1
Introduction

In this chapter we introduce logical reasoning and the idea of mechanizing it, touching briefly on important historical developments. We lay the groundwork for what follows by discussing some of the most fundamental ideas in logic as well as illustrating how symbolic methods can be implemented on a computer.

1.1 What is logical reasoning?

There are many reasons for believing that something is true. It may seem obvious or at least immediately plausible, we may have been told it by our parents, or it may be strikingly consistent with the outcome of relevant scientific experiments. Though often reliable, such methods of judgement are not infallible, having been used, respectively, to persuade people that the Earth is flat, that Santa Claus exists, and that atoms cannot be subdivided into smaller particles.

What distinguishes *logical* reasoning is that it attempts to avoid any unjustified assumptions and confine itself to inferences that are infallible and beyond reasonable dispute. To avoid making any unwarranted assumptions, logical reasoning cannot rely on any special properties of the objects or concepts being reasoned about. This means that logical reasoning must abstract away from all such special features and be equally valid when applied in other domains. Arguments are accepted as logical based on their conformance to a general *form* rather than because of the specific *content* they treat. For instance, compare this traditional example:

All men are mortal
Socrates is a man
Therefore Socrates is mortal

with the following reasoning drawn from mathematics:

All positive integers are the sum of four integer squares
15 is a positive integer
Therefore 15 is the sum of four integer squares

These two arguments are both correct, and both share a common pattern:

All X are Y
a is X
Therefore a is Y

This pattern of inference is logically valid, since its validity does not depend on the content: the meanings of ‘positive integer’, ‘mortal’ etc. are irrelevant. We can substitute anything we like for these X, Y and a, provided we respect grammatical categories, and the statement is still valid. By contrast, consider the following reasoning:

All Athenians are Greek
Socrates is an Athenian
Therefore Socrates is mortal

Even though the conclusion is perfectly true, this is not logically valid, because it does depend on the content of the terms involved. Other arguments with the same superficial form may well be false, e.g.

All Athenians are Greek
Socrates is an Athenian
Therefore Socrates is beardless

The first argument can, however, be turned into a logically valid one by making explicit a hidden assumption ‘all Greeks are mortal’. Now the argument is an instance of the general logically valid form:

All G are M
All A are G
s is A
Therefore s is M

At first sight, this forensic analysis of reasoning may not seem very impressive. Logically valid reasoning never tells us anything fundamentally new about the world – as Wittgenstein (1922) says, ‘I know nothing about the weather when I know that it is either raining or not raining’. In other words, if we *do* learn something new about the world from a chain of reasoning, it must contain a step that is *not* purely logical. Russell, quoted in Schilpp (1944) says:

Hegel, who deduced from pure logic the whole nature of the world, including the non-existence of asteroids, was only enabled to do so by his logical incompetence.[†]

But logical analysis can bring out clearly the necessary relationships *between* facts about the real world and show just where possibly unwarranted assumptions enter into them. For example, from 'if it has just rained, the ground is wet' it follows logically that 'if the ground is not wet, it has not just rained'. This is an instance of a general principle called *contraposition*: from 'if P then Q' it follows that 'if not Q then not P'. However, passing from 'if P then Q' to 'if Q then P' is *not* valid in general, and we see in this case that we cannot deduce 'if the ground is wet, it has just rained', because it might have become wet through a burst pipe or device for irrigation.

Such examples may be, as Locke (1689) put it, 'trifling', but elementary logical fallacies of this kind are often encountered. More substantially, deductions in mathematics are very far from trifling, but have preoccupied and often defeated some of the greatest intellects in human history. Enormously lengthy and complex chains of logical deduction can lead from simple and apparently indubitable assumptions to sophisticated and unintuitive theorems, as Hobbes memorably discovered (Aubrey 1898):

Being in a Gentleman's Library, Euclid's Elements lay open, and 'twas the 47 *El. libri* 1 [Pythagoras's Theorem]. He read the proposition. *By G—*, sayd he (he would now and then sweare an emphaticall Oath by way of emphasis) *this is impossible!* So he reads the Demonstration of it, which referred him back to such a Proposition; which proposition he read. That referred him back to another, which he also read. *Et sic deinceps* [and so on] that at last he was demonstratively convinced of that trueth. This made him in love with Geometry.

