem below, we list the properties the Boolean operations have.

Theorem 2.2.6 (Properties of Boolean operations)**.** *Let A, B and C be any sets.*

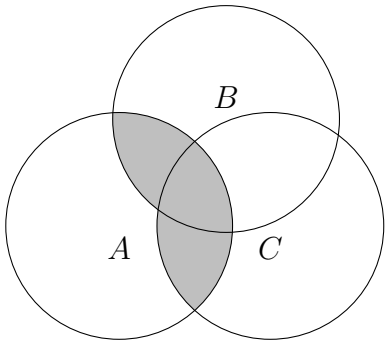
1. $A \cap (B \cap C) = (A \cap B) \cap C$. *Intersection is associative.*
2. $A \cap B = B \cap A$. *Intersection is commutative.*
3. $A \cap \emptyset = \emptyset = \emptyset \cap A$. *The empty set is the zero for intersection.*
4. $A \cup (B \cup C) = (A \cup B) \cup C$. *Union is associative.*
5. $A \cup B = B \cup A$. *Union is commutative.*
6. $A \cup \emptyset = A = \emptyset \cup A$. *The empty set is the identity for union.*
7. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. *Intersection distributes over union.*
8. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. *Union distributes over intersection.*
9. $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$. *De Morgan's law part one.*
10. $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$. *De Morgan's law part two.*
11. $A \cap A = A$. *Intersection is idempotent.*
12. $A \cup A = A$. *Union is idempotent.*

To *illustrate* these properties, we can use Venn diagrams.

Example 2.2.7. We illustrate property (7). The Venn diagram for

$$(A \cap B) \cup (A \cap C)$$

is given below.



This is exactly the same as the Venn diagram for

$$A \cap (B \cup C).$$

It follows that

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

at least as far as Venn diagrams are concerned.

To *prove* these properties hold, we have to proceed more formally and use PL.

Example 2.2.8. We prove part (7) of Theorem 2.2.6 to illustrate the method. We use the fact that

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r).$$

Our goal is to prove that

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

To do this, we have to prove that the set of elements belonging to the lefthand side is the same as the set of elements belonging to the righthand side. An element x either belongs to A or it does not. Similarly, it either belongs to

B or it does not, and it either belongs to C or it does not. Define p to be the statement '$x \in A$'. Define q to be the statement '$x \in B$'. Define r to be the statement '$x \in C$'. If p is true then x is an element of A, and if p is false then x is not an element of A. Using now the definitions of the Boolean operations, it follows that $x \in A \cap (B \cup C)$ precisely when the statement $p \wedge (q \vee r)$ is true. Similarly, $x \in (A \cap B) \cup (A \cap C)$ precisely when the statement $(p \wedge q) \vee (p \wedge r)$ is true. But these two statements have the same truth tables. It follows that an element belongs to the lefthand side precisely when it belongs to the righthand side. Consequently, the two sets are equal.

The fact that $A \cap (B \cap C) = (A \cap B) \cap C$ means that we can just write $A \cap B \cap C$ unambiguously without brackets. Similarly, we can write $A \cup B \cup C$ unambiguously. This can be extended to any number of unions and any number of intersections.

The Boolean algebra of subsets of a set

The set whose elements are all the subsets of X is called the *power set* of X and is denoted by $\mathsf{P}(X)$. It is important to remember that the power set of a set X contains both $\emptyset$ and X as elements.

Example 2.2.9. We find all the subsets of the set $X = \{a, b, c\}$ and so the power set of X. First there is the subset with no elements, the empty set. Then there are the subsets that contain exactly one element: $\{a\}, \{b\}, \{c\}$. Then the subsets containing exactly two elements: $\{a, b\}, \{a, c\}, \{b, c\}$. Finally, there is the whole set X. It follows that X has 8 subsets and so

$$\mathsf{P}(X) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, X\}.$$

Proposition 2.2.10. *Let X be a finite set with n elements. Then $|\mathsf{P}(X)| = 2^n$.*

Let $A \subseteq X$. Define $\overline{A} = X \setminus A$.

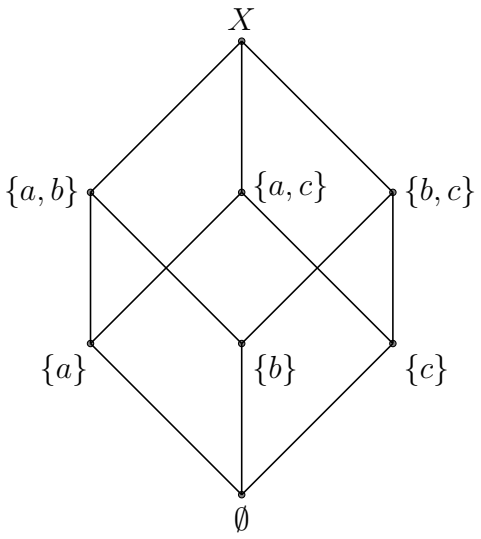
BA	Power sets
$+$	$\cup$
$\cdot$	$\cap$
$^-$	$^-$
1	X
0	$\emptyset$

The proof of the following is immediate by Theorem 2.2.6.

Proposition 2.2.11. *Let X be any set. Then $(\mathsf{P}(X), \cup, \cap, ^-, \emptyset, X)$ is a Boolean algebra.*

Example 2.2.12. We draw a graph of selected inclusions between the ele-

ments of the Boolean algebra $\mathsf{P}(X)$ where $X = \{a, b, c\}$.



The following is simply included for context.

Theorem 2.2.13.

1. *Any finite Boolean algebra has 2^n elements for some natural number n.*
2. *Any two finite Boolean algebras with the same number of elements are 'essentially the same' or* isomorphic.

We may now use Venn diagrams to illustrate certain Boolean expressions and so help us in simplify them.

Example 2.2.14. We return to the Boolean expression

$$x + yz + \bar{x}y + x\bar{y}z$$

that we simplified earlier. We interpret this in set theory where the Boolean variables x, y, z are replaced by sets X, Y, Z. Observe that $\bar{x}y$ translates into $Y \setminus X$. Thus the above Boolean expression is the set

$$X \cup (Y \cap Z) \cup (Y \setminus X) \cup ((X \cap Z) \setminus Y).$$

If you draw the Venn diagram of this set you get exactly $X \cup Y$. This translates into the Boolean expression $x + y$ which is what we obtained when we simplified the Boolean expression.

2.3 Binary arithmetic

In everyday life, we write numbers down using base 10. For computers, it is more natural to treat numbers as being in base 2 or *binary*. In this section, we briefly explain what this means.

A *string* is simply an ordered sequence of symbols. Strings are another important class of data structures. If the symbols used in the string are taken only from $\{0, 1\}$, the set of *binary digits*, then we have a *binary string*. All kinds of data can be represented by binary strings.

The key idea is that every natural number can be represented as a sum of powers of two. How to do this is illustrated by the following example. Recall that $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, ...

Example 2.3.1. Write 316 as a sum of powers of 2.

- We first find the highest power of 2 that is less than or equal to our number. We see that $2^8 < 316$ but $2^9 > 316$. We can therefore write $316 = 2^8 + 60$.
- We now repeat this procedure with 60. We find that $2^5 < 60$ but $2^6 > 60$. We can therefore write $60 = 2^5 + 28$.
- We now repeat this procedure with 28. We find that $2^4 < 28$ but $2^5 > 28$. We can therefore write $28 = 2^4 + 12$.
- We now repeat this procedure with 12. We find that $2^3 < 12$ but $2^4 > 12$. We can therefore write $12 = 2^8 + 4$. Of course $4 = 2^2$.

It follows that

$$316 = 2^8 + 2^5 + 2^4 + 2^3 + 2^2,$$

a sum of powers of two.

Once we have written a number as a sum of powers of two we can encode that information as a binary string. How to do so is illustrated by the following example that continues the one above.

Example 2.3.2. We have that

$$316 = 2^8 + 2^5 + 2^4 + 2^3 + 2^2.$$

We now set up the following table

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	1	1	1	0	0

which includes all powers of two up to and including the largest one used. We say that the binary string

100111100

is the *binary representation* of the number 316.

Given a number written in binary it is a simple matter to convert it back into standard base ten. Essentially the table above should be constructed and the sums of powers of two that occur should be added up.

All of arithmetic can be carried out working solely in base 2. However, we shall only need to describe how to do addition. The starting point is to consider how to add two one-bit binary numbers together.

		carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

However, this is not quite enough to enable us to add two arbitrary binary numbers together. The basic algorithm is the same as in base 10 except that you carry 2 rather than carry 10. Because of the carries you actually need to know how to add three binary digits rather than just two. The table below shows how.

			carry	sum
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

Example 2.3.3. We calculate $11 + 1101$ in binary using the second of the two tables above. We first pad out the first string with 0s to get 0011. We

now write the two binary numbers one above the other.

0	0	1	1
1	1	0	1
			0

Now work from right-to-left a column at a time adding up the digits you see and making any necessary carries. Observe that in the first column on the right there is no carry but it is more helpful, as we shall see, to think of this as a 0 carry. Here is the sequence of calculations.

	0	0	1	1
	1	1	0	1
				0
			1	0

	0	0	1	1
	1	1	0	1
			0	**0**
		1	1	0

	0	0	1	1
	1	1	0	1
		0	**0**	**0**
	1	1	1	0

	0	0	1	1
	1	1	0	1
1	**0**	**0**	**0**	**0**
1	1	1	1	0

We find that the sum of these two numbers is 10000.