Indeed, Euclid's seminal work *Elements of Geometry* established a particular style of reasoning that, further refined, forms the backbone of present-day mathematics. This style consists in asserting a small number of *axioms*, presumably with mathematical content, and deducing consequences from them using *purely logical reasoning*.[‡] Euclid himself didn't quite achieve a complete separation of logical and non-logical, but his work was finally perfected by Hilbert (1899) and Tarski (1959), who made explicit some assumptions such as 'Pasch's axiom'.

[†] To be fair to Hegel, the word *logic* was often used in a broader sense until quite recently, and what we consider logic would have been called specifically *deductive logic*, as distinct from *inductive logic*, the drawing of conclusions from observed data as in the physical sciences.

[‡] Arguably this approach is foreshadowed in the Socratic method, as reported by Plato. Socrates would win arguments by leading his hapless interlocutors from their views through chains of apparently inevitable consequences. When absurd consequences were derived, the initial position was rendered untenable. For this method to have its uncanny force, there must be no doubt at all over the steps, and no hidden assumptions must be sneaked in.

1.2 Calculemus!

'Reasoning is reckoning'. In the epigraph of this book we quoted Hobbes on the similarity between logical reasoning and numerical calculation. While Hobbes deserves credit for making this better known, the idea wasn't new even in 1651.[†] Indeed the Greek word *logos*, used by Plato and Aristotle to mean reason or logical thought, can also in other contexts mean computation or reckoning. When the works of the ancient Greek philosophers became well known in medieval Europe, *logos* was usually translated into *ratio*, the Latin word for reckoning (hence the English words rational, ratiocination, etc.). Even in current English, one sometimes hears 'I reckon that ... ', where 'reckon' refers to some kind of reasoning rather than literally to computation.

However, the connection between reasoning and reckoning remained little more than a suggestive slogan until the work of Gottfried Wilhelm von Leibniz (1646–1716). Leibniz believed that a system for reasoning by calculation must contain two essential components:

- a universal language (*characteristica universalis*) in which anything can be expressed;
- a calculus of reasoning (*calculus ratiocinator*) for deciding the truth of assertions expressed in the *characteristica.*

Leibniz dreamed of a time when disputants unable to agree would not waste much time in futile argument, but would instead translate their disagreement into the *characteristica* and say to each other '*calculemus*' (let us calculate). He may even have entertained the idea of having a machine do the calculations. By this time various mechanical calculating devices had been designed and constructed, and Leibniz himself in 1671 designed a machine capable of multiplying, remarking:

It is unworthy of excellent men to lose hours like slaves in the labour of calculations which could safely be relegated to anyone else if machines were used.

So Leibniz foresaw the essential components that make automated reasoning possible: a language for expressing ideas precisely, rules of calculation for manipulating ideas in the language, and the mechanization of such calculation. Leibniz's concrete accomplishments in bringing these ideas to fruition were limited, and remained little-known until recently. But though his work had limited direct influence on technical developments, his dream still resonates today.

[†] The Epicurian philosopher Philodemus, writing in the first century B.C., introduced the term *logisticos* (λογιστικός) to describe logic as the science of calculation.

1.3 Symbolism

Leibniz was right to draw attention to the essential first step of developing an appropriate language. But he was far too ambitious in wanting to express all aspects of human thought. Eventual progress came rather by extending the scope of the symbolic notations already used in mathematics. As an example of this notation, we would nowadays write '$x^2 \leq y + z$' rather than 'x multiplied by itself is less than or equal to the sum of y and z'. Over time, more and more of mathematics has come to be expressed in formal symbolic notation, replacing natural language renderings. Several sound reasons can be identified.

First, a well-chosen symbolic form is usually shorter, less cluttered with irrelevancies, and helps to express ideas more briefly and intuitively (at least to cognoscenti). For example Leibniz's own notation for differentiation, $\mathrm{d}y/\mathrm{d}x$, nicely captures the idea of a ratio of small differences, and makes theorems like the chain rule $\mathrm{d}y/\mathrm{d}x = \mathrm{d}y/\mathrm{d}u \cdot \mathrm{d}u/\mathrm{d}x$ look plausible based on the analogy with ordinary algebra.

Second, using a more stylized form of expression can avoid some of the ambiguities of everyday language, and hence communicate meaning with more precision. Doubts over the exact meanings of words are common in many areas, particularly law.[†] Mathematics is not immune from similar basic disagreements over exactly what a theorem says or what its conditions of validity are, and the consensus on such points can change over time (Lakatos 1976; Lakatos 1980).

Finally, and perhaps most importantly, a well-chosen symbolic notation can contribute to making mathematical reasoning itself easier. A simple but outstanding example is the 'positional' representation of numbers, where a number is represented by a sequence of numerals each implicitly multiplied by a certain power of a 'base'. In decimal the base is 10 and we understand the string of digits '179' to mean:

$$179 = 1 \times 10^2 + 7 \times 10^1 + 9 \times 10^0.$$

In binary (currently used by most digital computers) the base is 2 and the same number is represented by the string 10110011:

$$10110011 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

[†] For example 'Since the object of ss 423 and 425 of the Insolvency Act 1986 was to remedy the avoidance of debts, the word 'and' between paragraphs (a) and (b) of s 423(2) must be read conjunctively and not disjunctively.' (Case Summaries, *Independent* newspaper, 27th December 1993.)

These positional systems make it very easy to perform important operations on numbers like comparing, adding and multiplying; by contrast, the system of Roman numerals requires more involved algorithms, though there is evidence that many Romans were adept at such calculations (Maher and Makowski 2001). For example, we are normally taught in school to add decimal numbers digit-by-digit from the right, propagating a carry leftwards by adding one in the next column. Once it becomes second nature to follow the rules, we can, and often do, forget about the underlying meaning of these sequences of numerals. Similarly, we might transform an equation $x-3=5-x$ into $x=3+5-x$ and then to $2x=5+3$ without pausing each time to think about *why* these rules about moving things from one side of the equation to the other are valid. As Whitehead (1919) says, symbolism and formal rules of manipulation:

> [...] have invariably been introduced to make things easy. [...] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [...] Civilisation advances by extending the number of important operations which can be performed without thinking about them.

Indeed, such formal rules can be followed reliably by people who do *not* understand the underlying justification, or by computers. After all, computers are expressly designed to follow formal rules (programs) quickly and reliably. They do so without regard to the underlying justification, and will faithfully follow even erroneous sets of rules (programs with 'bugs').

1.4 Boole's algebra of logic

The word *algebra* is derived from the Arabic 'al-jabr', and was first used in the ninth century by Mohammed al-Khwarizmi (ca. 780–850), whose name lies at the root of the word 'algorithm'. The term 'al-jabr' literally means 'reunion', but al-Khwarizmi used it to describe in particular his method of solving equations by collecting together ('reuniting') like terms, e.g. passing from $x+4=6-x$ to $2x=6-4$ and so to the solution $x=1$.† Over the following centuries, through the European renaissance, algebra continued to mean, essentially, rules of manipulation for solving equations.

During the nineteenth century, algebra in the traditional sense reached its limits. One of the central preoccupations had been the solving of equations of higher and higher degree, but Niels Henrik Abel (1802–1829) proved in

† The first use of the phrase in Europe was nothing to do with mathematics, but rather the appellation 'algebristas' for Spanish barbers, who also set ('reunited') broken bones as a sideline to their main business.

1824 that there is no general way of solving polynomial equations of degree 5 and above using the 'radical' expressions that had worked for lower degrees. Yet at the same time the scope of algebra expanded and it became generalized. Traditionally, variables had stood for real numbers, usually unknown numbers to be determined. However, it soon became standard practice to apply all the usual rules of algebraic manipulation to the 'imaginary' quantity i assuming the formal property $i^2 = -1$. Though this procedure went for a long time without any rigorous justification, it was effective.