2.4 Circuit design

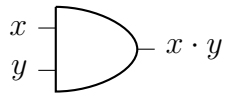
We now have enough theory to explain how Boolean algebras can be used in designing circuits. We shall reuse some results and ideas from PL interpreted in terms of Boolean algebras.

A computer circuit is a physical, not a mathematical, object but can be modelled by mathematics. Currently, all computers are constructed using binary logic, meaning that their circuits operate using two values of some physical property, such as voltage. Circuits come in two types: *sequential circuits* which have an internal memory and *combinatorial circuits* which do not. We shall only describe combinatorial circuits. Let C be a combinatorial circuit with m input wires and n output wires. We can think of C as consisting of n combinatorial circuits $C_1, \ldots, C_n$ each having m input wires but only one output wire each. The combinatorial circuit C_i tells us about how the ith output of the circuit C behaves with respect to inputs. Thus it is enough to describe combinatorial circuits with m inputs and 1 output. Such a circuit is said to describe a *Boolean function* $f\colon \mathbb{B}^m \to \mathbb{B}$ where $\mathbb{B}^m$ represents all the possible 2^m inputs. Such a function is described by means

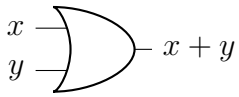
of an *input/output table*. Here is an example of such a table.

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

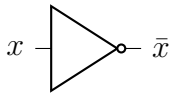
Our goal is to show that any such Boolean function can be constructed from certain simpler Boolean functions called *gates*. There are a number of different kinds of gates but we begin with the three basic ones. The *and-gate* is the function $\mathbb{B}^2 \to \mathbb{B}$ defined by $(x, y) \mapsto x \cdot y$. We use the following symbol to represent this function.



The *or-gate* is the function $\mathbb{B}^2 \to \mathbb{B}$ defined by $(x, y) \mapsto x + y$. We use the following symbol to represent this function.

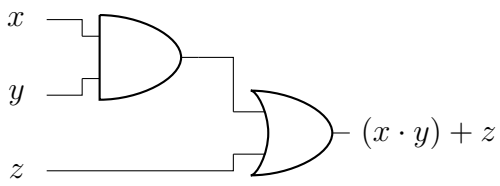


Finally, the *not-gate* is the function $\mathbb{B} \to \mathbb{B}$ defined by $x \mapsto \bar{x}$. We use the following symbol to represent this function.

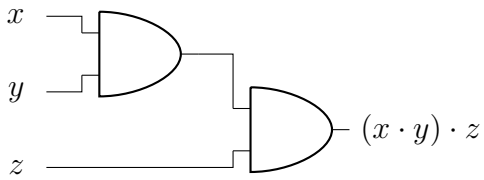


Diagrams constructed using gates are called *circuits* and show how Boolean functions can be computed as we shall see. Such mathematical circuits can be converted into physical circuits with gates being constructed from simpler circuit elements called transistors which operate like electronic switches. We shall show how in the next section.

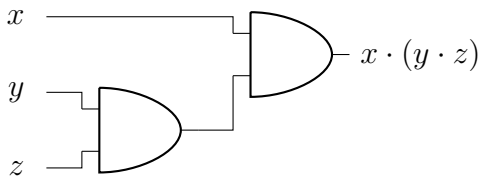
Example 2.4.1. Here is a simple circuit.



Example 2.4.2. Because of the associativity of $\cdot$, the circuit



and the circuit



compute the same function. Similar comments apply to the operation $+$.

The main theorem in circuit design is the following. It is nothing other than the Boolean algebra version of Theorem 1.6.6, the result that says that every truth function arises as the truth table of a wff.

Theorem 2.4.3 (Fundamental theorem of circuit design)**.** *Every Boolean function $f\colon \mathbb{B}^m \to \mathbb{B}$ can be constructed from and-gates, or-gates and not-gates.*

Proof. Assume that f is described by means of an input/output table. We deal first with the case where f is the constant function to 0. In this case,

$$f(x_1, \ldots, x_m) = (x_1 \cdot \overline{x_1}) \cdot x_2 \cdot \ldots \cdot x_m.$$

Next we deal with the case where the function f takes the value 1 exactly once. Let $\mathbf{a} = (a_1, \ldots, a_m) \in \mathbb{B}^m$ be such that $f(a_1, \ldots, a_m) = 1$. Define $\mathbf{m} = y_1 \cdot \ldots \cdot y_m$, called the *minterm* associated with $\mathbf{a}$, as follows:

$$y_i = \begin{cases} x_i & \text{if } a_i = 1 \\ \overline{x_i} & \text{if } a_i = 0. \end{cases}$$

Then $f(\mathbf{x}) = y_1 \cdot \ldots \cdot y_m$. Finally, we deal with the case where the function f is none of the above. Let the inputs where f takes the value 1 be $\mathbf{a}_1, \ldots, \mathbf{a}_r$, respectively. Construct the corresponding minterms $\mathbf{m}_1, \ldots, \mathbf{m}_r$, respectively. Then

$$f(\mathbf{x}) = \mathbf{m}_1 + \ldots + \mathbf{m}_r.$$

□

Example 2.4.4. We illustrate the proof of Theorem 2.4.3 by means of the following input/output table.

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

The three elements of $\mathbb{B}^3$ where f takes the value 1 are $(0,0,1)$, $(0,1,0)$ and $(1,0,0)$. The minterms corresponding to each of these inputs are $\bar{x} \cdot \bar{y} \cdot z$, $\bar{x} \cdot y \cdot \bar{z}$ and $x \cdot \bar{y} \cdot \bar{z}$, respectively. It follows that

$$f(x,y,z) = \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z}.$$

We could if we wished attempt to simplify this Boolean expression. This becomes important when we wish to convert it into a circuit.

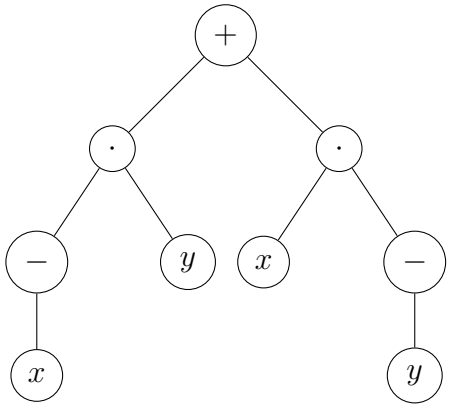
Example 2.4.5. The input/output table below

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

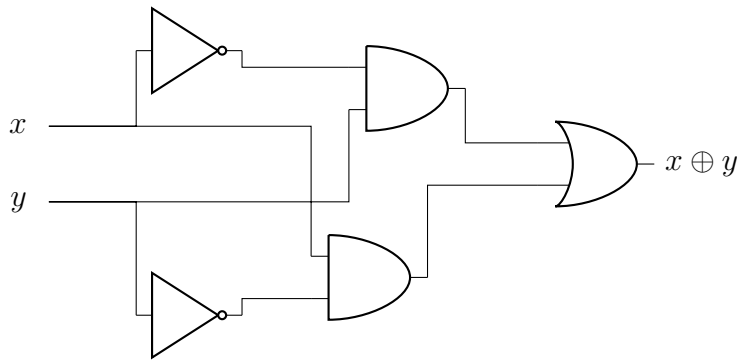
defines *exclusive or* or *xor*. By Theorem 2.4.3, we have that

$$x \oplus y = \bar{x} \cdot y + x \cdot \bar{y}.$$

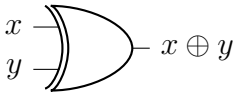
We may describe this by means of a parse tree just as we did in the case of wff.



This parse tree may be converted into a circuit in a series of stages but requires two further circuit elements. First, we require two input wires: one labelled x and one labelled y. But in the parse tree x occurs twice and y occurs twice. We therefore need a new circuit element called *fanout*. This has one input and then branches with each branch carrying a copy of the input. In this case, we need one fanout with input x and two outward branches and another with input y and two outward branches. In addition, we need to allow wires to cross but not to otherwise interact. This is called *interchange* and forms the second additional circuit element we need. Finally, we replace the Boolean symbols by the corresponding gates and rotate the diagram ninety degrees clockwise so that the inputs come in from the left and the output emerges from the right. We therefore obtain the following circuit diagram.



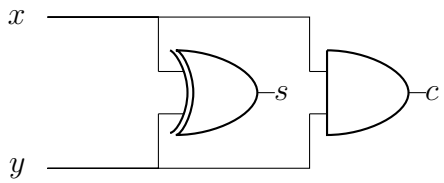
For our subsequent examples, it is convenient to abbreviate the circuit for xor by means of a single circuit symbol called an *xor-gate*.



Example 2.4.6. Our next circuit is known as a *half-adder* which has two inputs and two outputs and is defined by the following input/output table.

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

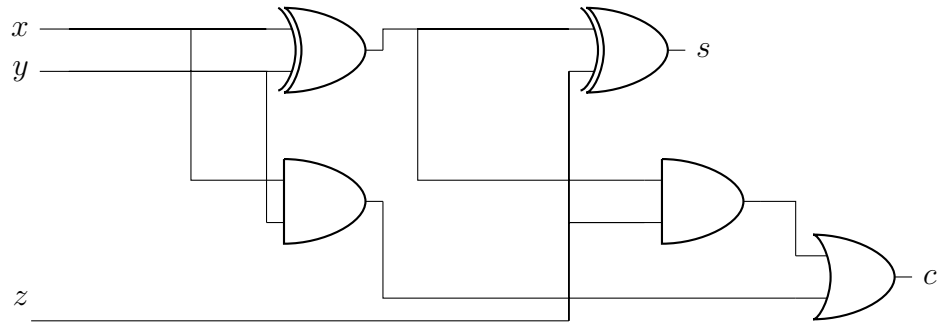
This treats the input x and y as numbers in binary and then outputs their sum. Observe that $s = x \oplus y$ and $c = x \cdot y$. Thus using the previous example we may construct a circuit that implements this function.