Algebraic methods were even applied to objects that were not numbers in the usual sense, such as matrices and Hamilton's 'quaternions', even at the cost of abandoning the usual 'commutative law' of multiplication $xy = yx$. Gradually, it was understood that the underlying interpretation of the symbols could be ignored, provided it was established once and for all that the rules of manipulation used are all valid under that interpretation. The state of affairs was described clear-sightedly by George Boole (1815–1864).

> They who are acquainted with the present state of the theory of Symbolic Algebra, are aware, that the validity of the processes of analysis does not depend upon the interpretation of the symbols which are employed, but solely on their laws of combination. Every system of interpretation which does not affect the truth of the relations supposed, is equally admissible, and it is true that the same process may, under one scheme of interpretation, represent the solution of a question on the properties of numbers, under another, that of a geometrical problem, and under a third, that of a problem of dynamics or optics. (Boole 1847)

Boole went on to observe that nevertheless, by historical or cultural accident, all algebra at the time involved objects that were in some sense quantitative. He introduced instead an algebra whose objects were to be interpreted as 'truth-values' of true or false, and where variables represent *propositions.*[†] By a proposition, we mean an assertion that makes a declaration of fact and so may meaningfully be considered either true or false. For example, '$1 < 2$', 'all men are mortal', 'the moon is made of cheese' and 'there are infinitely many prime numbers p such that $p + 2$ is also prime' are all propositions, and according to our present state of knowledge, the first two are true, the third false and the truth-value of the fourth is unknown (this is the 'twin primes conjecture', a famous open problem in mathematics).

We are familiar with applying to numbers various arithmetic operations like unary 'minus' (negation) and binary 'times' (multiplication) and 'plus' (addition). In an exactly analogous way, we can combine truth-values using

[†] Actually Boole gave two different but related interpretations: an 'algebra of classes' and an 'algebra of propositions'; we'll focus on the latter.

so-called *logical connectives*, such as unary 'not' (logical negation or complement) and binary 'and' (conjunction) and 'or' (disjunction).[†] And we can use letters to stand for arbitrary *propositions* instead of *numbers* when we write down expressions. Boole emphasized the connection with ordinary arithmetic in the precise formulation of his system and in the use of the familiar algebraic notation for many logical constants and connectives:

0	false
1	true
pq	p and q
$p+q$	p or q

On this interpretation, many of the familiar algebraic laws still hold. For example, 'p and q' always has the same truth-value as 'q and p', so we can assume the commutative law $pq = qp$. Similarly, since 0 is false, '0 and p' is false whatever p may be, i.e. $0p = 0$. But the Boolean algebra of propositions satisfies additional laws that have no counterpart in arithmetic, notably the law $p^2 = p$, where p^2 abbreviates pp.

In everyday English, the word 'or' is ambiguous. The complex proposition 'p or q' may be interpreted either inclusively (p or q or both) or exclusively (p or q but not both).[‡] In everyday usage it is often implicit that the two cases are mutually exclusive (e.g. 'I'll do it tomorrow or the day after'). Boole's original system restricted the algebra so that $p + q$ only made sense if $pq = 0$, rather as in ordinary algebra x/y only makes sense if $y \neq 0$. However, following Boole's successor William Stanley Jevons (1835–1882), it became customary to allow use of 'or' without restriction, and interpret it in the inclusive sense. We will always understand 'or' in this now-standard sense, 'p or q' meaning 'p or q *or both*'.

Mechanization

Even before Boole, machines for logical deduction had been developed, notably the 'Stanhope demonstrator' invented by Charles, third Earl of Stanhope (1753–1816). Inspired by this, Jevons (1870) subsequently designed and built his 'logic machine', a piano-like device that could perform certain calculations in Boole's algebra of classes. However, the limits of mechanical

[†] Arguably *disjunction* is something of a misnomer, since the two truth-values need not be disjoint, so some like Quine (1950) prefer *alternation*. And the word 'connective' is a misnomer in the case of unary operations like 'not', since it does not connect two propositions, but merely negates a single one. However, both usages are well-established.

[‡] Latin, on the other hand, has separate phrases 'p vel q' and 'aut p aut q' for the inclusive and exclusive readings, respectively.

engineering and the slow development of logic itself meant that the mechanization of reasoning really started to develop somewhat later, at the start of the modern computer age. We will cover more of the history later in the book in parallel with technical developments. Jevons's original machine can be seen in the Oxford Museum for the History of Science.[†]

Logical form

In Section 1.1 we talked about arguments 'having the same form', but did not define this precisely. Indeed, it's hard to do so for arguments expressed in English and other natural languages, which often fail to make the logical structure of sentences apparent: superficial similarities can disguise fundamental structural differences, and vice versa. For example, the English word 'is' can mean 'has the property of being' ('4 is even'), or it can mean 'is the same as' ('$2 + 2$ is 4'). This example and others like it have often generated philosophical confusion.

Once we have a precise symbolism for logical concepts (such as Boole's algebra of logic) we can simply say that two arguments have the same form if they are both instances of the same formal expression, consistently replacing variables by other propositions. And we can use the formal language to make a mathematically precise definition of logically valid arguments. This is not to imply that the definition of logical form and of purely logical argument is a philosophically trivial question; quite the contrary. But we are content not to solve this problem but to finesse it by adopting a precise mathematical definition, rather as Hertz (1894) evaded the question of what 'force' means in mechanics. After enough concrete experience we will briefly consider (Section 7.8) how our demarcation of the logical arguments corresponds to some traditional philosophical distinctions.

1.5 Syntax and semantics

An unusual feature of logic is the careful separation of symbolic expressions and what they stand for. This point bears emphasizing, because in everyday mathematics we often pass unconsciously to the mathematical objects denoted by the symbols. For example when we read and write '12' we think of it as a number, a member of the set $\mathbb{N}$, not as a sequence of two numeral symbols used to represent that number. However, when we want to make precise our formal manipulations, whether these be adding decimal numbers

[†] See `www.mhs.ox.ac.uk/database/index.htm?fname=brief&invno=18230` for some small pictures.

digit-by-digit or using algebraic laws to rearrange symbolic expressions, we need to maintain the distinction. After all, when deriving equations like $x + y = y + x$, the whole point is that the mathematical objects denoted are the same; we cannot directly talk about such manipulations if we only consider the underlying meaning.

Typically then, we are concerned with (i) some particular set of allowable formal expressions, and (ii) their corresponding meanings. The two are sharply distinguished, but are connected by an *interpretation*, which maps expressions to their meanings:

$$\begin{tikzcd}
\boxed{\text{Expression}} \arrow[r, "\text{Interpretation}"] & \boxed{\text{Meaning}}
\end{tikzcd}$$

The distinction between formal expressions and their meanings is also important in linguistics, and we'll take over some of the jargon from that subject. Two traditional subfields of linguistics are *syntax*, which is concerned with the grammatical formation of sentences, and *semantics*, which is concerned with their meanings. Similarly in logic we often refer to methods as 'syntactic' if 'like algebraic manipulations' they are considered in isolation from meanings, and 'semantic' or 'semantical' if meanings play an important role. The words 'syntax' and 'semantics' are also used in linguistics with more concrete meanings, and these too are adopted in logic.

- The *syntax* of a language is a system of grammar laying out rules about how to produce or recognize grammatical phrases and sentences. For example, we might consider 'I went to the shop' grammatical English but not 'I shop to the went' because the noun and verb are swapped. In logical systems too, we will often have rules telling us how to generate or recognize well-formed expressions, perhaps for example allowing '$x + 1$' but not '$+1\times$'.
- The *semantics* of a particular word, symbol, sign or phrase is simply its meaning. More broadly, the semantics of a language is a systematic way of ascribing such meanings to all the (grammatical) expressions in the language. Translated into linguistic jargon, choosing an interpretation amounts exactly to giving a semantics to the language.