Example 2.4.7. Our final circuit is known as a *full-adder* which has three inputs and two outputs and is defined by the following input/output table.

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This treats the three inputs as numbers in binary and adds them together. The following circuit realizes this behaviour using two half-adders completed with an or-gate.



Example 2.4.8. Full-adders are the building blocks from which all arithmetic computer circuits can be built. Specifically, suppose that we want to add two four bit binary numbers together where we pad the numbers out by adding 0 at the front if necessary. Denote them by $m = a_3a_2a_1a_0$ and $n = b_3b_2b_1b_0$. The sum $m + n$ in base 2 is computed in a similar way to calculating a sum in base 10. Thus first calculate $a_0 + b_0$, write down the sum bit, and pass any carry to be added to $a_1 + b_1$ and so on. Although $a_0 + b_0$ can be computed by a half-adder subsequent additions may require the addition of three bits because of the presence of a carry bit. For this reason, we actually use four full-adders joined in series with the rightmost full-adder having one of its inputs set to 0.

2.5 Transistors

Because of De Morgan's laws, every Boolean expression is equal to one in which only $\cdot$ and $^-$ appear. We shall prove that and-gates and not-gates can be constructed from transistors and so we will have proved that every combinatorial circuit can be constructed from transistors. We start by recalling the input/output table of a transistor. I shall regard the transistor as a new Boolean operation that I shall write as $x \,\square\, y$. This is certainly not standard notation (there is none) but we only need this notation in this section.

x	y	$x \,\square\, y$
0	0	0
1	0	1
0	1	0
1	1	0

It is important to observe that $x \,\square\, y \neq y \,\square\, x$. Observe that $x \,\square\, y = x \cdot \bar{y}$. We now carry out a couple of calculations.

1. $1 \square y = 1\bar{y} = \bar{y}$. Thus by fixing $x = 1$ we can negate y.
2. Observe that $x \cdot y = x \cdot \bar{\bar{y}}$. Thus

$$x \cdot y = x \square \bar{y} = x \square (1 \square y).$$

We have therefore proved the following.

Theorem 2.5.1 (The fundamental theorem of transistors)**.** *Every combinatorial circuit can be constructed from transistors.*

Exercises 7

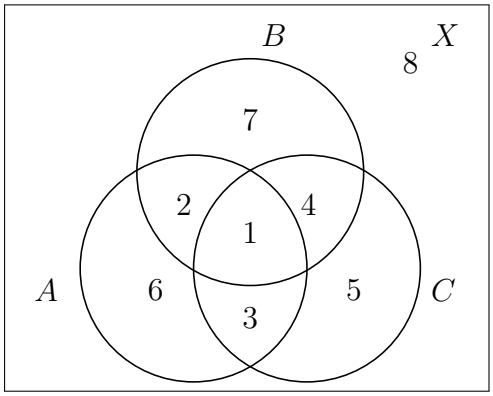
These cover all of Chapter 2

---

1. Let $A = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$, $B = \{\spadesuit, \diamondsuit, \clubsuit, \heartsuit\}$ and $C = \{\spadesuit, \diamondsuit, \clubsuit, \heartsuit, \clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$. Is it true or false that $A = B$ and $B = C$? Explain.
2. Find all subsets of the set $\{a, b, c, d\}$.
3. Let $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Write down the following subsets of X:
   (a) The subset A of even elements of X.
   (b) The subset B of odd elements of X.
   (c) $C = \{x\colon\ x \in X \text{ and } x \geq 6\}$.
   (d) $D = \{x\colon\ x \in X \text{ and } x > 10\}$.
   (e) $E = \{x\colon\ x \in X \text{ and } x \text{ is prime}\}$.
   (f) $F = \{x\colon\ x \in X \text{ and } (x \leq 4 \text{ or } x \geq 7)\}$.
4. Write down how many elements each of the following sets contains.
   (a) $\emptyset$.
   (b) $\{\emptyset\}$.
   (c) $\{\emptyset, \{\emptyset\}\}$.

(d) $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$.

5. This question concerns the following diagram where $A, B, C \subseteq X$. Use Boolean operators to describe each of the eight regions. For example, region (1) is $A \cap B \cap C$.



6. Draw Venn diagrams for each of the following sets.

   (a) $A \cup (B \cap C) \cup (\overline{A} \cap B) \cup (A \cap \overline{B} \cap C)$.

   (b) Show that

   $$(\overline{A} \cap \overline{B} \cap C) \cup (\overline{A} \cap B \cap C) \cup (A \cap \overline{B})$$

   and

   $$(A \cap \overline{B}) \cup (\overline{A} \cap C)$$

   are equal.

   (c) Show that

   $$(A \cap B \cap C) \cup (\overline{A} \cap B \cap C) \cup \overline{B} \cup \overline{C}$$

   and X are equal.

7. (a) Prove that

   $$a \vee (b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge \neg b \wedge c)$$

   and

   $$a \vee b$$

   are logically equivalent.

(b) Prove that

$$(\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c) \vee (a \wedge \neg b)$$

and

$$(a \wedge \neg b) \vee (\neg a \wedge c)$$

are logically equivalent.

(c) Prove that

$$(a \wedge b \wedge c) \vee (\neg a \wedge b \wedge c) \vee \neg b \vee \neg c$$

is a tautology.

8. What is the connection between Questions 6 and 7 above?

9. Convert the following numbers into binary.

   (a) 10.

   (b) 42.

   (c) 153.

   (d) 2001.

10. Convert the following binary numbers into decimal.

    (a) 111.

    (b) 1010101.

    (c) 111000111.

11. carry out the following additions in binary.

    (a) $11 + 11$.

    (b) $10110011 + 1100111$.

    (c) $11111 + 11011$.

    The following questions are generally trickier and you may have to play around with the algebra quite a bit before getting the answer out. That's normal.

12. Prove that $b + b = b$ for all $b \in B$ in a Boolean algebra.

13. Prove that $0b = 0$ for all $b \in B$ in a Boolean algebra.

14. Prove that $1 + b = 1$ for all $b \in B$ in a Boolean algebra.

15. Prove the following absorption laws in any Boolean algebra.

    (a) $a + ab = a$.

    (b) $a(a + b) = a$.

    (c) $a + \bar{a}b = a + b$.

    (d) $a(\bar{a} + b) = ab$.

16. The aim of this question is to prove that De Morgan's laws follow from the Boolean algebra axioms and do not need to be assumed.

    (a) Prove that if $a + b = 1$ and $ab = 0$ then $b = \bar{a}$.

    (b) Prove that $\bar{\bar{a}} = a$.

    (c) Prove De Morgan's laws.

        i. $\overline{(a + b)} = \bar{a}\bar{b}$.

        ii. $\overline{ab} = \bar{a} + \bar{b}$.

17. Simplify each of the following Boolean algebra expressions as much as possible. You might find it useful to draw Venn diagrams first.

    (a) $x + \bar{x}y$.

    (b) $x\bar{y}\bar{x} + xy\bar{x}$

    (c) $\bar{x}\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z$.

    (d) $xy + x\bar{y} + \bar{x}y$.

    (e) $x + yz + \bar{x}y + \bar{y}xz$.

18. Let $\mathbb{B} = \{0, 1\}$ be a two-element Boolean algebra. Prove that $+$ must behave like $\vee$, that $\cdot$ must behave like $\wedge$ and that complementation must behave like negation. [Hint: this question is not asking you to show that $\{F, T\}$ is a Boolean algebra.]

19. Define a new logical connective $\star$ by $p \star q = p \wedge \neg q$. How is this related to the transistor? Does it form an adequate set of connectives on its own?

Chapter 3

First-order logic

> *7. What we cannot speak about we must pass over in silence.* —
> Tractatus Logico-Philosophicus, Ludwig Wittgenstein.

PL is useful, as we have seen, but also very limited. The goal of this final part of the course is to add features to PL that will make it more powerful and more useful although there will be a price to pay in that the resulting system will be intrinsically harder to work with. We shall study what is known as *first-order logic* (FOL) and sometimes *predicate logic*. Essentially

$$\text{FOL} = \text{PL} + \text{predicates} + \text{quantifiers}.$$

This logic is the basis of applications of logic to CS, such as PROLOG. For mathematics, we have the following

$$\boxed{\text{Mathematics} = \text{Set theory} + \text{FOL}}$$

3.1 Splitting the atom: names, predicates and relations

Recall that a statement is a sentence which is capable of being either true or false. In PL, statements can only be analysed further in terms of the usual PL connectives until we get to atoms. The atoms cannot then be further

analysed. We shall show how, in fact, atoms can be split by using some new ideas.

A *name* is a word that picks out a specific individual. For example, 2, π, Darth Vader are all names. Once something has been named, we can refer to it by that name.

At its simplest, a sentence can be analysed into a *subject* and a *predicate*. For example, the sentence 'grass is green' has 'grass' as its subject and 'is green' as its predicate. To make the nature of the predicate clearer, we might write '— is green' to indicate the slot into which a name could be fitted. Or, more formally, we could use a *variable* and write 'x is green'. We could symbolize this predicate thus '$G(x) =$ 'x is green'. We may replace the variable x by names to get honest statements that may or may not be true. Thus G(grass) is the statement 'grass is green' whereas G(cheese) is the statement 'cheese is green'. This is an example of a *1-place predicate* because there is exactly one slot into which a name can be placed to yield a statement. Other examples of 1-place predicates are: 'x is a prime' and 'x likes honey'.

There are also *2-place predicates*. For example,

$$P(x, y) = \text{ '}x \text{ is the father of } y\text{'}$$

is such a predicate because there are two slots that can be replaced by names to yield statements. Thus (spoiler alert) P(Darth Vader, Luke Skywalker) is true but P(Winnie-the-Pooh, Darth Vader) is false.

More generally, $P(x_1, \ldots, x_n)$ denotes an *n-place predicate* or an *n-ary* predicate. We say that the predicate has *arity* n. Here $x_1, \ldots, x_n$ are n variables that mark the n positions into which names can be slotted.

Most of the predicates we shall meet will have arities 1 or 2. But in theory there is no limit. Thus $F(x_1, x_2, x_3)$ is the 3-place predicate 'x_1 fights x_2 with a x_3'. By inserting names, we get the statement F(Luke Skywalker, Darth Vader, banana) (deleted scene).

In FOL, names are called *constants* and are usually denoted by the rather more mundane $a, b, c, \ldots$ or $a_1, a_2, a_3 \ldots$. *Variables* are denoted by $x, y, z \ldots$ or $x_1, x_2, x_3, \ldots$. They have no fixed meaning but serve as place-holders into which names can be slotted.

An *atomic formula* is a predicate whose slots are filled with either variables or constants.

We now turn to the question of what predicates do. 1-place predicates

describe sets[1]. Thus the 1-place predicate $P(x)$ describes the set

$$P = \{a\colon P(a) \text{ is true } \}$$

although this is usually written simply as

$$P = \{a\colon P(a)\}.$$

For example, if $P(x)$ is the 1-place predicate 'x is a prime' then the set it describes is the set of prime numbers. To deal with what 2-place predicates describe we need some new notation. An *ordered pair* is written (a, b) where a is the *first component* and b is the *second component*. Observe that we use round brackets. As the name suggests: order matters and so $(a, b) \neq (b, a)$, unlike in set notation. Let $P(x, y)$ be a 2-place predicate. Then it describes the set

$$P = \{(a, b)\colon P(a, b) \text{ is true } \}$$

which is usually just written

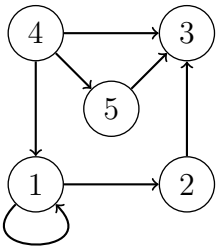
$$P = \{(a, b)\colon P(a, b)\}.$$

A set of ordered pairs where the elements are taken from some set X is called a *binary relation* on the set X. Thus 2-place predicates describe binary relations. We often denote binary relations by Greek letters. There is a nice graphical way to represent binary relations at least when the set they are defined on is not too big. Let ρ be a binary relation defined on the set X. We draw a *directed graph* or *digraph* of ρ. This consists of *vertices* labelled by the elements of X and *arrows* where an arrow is drawn from a to b precisely when $(a, b) \in \rho$.

Example 3.1.1. The binary relation

$$\rho = \{(1, 1), (1, 2), (2, 3), (4, 1), (4, 3), (4, 5), (5, 3)\}$$

is defined on the set $X = \{1, 2, 3, 4, 5\}$. Its corresponding directed graph is



[1]At least informally. There is the problem of Russell's paradox. How that can be dealt with, if indeed it can be, is left to a more advanced course.

Example 3.1.2. Binary relations are very common in mathematics. Here are some examples.

1. The relation $x \mid y$ is defined on the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ if x exactly divides y.

2. The relations $\leq$ (less than or equal to) and $<$ (strictly less than) are defined on $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

3. The relation $\subseteq$ (is a subset of) is defined between sets.

4. the relation $\in$ (is an element of) is defined between sets.

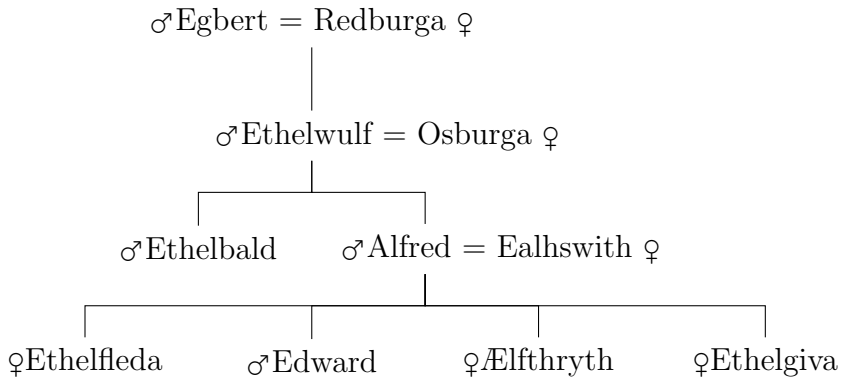
5. The relation $\equiv$ (logical equivalence) is defined on the set of wff.

For convenience, I shall only use 1-place predicates and 2-place predicates (and so also only sets and binary relations), but, in principle, there is no limit on the arities of predicates and we can study *ordered n-tuples* just as well as ordered pairs.

3.2 Structures

We begin with two examples.

Example 3.2.1. Here is part of the family tree of some Anglo-Saxon kings and queens.



We shall analyse the mathematics behind this family tree. First, we have a set D of kings and queens called the domain. This consists of eleven Tolkienesque elements

$$D = \left\{ \begin{array}{ccccccc} \text{Egbert}, & \text{Redburga}, & \text{Ethelwulf}, & \text{Osburga}, & \text{Ethelbald}, & \text{Alfred}, \\ \text{Eahlswith}, & \text{Ethelfleda}, & \text{Edward}, & \text{Ælfthryth}, & \text{Ethelgiva} & & \end{array} \right\}.$$

But we have additional information. Beside each name is a symbl ♂ or ♀ which means that that person is respectively male or female. This is just a way of defining two subsets of D:

$$M = \{\text{Egbert}, \text{Ethelwulf}, \text{Ethelbald}, \text{Alfred}, \text{Edward}\}$$

and

$$F = \{\text{Redburga}, \text{Osburga}, \text{Eahlswith}, \text{Ethelfleda}, \text{Ælfthryth}, \text{Ethelgiva}\}.$$

There are also two other peices of information. The most obvious are the lines linking one generation to the next. This is the binary relation defined by the 2-place predicate 'x is the parent of y'. It is the set of ordered pairs π. For example,

$$(\text{Ethelwulf}, \text{Alfred}), (\text{Osburga}, \text{Alfred}) \in \pi.$$

A little less obvious is the notation $=$ which stands for the binary relation defined by the 2-place predicate 'x is married to y'. It is the set of ordered pairs μ. For example,

$$(\text{Egbert}, \text{Redburga}), (\text{Redburga}, \text{Egbert}), (\text{Ethelwulf}, \text{Osburga}) \in \mu.$$

It follows that the information contained in the family tree is also contained in the following package

$$(D, M, F, \pi, \mu).$$

Example 3.2.2. This looks quite different at first from the previous example but is mathematically very closely related to it. Define

$$(\mathbb{N}, \mathbb{E}, \mathbb{O}, \leq, |)$$

where $\mathbb{E}$ and $\mathbb{O}$ are, respectively, the sets of odd and even natural numbers and $\leq$ and $|$ are binary relations.

We define a *structure* to consist of a non-empty set D, called the *domain*, together with a finite selection of subsets, binary relations, etc.

FOL is a language that will enable us to talk about structures.

3.3 Quantification: $\forall$, $\exists$

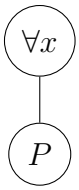
We begin with an example. Let $A(x)$ be the 1-place predicate 'x is made of atoms'. I want to say that 'everything is made from atoms'. In PL, I have no choice but to use infinitely many conjunctions

$$A(\text{jelly}) \wedge A(\text{ice-cream}) \wedge A(\text{blancmange}) \wedge \ldots$$

This is inconvenient. We get around this by using what is called the *universal quantifier* $\forall$. We write

$$(\forall x)A(x)$$

which should be read 'for all x, x is made from atoms' or 'for each x, x is made from atoms'. The variable does not have to be x, we have all of these $(\forall y)$, $(\forall z)$ and so on. As we shall see, you should think of $(\forall x)$ and its ilk as being a new unary connective. Thus



is the parse tree for $(\forall x)P$.

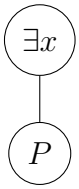
The corresponding infinite disjunction

$$P(a) \vee P(b) \vee P(c) \vee \ldots$$

is true when at least one of the terms of true. There is a corresponding *existential quantifier* $\exists$. We write

$$(\exists x)A(x)$$

which should be read 'there exists an x, such that $P(x)$ is true' or 'there is at least one x, such that $P(x)$ is true'. The variable does not have to be x, we have all of these $(\exists y)$, $(\exists z)$ and so on. You should also think of $(\exists x)$ and its ilk as being a new unary connective. Thus



is the parse tree for $(\exists x)P$.