Object language and metalanguage

It may be confusing that we will be describing formal rules for performing logical reasoning, and yet will reason *about* those rules using . . . logic! In this connection, it's useful to keep in mind the distinction between the (formal) logic we are talking about and the (everyday intuitive) logic we are using to reason about it. In order to emphasize the contrast we will sometimes deploy the following linguistic jargon. A *metalanguage* is a language used to talk *about* another distinct *object language*, and likewise a *metalogic* is used to reason about an *object logic*. Thus, we often call the theorems we derive about formal logic and automated reasoning systems *metatheorems* rather than merely *theorems*. This is not (only) to sound more grandiose, but to emphasize the distinction from 'theorems' expressed *inside* those formal systems. Likewise, metalogical reasoning applied to formalized mathematical proofs is often called *metamathematics* (see Section 7.1). By the way, our chosen programming language OCaml is derived from Edinburgh ML, which was expressly designed for writing theorem proving programs (Gordon, Milner and Wadsworth 1979) and whose name stands for Meta Language. This object–meta distinction (Tarski 1936; Carnap 1937) isn't limited to logical languages. For instance, in a Russian language lesson given in English, we can consider Russian to be the object language and English the metalanguage.

Abstract and concrete syntax

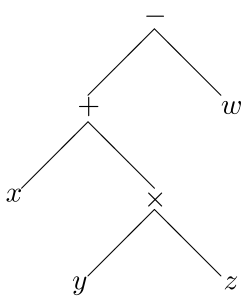
Fine details of syntax are of no fundamental importance. Some mathematics is typed, some is handwritten, and people make various essentially arbitrary choices that do not change anything about the structural way symbols are used together. When mechanizing logic on the computer, we will, for simplicity, restrict ourselves to the usual stock of ASCII characters,[†] which includes unaccented Latin letters, numbers and some common punctuation signs and spaces. For the fancy letters and special symbols that many logicians use, we will use other letters or words, e.g. 'forall' instead of '$\forall$'. We will, however, continue to employ the usual symbols in theoretical discussions. This continual translation may even be helpful to the reader who hasn't seen or understood the symbols before.

Regardless of how the symbolic expressions are read or written, it's more convenient to manipulate them in a form better reflecting their structure. Consider the expression '$x + y \times z - w$' in ordinary algebra. This linear form

[†] See `en.wikipedia.org/wiki/ASCII`.

obscures the meaningful structure. To understand which operators have been applied to which subexpressions, or even what constitutes a subexpression, we need to know rules of precedence and associativity, e.g. that '$\times$' 'binds tighter' than '$+$'. For instance, despite their apparent similarity in the linear form, '$y \times z$' is a subexpression while '$x + y$' is not. Even if we make the structure explicit by fully bracketing it as '$(x + (y \times z)) - w$', basic useful operations on expressions like finding subexpressions, or evaluating the expression for particular values of the variables, become tiresome to describe precisely; one needs to shuffle back and forth over the formula matching up brackets.

A 'tree' structure is much better: just as a family tree makes relations among family members clearly apparent, a tree representation of an expression displays its structure and makes most important manipulations straightforward. As in genealogy, it's customary to draw trees growing downwards on the printed page, so the same expression might be represented as follows:



Generally we refer to the (mainly linear) format used by people as the *concrete syntax*, and the structural (typically tree-like) form used for manipulations as the *abstract syntax*. Trees like the above are often called *abstract syntax trees* (ASTs) and are widely used as the internal representation of formal languages in all kinds of symbolic programs, including the compilers that translate high-level programming languages into machine instructions.

Despite their making the structure of an expression clearer, most people prefer not to think or communicate using trees, but to use the less structured concrete syntax.† Hence in our theorem-proving programs we will need to translate input from concrete syntax to abstract syntax, and translate output back from abstract syntax to concrete syntax. These two tasks, known to computer scientists as *parsing* and *prettyprinting*, are now well understood

† This is not to say that concrete syntax is necessarily a linear sequence of symbols. Mathematicians often use semi-graphical symbolism (matrix notation, commutative diagrams), and the pioneering logical notation introduced by Frege (1879) was tree-like.

and fairly routine. The small overhead of writing parsers and prettyprinters is amply repaid by the greater convenience of the tree form for internal manipulation. There are enthusiastic advocates of systems of concrete syntax such as 'Polish notation', 'reverse Polish notation (RPN)' and LISP 'S-expressions', where our expression would be denoted, respectively, by

```
- + x × y z w
x y z × + w -
(- (+ x (× y z)) w)
```

but we will use more traditional notation, with infix operators like '+' and rules of precedence and bracketing.†

1.6 Symbolic computation and OCaml

In the early days of modern computing it was commonly believed that computers were essentially devices for numeric calculation (Ceruzzi 1983). Their input and output devices were certainly biased in that direction: when Samuels wrote the first checkers (draughts) program at IBM in 1948, he had to encode the output as a number because that was all that could be printed.‡ However, it had already been recognized, long before Turing's theoretical construction of a universal machine (see Section 7.5), that the potential applicability of computers was much wider. For example, Ada Lovelace observed in 1842 (Huskey and Huskey 1980):§

Many persons who are not conversant with mathematical studies, imagine that because the business of [Babbage's analytical] engine is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.

There are now many programs that perform symbolic computation, including various quite successful 'computer algebra systems' (CASs). Theorem proving programs bear a strong family resemblance to CASs, and even overlap in some of the problems they can solve (see Section 5.11, for example).

† Originally the spartan syntax of LISP 'S-expressions' was to be supplemented by a richer and more conventional syntax of 'M-expressions' (meta-expressions), and this is anticipated in some of the early publications like the LISP 1.5 manual (McCarthy 1962). However, such was the popularity of S-expressions that M-expressions were seldom implemented and never caught on.

‡ Related in his speech to the 1985 International Joint Conference on Artificial Intelligence.

§ See `www.fourmilab.to/babbage/sketch.html`.

The preoccupations of those doing symbolic computation have influenced their favoured programming languages. Whereas many system programmers favour C, numerical analysts FORTRAN and so on, symbolic programmers usually prefer higher-level languages that make typical symbolic operations more convenient, freeing the programmer from explicit details of memory representation etc. We've chosen to use Objective CAML (OCaml) as the vehicle for the programming examples in this book. Our code does not use any of OCaml's more exotic features, and should be easy to port to related functional languages such as F♯, Standard ML or Haskell.

Our insistence on using explicit OCaml code may be disquieting for those with no experience of computer programming, or for those who only know imperative and relatively low-level languages like C or Java. However, we hope that with the help of Appendix 2 and additional study of some standard texts recommended at the end of this chapter, the determined reader will pick up enough OCaml to follow the discussion and play with the code. As a gentle introduction to symbolic computation in OCaml, we will now implement some simple manipulations in ordinary algebra, a domain that will be familiar to many readers.

The first task is to define a datatype to represent the abstract syntax of algebraic expressions. We will allow expressions to be built from numeric constants like `0`, `1` and `33` and named variables like `x` and `y` using the operations of addition ('`+`') and multiplication ('`*`'). Here is the corresponding recursive datatype declaration:

```
type expression =
   Var of string
 | Const of int
 | Add of expression * expression
 | Mul of expression * expression;;
```

That is, an expression is either a variable identified by a string, a constant identified by its integer value, or an addition or multiplication operator applied to two subexpressions. (A '`*`' indicates that the domain of a type constructor is a Cartesian product, so it can take two expressions as arguments. It is nothing to do with the multiplication being defined!) We can use the syntax constructors introduced by this type definition to create the symbolic representation for any particular expression, such as $2 \times x + y$:

```
# Add(Mul(Const 2,Var "x"),Var "y");;
- : expression = Add (Mul (Const 2, Var "x"), Var "y")
```