3.4 Syntax

A *first-order language* consists of a choice of predicate letters with given arities. Which ones you choose depends on what problems you are interested in. In addition to our choice of predicate letters, we also have, for free, our logical symboms carried forward from PL: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus$; we also have variables $x, y, \ldots, x_1, x_2, \ldots$; constants (names) $a, b \ldots a_1, a_2, \ldots$; and quantifiers $(\forall x)$, $(\forall y), \ldots (\forall x_1), (\forall x_2), \ldots$. Recall that an *atomic formula* is a predicate letter with variables or constants inserted in all the available slots. We can now define a *formula* or *wff* in FOL.

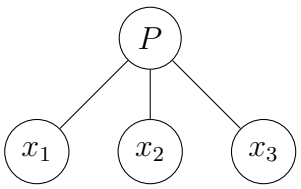
1. All atomic formulae are wff.

2. If A and B are wff so too are $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$, $(A \oplus B)$, and $(\forall x)A$, for any variable x and $(\exists x)A$ for any variable x.

3. All wff arise by repeated application of steps (1) or (2) a finite number of times.

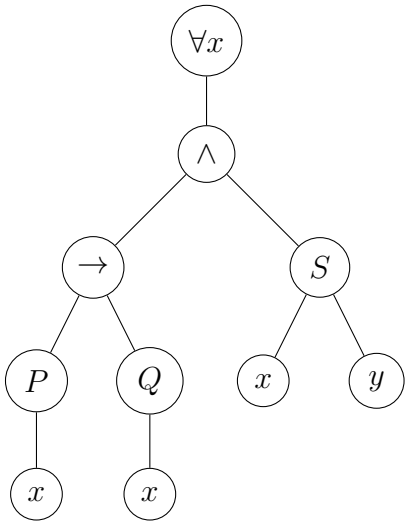
I should add that I will carry forward to FOL from PL the same conventions concerning the use of brackets without further comment.

We may adapt parse trees to FOL. An expression such as $P(x_1, x_2, x_3)$, for example, gives rise to the parse tree



The quantifiers $(\forall x)$ and $(\exists x)$ are treated as unary operators as we have seen. The leaves of the parse tree are either variables or constants.

Example 3.4.1. The formula $(\forall x)[(P(x) \rightarrow Q(x)) \wedge S(x, y)]$ has the parse tree



Sentences

We now come to a fundamental, and quite difficult, definition. Choose any vertex v in a tree. The part of the tree, including v itself, that lies below v is clearly also a tree. It is called the *subtree determined by* v. In a parse tree, the subtree determined by an occurrence of $(\forall x)$ is called the *scope* of this occurrence of the quantifier. Similarly for $(\exists x)$. An occurrence of a variable x is called *bound* if it occurs within the scope of an occurrence of either $(\forall x)$ and $(\exists x)$. Otherwise, this occurrence is called *free*. Observe that the occurrence of x in both $(\forall x)$ and $(\exists x)$ is always bound.

Example 3.4.2. Consider the formula $(P(x) \to Q(y)) \to (\exists y)R(x, y)$. The scope of $(\exists y)$ is $R(x, y)$. Thus the x occurring in $R(x, y)$ is free and the y occurring in $R(x, y)$ is bound.

A formula that has a free occurrence of some variable is called *open* otherwise it is called *closed*. A closed wff is called a *sentence*. FOL is about sentences.

Example 3.4.3. Consider the wff

$$(\exists x)(F(x) \wedge G(x)) \to ((\exists x)F(x) \wedge (\exists x)G(x)).$$

I claim this is a sentence. There are three occurrences of $(\exists x)$. The first occurrence binds the x in $F(x) \wedge G(x)$. The second occurrence binds the

occurrence of x in the second occurrence of $F(x)$. The third occurrence binds the occurrence of x in the second occurrence of $G(x)$. It follows that every occurrence of x in this wff is bound and there are no other variables. Thus the wff is closed as claimed.

I shall explain why sentences are the natural things to study once I have defined the semantics of FOL.

3.5 Semantics

Let L be a first-order language. For concreteness, suppose that it consists of two predicate letters where P is a 1-place predicate letter and Q is a 2-place predicate letter. An *interpretation* I of L is any structure (D, A, ρ) where D, the domain, is a non-empty set, $A \subseteq D$ is a subset and ρ is a binary relation on D. We interpret P as A and Q as ρ. Thus the wff $(\exists x)(P(x) \wedge Q(x,x))$ *is interpreted as* $(\exists x \in D)((x \in A) \wedge ((x,x) \in \rho))$. Under this interpretation, every sentence S in the language makes an assertion about the elements of D using A and ρ. We say that I *is a model of* S, written $I \vDash S$, if S is true when interpreted in this way in I. A sentence S is said to be *universally (or logically) valid*, written $\vDash S$, if it is true in **all** interpretations.

> *Whereas in PL we studied tautologies, in FOL we study logically valid formulae.*

We write $S_1 \equiv S_2$ to mean that the sentences S_1 and S_2 are true in exactly the same interpretations. It is not hard to prove that this is equivalent to showing that $\vDash S_1 \leftrightarrow S_2$.

The following example should help to clarify these definitions. Specifically, why it is sentences that we are interested in.

Example 3.5.1. Interpret the 2-place predicate symbol $P(x,y)$ as the binary relation 'x is the father of y' where the domain is the set of people. Observe that the phrase 'x is the father of y' is neither true nor false since we know nothing about x and y. Consider now the wff $(\exists y)P(x,y)$. This says 'x is a father'. This is still neither true nor false since x is not specified. Finally, consider the wff $S_1 = (\forall x)(\exists y)P(x,y)$. This says 'everyone is a father'. Observe that it is a sentence and that it is also a statement. In this case, it is false. I should add that in reading a sequence of quantifiers work your way in from left to right. The new sentence $S_2 = (\exists x)(\forall y)P(x,y)$ is a different

statement. It says that there is someone who is the father of everyone. This is also false. The new sentence $S_3 = (\forall y)(\exists x)P(x, y)$ says that 'everyone has a father' which is true.

We now choose a new interpretation. This time we interpret $P(x, y)$ as '$x \leq y$' and the domain as the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. The sentence S_1 says that for each natural number there is a natural number at least as big. This is true. The sentence S_2 says that there exists a natural number that is less than or equal to any natural number. This is true because 0 has exactly that property. Finally, sentence S_3 says that for each natural number there is a natural number that is no bigger which is true.

As we see in the above examples, sentences say something, or rather can be interpreted as saying something. Thus a sentence interpreted in some structure is a statement about that structure and is therefore true or false in that interpretation.

I should add that sentences also arise (and this is a consequence of the definition but perhaps not obvious), when constants, that is names, are substituted for variables. Thus 'Darth Vader is the father of Winnie-the-Pooh' is a sentence, that happens to be false (and I believe has never been uttered by anyone before).

3.6 De Morgan's laws for $\forall$ and $\exists$

The universal quantifier $\forall$ is a sort of infinite version of $\wedge$ and the existential quantifier is a sort of infinite version of $\vee$. The following result is therefore not surprising.

Theorem 3.6.1 (De Morgan for quantifiers)**.**

1. $\neg(\forall x)A \equiv (\exists x)\neg A$.
2. $\neg(\exists x)A \equiv (\forall x)\neg A$.

3.7 Truth trees for FOL

We begin with a motivating example.

Example 3.7.1. Consider the following famous argument.

1. All man are mortal.

2. Socrates is a man.

3. Therefore socrates is mortal.

Here (1) and (2) are the assumptions and (3) is the conclusion. If you agree to the truth of (1) and (2) then you are obliged to accept the truth of (3). This cannot be verified using PL but we can use what we have introduced so far to analyse this argument and prove that it is valid. We introduce some predicate symbols. We interpret $M(x)$ to be 'x is mortal', and $H(x)$ to be 'x is a man'. Our argument above has the following form.

1. $(\forall x)(H(x) \to M(x))$.

2. H(Socrates).

3. Therefore M(Socrates).

We prove that this argument is valid. If (1) is true, then it is true for every named individual a and so $H(a) \to M(a)$. Thus for the particular individual Socrates we have that H(Socrates) $\to M$(Socrates). But we are told in (2) that H(Socrates) is true. We are now in the world of PL and we have that

$$H(\text{Socrates}) \to M(\text{Socrates}), H(\text{Socrates}) \vDash M(\text{Socrates}).$$

Thus M(Socrates) is true.

We now generalize the key idea used in the above argument. Given a sentence $(\forall x)A(x)$, where $A(x)$ here means some wff containing x, then if we replace all free occurrences of x in $A(x)$ by a constant a, we have *instantiated the universal quantifier at a*. There is a similar procedure for existential quantification.

Example 3.7.2. If we instantiate the wff $(\exists y)[(\exists x)P(x) \to P(y)]$ at y by the constant a we get the wff $(\exists x)P(x) \to P(a)$.

The leading idea in what follows is this: *convert FOL sentences into PL wff by means of instantiation.*

Truth tree rules for FOL

- All PL truth tree rules are carried forward.
- De Morgan's rules for quantifiers

$$\begin{array}{c} \neg(\forall x)A \\ | \\ (\exists x)\neg A \end{array} \qquad \begin{array}{c} \neg(\exists x)A \\ | \\ (\forall x)\neg A \end{array}$$

- New name rule.

$$\begin{array}{c} (1)\ (\exists x)A(x)\checkmark \\ | \\ A(a) \end{array}$$

  where we add $A(a)$ at the bottom of all branches containing (1) and where a is a constant that does not already appear in the branch containing (1).

- Never ending rule.

$$\begin{array}{c} (2)\ (\forall x)A(x)\ * \\ | \\ A(a) \end{array}$$

  where we add $A(a)$ at the bottom of a branch containing (2) and a is any constant appearing in the branch containing (2) or a is a new constant if no constants have yet been introduced. [The rationale for the latter is that all domains are non-empty]. We have used the $*$ to mean that the wff is never used up.

We use truth trees for FOL is the same way as in PL.

- To prove that $\vDash X$ we show that some truth tree with root $\neg X$ closes.
- To prove that $X_1, \ldots, X_n \vDash X$ we show that some truth tree with root $X_1, \ldots, X_n, \neg X$ closes.
- To prove that $X \equiv Y$ prove that $\vDash X \leftrightarrow Y$.

We shall describe how these rules are applied by means of examples.

Examples 3.7.3.

1. Show that the following is a valid argument

$$(\forall x)(H(x) \to M(x)), H(a) \vDash M(a).$$

Here is the truth tree.

$$\begin{array}{c} (\forall x)(H(x) \to M(x)) * \\ H(a) \\ \neg M(a) \\ | \\ H(a) \to M(a) \checkmark \\ \swarrow \quad \searrow \\ \text{✗}\neg H(a) \qquad M(a) \text{ ✗} \end{array}$$

2. Show that the following argument is valid

$$(\exists x)(\forall y)F(x,y) \vDash (\forall y)(\exists x)F(x,y).$$

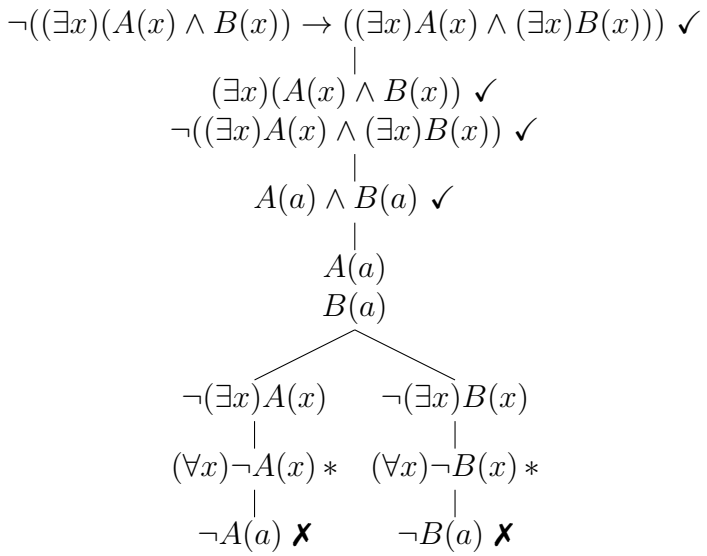
Here is the truth tree.

$$\begin{array}{c} (\exists x)(\forall y)F(x,y) \checkmark \\ \neg(\forall y)(\exists x)F(x,y)\checkmark \\ | \\ (\exists y)\neg(\exists x)F(x,y)\checkmark \\ | \\ (\exists y)(\forall x)\neg F(x,y)\checkmark \\ | \\ (\forall y)F(a,y) * \\ | \\ (\forall x)\neg F(x,b) * \\ | \\ F(a,b) \\ | \\ \neg F(a,b) \text{ ✗} \end{array}$$

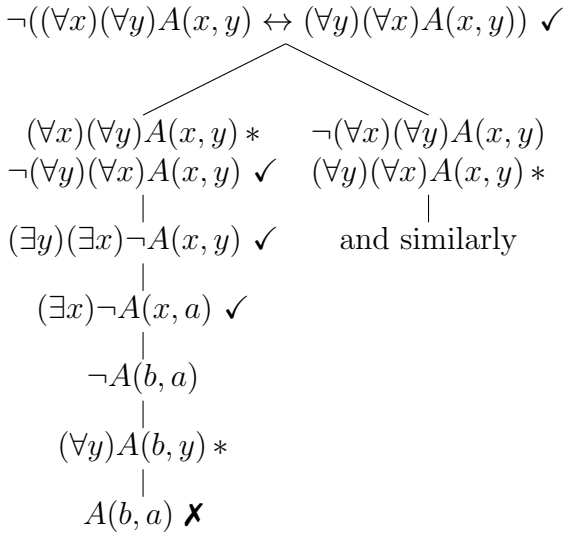
3. Prove that

$$\vDash (\exists x)(A(x) \wedge B(x)) \to ((\exists x)A(x) \wedge (\exists x)B(x)).$$

Here is the truth tree.



4. Prove that $(\forall x)(\forall y)A(x,y) \equiv (\forall y)(\forall x)A(x,y)$. Here is the truth tree.



There are, however, some major differences between PL and FOL when it comes to the behaviour of truth trees.

Examples 3.7.4.

1. Truth trees can have infinite branches. Here is an example.

$$
\begin{array}{c}
(\forall x)(\exists y)R(x,y)\ * \\
| \\
(\exists y)R(a_1,y)\ \checkmark \\
| \\
R(a_1,a_2) \\
| \\
(\exists y)R(a_2,y)\ \checkmark \\
| \\
R(a_2,a_3) \\
| \\
\text{and so on}
\end{array}
$$

2. There are sentences that have no finite models. See Question 11 of Exercises 9.

3. The order in which the rules for truth trees are applied in FOL does matter. If we place at the root the wff $(\forall x)(\exists y)P(x,y)$ and $P(a)\wedge\neg P(a)$ then we can get an infinite tree if we repeatedly apply the tree rules to the first wff but an immediate contradiction, and so a closed finite truth tree, if we start with the second wff instead.

3.8 The *Entscheidungsproblem*

Let's start with PL. If I give you a wff, you can decide, using a truth table for example, in a finite amount of time whether that wff is a tautology or not. The decision you make does not require any intelligence; it uses an algorithm. We say that the problem of deciding whether a wff in PL is a tautology is decidable.

Let's turn now to FOL. The analogous question of whether a formula in FOL is universally valid or not is usually known by its German form, the *Entscheidungsproblem*, because it was formulated by the great German mathematician David Hilbert in 1928. Unlike the case of PL, there is no algorithm for deciding this question. This was proved independently in 1936 by Alonzo Church in the US and by Alan Turing in the UK. Turing's resolution to this question was far-reaching since it involved him in formulating, mathematically, what we would now call a computer.

Exercises 8

The following were adapted from Chapter 1 of Volume II of [15]. There are some liberties taken in the solutions. Read [15] for more background.

In the following exercises, use this transcription guide:

a: Athelstan (name)

e: Ethelgiva (name)

c: Cenric (name)

$M(x)$: x is marmalade — the colour, not what you put on your toast (1-place predicate)

$C(x)$: x is a cat (1-place predicate)

$L(x, y)$: x likes y (2-place predicate)

$T(x, y)$: x is taller than y (2-place predicate)

1. Transcribe the following formulae into English.

   (a) $\neg L(a, a)$.
   (b) $L(a, a) \rightarrow \neg T(a, a)$.
   (c) $\neg(M(c) \vee L(c, e))$.
   (d) $C(a) \leftrightarrow (M(a) \vee L(a, e))$.
   (e) $(\exists x)T(x, c)$.
   (f) $(\forall x)L(a, x) \wedge (\forall x)L(c, x)$.
   (g) $(\forall x)(L(a, x) \wedge L(c, x))$.
   (h) $(\exists x)T(x, a) \vee (\exists x)T(x, c)$.
   (i) $(\exists x)(T(x, a) \vee T(x, c))$.
   (j) $(\forall x)(C(x) \rightarrow L(x, e))$.
   (k) $(\exists x)(C(x) \wedge \neg L(e, x))$.
   (l) $\neg(\forall x)(C(x) \rightarrow L(e, x))$.
   (m) $(\forall x)[C(x) \rightarrow (L(c, x) \vee L(e, x))]$.

(n) $(\exists x)[C(x) \wedge (M(x) \wedge T(x, c))]$.

2. For each formula in Question 1, draw its parse tree.

3. Transcribe the following English sentences into formulae.

   (a) Everyone likes Ethelgiva.

   (b) Everyone is liked by either Cenric or Athelstan.

   (c) Either everyone is liked by Athelstan or everyone is liked by Cenric.

   (d) Someone is taller than both Athelstan and Cenric.

   (e) Someone is taller than Athelstan and someone is taller than Cenric.

   (f) Ethelgiva likes all cats.

   (g) All cats like Ethelgiva.

   (h) Ethelgiva likes some cats.

   (i) Ethelgiva likes no cats.

   (j) Anyone who likes Ethelgiva is not a cat.

   (k) No one who likes Ethelgiva is a cat.

   (l) Somebody who likes Athelstan likes Cenric.

   (m) No one likes both Athelstan and Cenric.

Exercises 9

---

1. This question is about interpreting sequences of quantifiers. Remember that $\forall x$ can be read both as *for all* x as well as *for each* x. Consider the following 8 sentences involving the 2-place predicate P.

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

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

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

(d) $(\exists x)(\exists y)P(x,y)$.

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

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

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

(h) $(\exists y)(\exists x)P(x,y)$.

Consider two interpretations. The first has domain $\mathbb{N}$, the set of natural numbers, and P is interpreted as $\leq$. The second has domain 'all people', and P is interpreted by the binary relation 'is the father of'. Write down what each of the sentences means in each interpretation and state whether the interpretation is a model of the sentence or not. You may use the fact that (a)≡(e) and (d)≡(h).

2. Prove that
$$(\forall x)R(x) \models (\exists x)R(x).$$
Explain informally why it is a valid argument.

3. Prove that
$$(\forall x)(\forall y)R(x,y) \models (\forall x)R(x,x).$$

4. Prove that
$$(\exists x)(\exists y)F(x,y) \equiv (\exists y)(\exists x)F(x,y).$$

5. Prove that
$$(\forall x)(P(x) \wedge Q(x)) \equiv ((\forall x)P(x) \wedge (\forall x)Q(x)).$$

6. Prove that
$$(\exists x)(P(x) \vee Q(x)) \equiv ((\exists x)P(x) \vee (\exists x)Q(x)).$$

7. Prove that
$$((\forall x)P(x) \vee (\forall x)Q(x)) \rightarrow (\forall x)(P(x) \vee Q(x))$$
is logically valid. On the other hand, show that the following is not logically valid by constructing a counterexample
$$(\forall x)(P(x) \vee Q(x)) \rightarrow ((\forall x)P(x) \vee (\forall x)Q(x)).$$

8. Prove that
$$(\exists x)[D(x) \to (\forall y)D(y)]$$
is universally valid. Interpret $D(x)$ as 'x drinks' with domain people. What does the above wff say in this interpretation? Does this seem plausible? Resolve the issue.

9. For each of the following formulae draw a parse tree and demonstrate that the formula is closed and therefore a sentence. Then show that each sentence is logically valid.

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

   (b) $(\exists x)P(x) \to (\exists y)P(y)$.

   (c) $(\forall y)((\forall x)P(x) \to P(y))$.

   (d) $(\exists y)(P(y) \to (\forall x)P(x))$.

   (e) $\neg(\exists y)P(y) \to [(\forall y)((\exists x)P(x) \to P(y))]$.

10. This question marks the beginning of using our logic in mathematical proof. Let R be a 2-place predicate symbol. We say that an interpretation of R has the respective property (in italics) if it is a model of the corresponding sentence.

    - *Reflexive:* $(\forall x)R(x,x)$.
    - *Irreflexive:* $(\forall x)\neg R(x,x)$.
    - *Symmetric:* $(\forall x)(\forall y)(R(x,y) \to R(y,x))$.
    - *Asymmetric:* $(\forall x)(\forall y)(R(x,y) \to \neg R(y,x))$.
    - *Transitive:* $(\forall x)(\forall y)(\forall z)(R(x,y) \wedge R(y,z) \to R(x,z))$.

    Illustrate these definitions by using directed graphs.

    Prove the following using truth trees.

    (a) If R is asymmetric then it is irreflexive.

    (b) If R is transitive and irreflexive then it is asymmetric.

11. Put $S = F_1 \wedge F_2 \wedge F_3$ where

    - $F_1 = (\forall x)(\exists y)R(x,y)$.

- $F_2 = (\forall x)\neg R(x, x)$.
- $F_3 = (\forall x)(\forall y)(\forall z)[R(x, y) \wedge R(y, z) \rightarrow R(x, z)]$.

Prove first that the following is a model of S: the domain is $\mathbb{N}$ and $R(x, y)$ is interpreted as $x < y$. Next prove that S has *no finite models*. That is, no models in which the domain is finite.

Chapter 4

2015 Exam paper and solutions

4.1 Exam paper

Each question is worth 20 marks

1. (a) Construct truth-tables for each of the following wff.

   i. $p \wedge q$.

   ii. $p \vee q$.

   iii. $p \rightarrow q$.

   iv. $p \leftrightarrow q$.

   [1 mark each]

   (b) Construct truth-tables and parse trees for each of the following wff.

   i. $\neg p \vee q$.

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

   iii. $\neg(p \vee q)$.

   iv. $\neg(p \wedge q)$.

   [1 mark each]

   (c) Construct the truth-table of $(p \leftrightarrow q) \wedge (p \rightarrow \neg r)$. [4 marks]

   (d) Construct a wff in disjunctive normal form that has the following

truth-table. [4 marks]

p	q	r	A
T	T	T	F
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

(e) Prove using truth-tables that $p \vee (q \wedge r)$ is logically equivalent to $(p \vee q) \wedge (p \vee r)$. [4 marks]

2. (a) Define the binary connective $p \downarrow q = \neg(p \vee q)$. Show that $p \to q$ is logically equivalent to a wff in which the only binary connective that appears is $\downarrow$. [4 marks]

(b) Show that the wff $(x \wedge \neg y) \vee (\neg x \wedge y)$ is logically equivalent to a wff in CNF (conjunctive normal form). [4 marks]

(c) Show that $p \to q, \neg p \to q \vDash q$ is a valid argument. [4 marks]

(d) Use **truth trees** to determine whether the following is a tautology

$$[(p \to q) \wedge (q \to r)] \to [(p \vee q) \to r].$$

[4 marks]

(e) Use **truth trees** to determine whether the following is a valid argument

$$p \vee q, p \to r, q \to s \vDash r \vee s.$$

[4 marks]

3. In this question, you should use the Boolean algebra axioms listed at the end of this exam paper. You should also assume that $a^2 = a$ and $a + a = a$ for all elements a of a Boolean algebra.

(a) Prove $a0 = 0$. [2 marks]

(b) Prove $a + 1 = 1$. [2 marks]

(c) Prove $a + ab = a$. [2 marks]

(d) Prove $a + \bar{a}b = a + b$. [2 marks]

(e) Draw a Venn diagram to illustrate the following Boolean expression.

$$\bar{x}\,\bar{y}\,\bar{z} + \bar{x}y\bar{z} + x\bar{y}\,\bar{z} + xy\bar{z}.$$

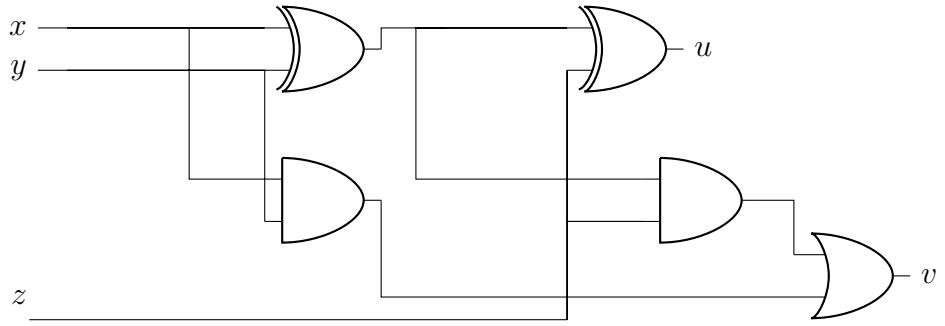
[2 marks]

(f) Simplify the Boolean expression

$$\bar{x}\,\bar{y}\,\bar{z} + \bar{x}y\bar{z} + x\bar{y}\,\bar{z} + xy\bar{z}$$

as much as possible using properties of Boolean algebras. Any properties used should be clearly stated. [4 marks]

(g) The following diagram shows a circuit with three inputs and two outputs. The symbols are recalled at the end of the exam paper.



Draw up an input/output table for this circuit and describe what it is doing. [6 marks]

4. (a) Let A and B be 1-place predicate symbols. Construct a structure in which $(\exists x)A(x) \wedge (\exists x)B(x)$ is true but $(\exists x)(A(x) \wedge B(x))$ is false. [10 marks]

(b) Prove using truth trees that

$$(\exists x)(A(x) \wedge B(x)) \rightarrow [(\exists x)A(x) \wedge (\exists x)B(x)]$$

is universally valid. [10 marks]

Boolean algebra axioms

(B1) $(x + y) + z = x + (y + z)$.

(B2) $x + y = y + x$.

(B3) $x + 0 = x$.

(B4) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

(B5) $x \cdot y = y \cdot x$.

(B6) $x \cdot 1 = x$.

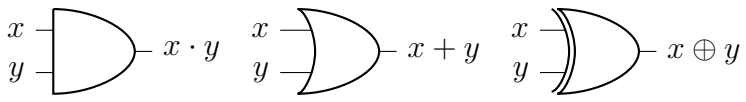
(B7) $x \cdot (y + z) = x \cdot y + x \cdot z$.

(B8) $x + (y \cdot z) = (x + y) \cdot (x + z)$.

(B9) $x + \bar{x} = 1$.

(B10) $x \cdot \bar{x} = 0$.

Circuit symbols



4.2 Solutions

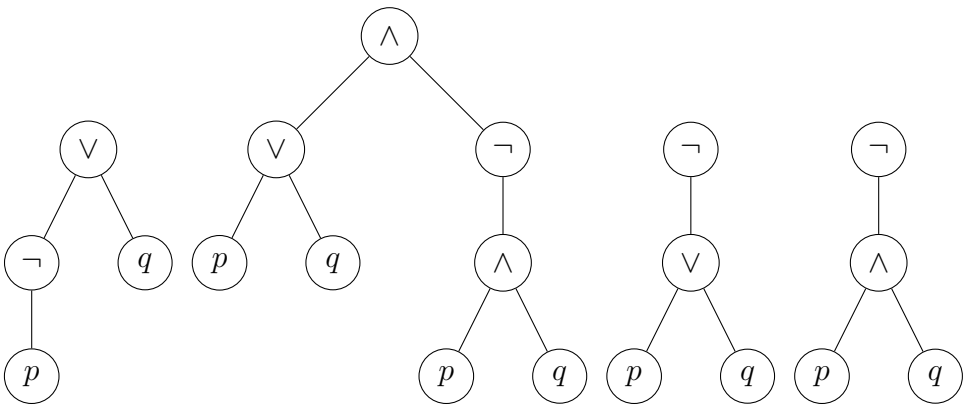
1. (a) For each correct truth table [1 mark].

p	q	(i) $p \wedge q$	(ii) $p \vee q$	(iii) $p \rightarrow q$	(iv) $p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

(b) For each correct truth table [$\frac{1}{2}$ mark].

p	q	(i)	(ii)	(iii)	(iv)
T	T	T	F	F	F
T	F	F	T	F	T
F	T	T	T	F	T
F	F	T	F	T	T

For each correct parse tree [$\frac{1}{2}$ mark].



(c) Correct truth table [4 marks].

p	q	r	$(p \leftrightarrow q) \wedge (p \rightarrow \neg r)$
T	T	T	F
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	T
F	F	F	T

(d) $(p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge \neg q \wedge r)$ [4 marks].

(e) Construct truth tables for $p \vee (q \wedge r)$ and $(p \vee q) \wedge (p \vee r)$ separately and show that they are equal.

p	q	r	$p \vee (q \wedge r)$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

2. (a) Now

$$p \rightarrow q \equiv \neg p \vee q \equiv \neg(\neg(\neg p \vee q)) \equiv \neg(\neg p \downarrow q).$$

But $\neg x \equiv x \downarrow x$. Thus

$$p \rightarrow q \equiv ((p \downarrow p) \downarrow q) \downarrow ((p \downarrow p) \downarrow q).$$

[4 marks]

(b) Use the distributivity law to get

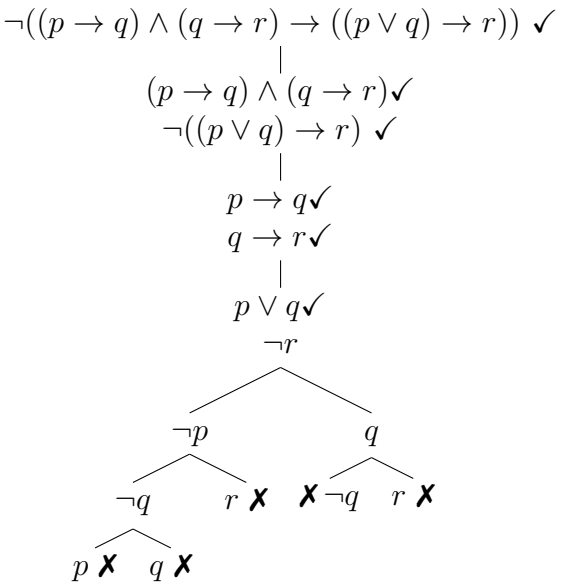
$$(x \wedge \neg y) \vee (\neg x \wedge y) \equiv (x \vee \neg x) \wedge (\neg y \vee \neg x) \wedge (x \vee y) \wedge (\neg y \vee y).$$

This can be simplified further though this is not required. [4 marks]

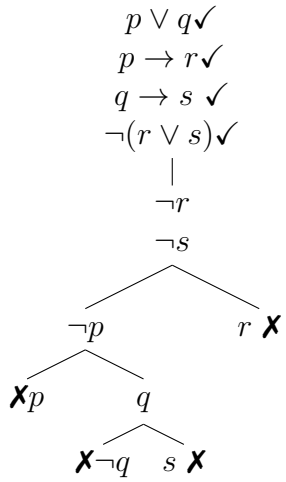
(c) Any correct method allowed (including truth trees). For example, show that

$$\vDash ((p \to q) \wedge (\neg p \to q)) \to q.$$

(d) All the branches in the following truth tree close and so the wff is a tautology [4 marks].



(e) All the branches in the following truth tree close and so the argument is valid [4 marks].



3. (a)

$$\begin{aligned} a \cdot 0 &= a \cdot (a \cdot \bar{a}) \text{ by (B10)} \\ &= (a \cdot a) \cdot \bar{a} \text{ by (B4)} \\ &= a \cdot \bar{a} \text{ since } a^2 = a \\ &= 0 \text{ by (B10).} \end{aligned}$$

(b) This is the dual proof to (a).

$$\begin{aligned} a + 1 &= a + (a + \bar{a}) \text{ by (B9)} \\ &= (a + a) + \bar{a} \text{ by (B1)} \\ &= a + \bar{a} \text{ since } a + a = a \\ &= 1 \text{ by (B9).} \end{aligned}$$

(c)

$$\begin{aligned} a + a \cdot b &= a \cdot 1 + a \cdot b \text{ by (B6)} \\ &= a \cdot (1 + b) \text{ by (B7)} \\ &= a \cdot 1 \text{ by (b) above} \\ &= a \text{ by (B6).} \end{aligned}$$

(d)

$$\begin{aligned} a + b &= a + 1b \text{ by (B6)} \\ &= a + (a + \bar{a})b \text{ by (B9)} \\ &= a + ab + \bar{a}b \text{ by (B7) and (B5)} \\ &= a + \bar{a}b \text{ by (c) above} \end{aligned}$$

(e) Label three circles X, Y and Z, intersecting and enclosed in a rectangle. Then the Venn diagram is the complement of Z.

(f) By (e), we expect $\bar{z}$. Here is a proof.

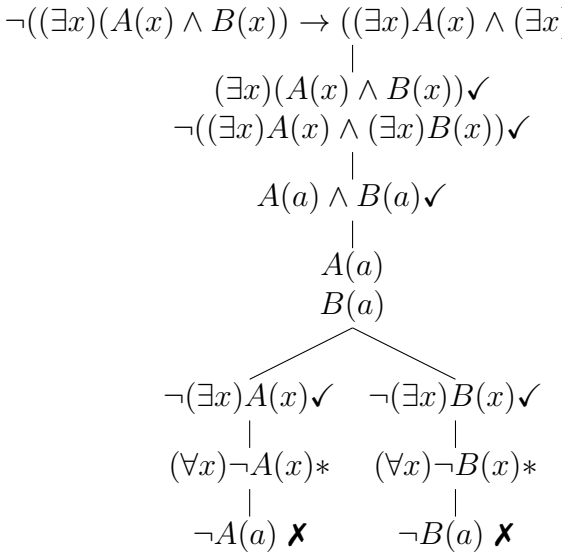
$$\begin{aligned} \bar{x}\,\bar{y}\,\bar{z} + \bar{x}y\bar{z} + x\bar{y}\,\bar{z} + xy\bar{z} &= (\bar{x}\,\bar{y} + \bar{x}y + x\bar{y} + xy)\bar{z} \text{ by (B7)} \\ &= (\bar{x}(\bar{y} + y) + x(\bar{y} + y))\bar{z} \text{ by (B7)} \\ &= (\bar{x}1 + x1)\bar{z} \text{ by (B9)} \\ &= (\bar{x} + x)\bar{z} \text{ by (B6)} \\ &= 1\bar{z} \text{ by (B9)} \\ &= \bar{z} \text{ by (B6).} \end{aligned}$$

(g) This is a full-adder.

x	y	z	u	v
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

4. (a) There are many correct answers. Here is one. Define the domain to be $\mathbb{N}$. Interpret $A(x)$ as 'x is even' and $B(x)$ as 'x is odd'. Now in this interpretation, $(\exists x)A(x) \wedge (\exists x)B(x)$ is true since 0 is even and 1 is odd. However, $(\exists x)(A(x) \wedge B(x))$ is false since there is no number that is both odd and even. [10 marks]

(b) The truth tree below closes and so the wff is universally valid.



Bibliography

[1] S. Aaronson, *Quantum computing since Democritus*, CUP, 2013.

[2] A. Doxiadis, C. H. Papadimitriou, *Logicomix*, Bloomsbury, 2009.

[3] M. R. Garey, D. S. Johnson, *Computers and intractability. A guide to the theory of NP-completeness*, W. H. Freeman and Company, 1997.

[4] R. Hammack, *Book of proof*, VCU Mathematics Textbook Series, 2009. This book can be downloaded for free from `http://www.people.vcu.edu/~rhammack/BookOfProof/index.html`.

[5] D. Harel, *Computers Ltd. What they really can't do*, OUP, 2012.

[6] D. R. Hofstadter, *Gödel, Escher, Bach: an eternal golden braid*, Basic Books, 1999.

[7] G. Labrecque, Truth tree generator, `http://gablem.com/logic`.

[8] S. Lipschutz, M. Lipson, *Discrete mathematics*, second edition, McGraw-Hill, 1997.

[9] A. A. Milne, *Winnie-the-Pooh*, Methuen, 2000.

[10] C. Petzold, *Code: the hidden language of computer hardware and software*, Microsoft Press, 2000.

[11] C. Petzold, *The annotated Turing*, John Wiley & Sons, 2008.

[12] D. Scott et al, *Notes on the formalization of logic: Parts I and II*, Oxford, July, 1981.

[13] R. M. Smullyan, *First-order logic*, Dover, 1995.

[14] R. R. Stoll, *Set theory and logic*, W. H. Freeman and Company, 1961.

[15] P. Teller, *A modern formal logic primer*, Prentice Hall, 1989. This book can be downloaded for free from `http://www.ma.hw.ac.uk/~markl/teaching/LOGIC/Teller.html`.

[16] L. E. Turner, Truth table generator, `http://turner.faculty.swau.edu/mathematics/materialslibrary/truth/`.

[17] M. Zegarelli, *Logic for dummies*, Wiley Publishing, 2007.