A simple but representative example of symbolic computation is applying specified transformation rules like $0 + x \longrightarrow x$ and $3 + 5 \longrightarrow 8$ to 'simplify' an expression. Each rule is expressed in OCaml by a starting and finishing *pattern*, e.g. `Add(Const(0),x) -> x` for a transformation $0+x \longrightarrow x$. (The special pattern '`_`' matches anything, so the last line ensures that if none of the other patterns match, `expr` is returned unchanged.) When the function is applied, OCaml will run through the rules in order and apply the first one whose starting pattern matches the input expression `expr`, replacing variables like `x` by the relevant subexpression.

```
let simplify1 expr =
  match expr with
    Add(Const(m),Const(n)) -> Const(m + n)
  | Mul(Const(m),Const(n)) -> Const(m * n)
  | Add(Const(0),x) -> x
  | Add(x,Const(0)) -> x
  | Mul(Const(0),x) -> Const(0)
  | Mul(x,Const(0)) -> Const(0)
  | Mul(Const(1),x) -> x
  | Mul(x,Const(1)) -> x
  | _ -> expr;;
```

However, simplifying just once is not necessarily adequate; we would like instead to simplify repeatedly until no further progress is possible. To do this, let us apply the above function in a bottom-up sweep through an expression tree, which will simplify in a cascaded manner. In traditional OCaml recursive style, we first simplify any immediate subexpressions as much as possible, then apply `simplify1` to the result:[†]

```
let rec simplify expr =
  match expr with
    Add(e1,e2) -> simplify1(Add(simplify e1,simplify e2))
  | Mul(e1,e2) -> simplify1(Mul(simplify e1,simplify e2))
  | _ -> simplify1 expr;;
```

Rather than a simple bottom-up sweep, a more sophisticated approach would be to mix top-down and bottom-up simplification. For example, if E is very large it would seem more efficient to simplify $0 \times E$ immediately to 0 without any examination of E. However, this needs to be implemented with care to ensure that all simplifiable subterms are simplified without the danger of looping indefinitely. Anyway, here is our simplification function in action on the expression $(0 \times x + 1) * 3 + 12$:

[†] We could leave `simplify1` out of the last line, since no simplification will be applicable to any expression reaching this case, but it seems more thematic to include it.

```
# let e = Add(Mul(Add(Mul(Const(0),Var "x"),Const(1)),Const(3)),
              Const(12));;
val e : expression =
  Add (Mul (Add (Mul (Const 0, Var "x"), Const 1), Const 3), Const 12)
# simplify e;;
- : expression = Const 15
```

Getting this far is straightforward using standard OCaml functional programming techniques: recursive datatypes to represent tree structures and the definition of functions via pattern-matching and recursion. We hope the reader who has not used similar languages before can begin to see why OCaml is appealing for symbolic computing. But of course, those who are fond of other programming languages are more than welcome to translate our code into them.

As planned, we will implement a parser and prettyprinter to translate between abstract syntax trees and concrete strings (‘`x + 0`’), setting them up to be invoked automatically by OCaml for input and output of expressions. We model our concrete syntax on ordinary algebraic notation, except that in a couple of respects we will follow the example of computer languages rather than traditional mathematics. We allow arbitrarily long ‘words’ as variables, whereas mathematicians traditionally use mostly single letters with superscripts and subscripts; this is especially important given the limited stock of ASCII characters. And we insist that multiplication is written with an explicit infix symbol (‘x `*` y’), rather than simple juxtaposition (‘$x\ y$’), which later on we will use for function application. In everyday mathematics we usually rely on informal cues like variable names and background knowledge to see at once that $f(x+1)$ denotes function application whereas $y(x+1)$ denotes multiplication, but this kind of context-dependent parsing is a bit more complicated to implement.

1.7 Parsing

Translating concrete into abstract syntax is a well-understood topic because of its central importance to programming language compilers, interpreters and translators. It is now conventional to separate the transformation into two separate stages:

- lexical analysis (scanning) decomposes the sequences of input characters into ‘tokens’ (roughly speaking, words);
- parsing converts the linear sequences of tokens into an abstract syntax tree.

For example, lexical analysis might split the input ‘`v10 + v11`’ into three tokens ‘`v10`’, ‘`+`’ and ‘`v11`’, coalescing adjacent alphanumeric characters into words and throwing away any number of spaces (and perhaps even line breaks) between these tokens. Parsing then only has to deal with sequences of tokens and can ignore lower-level details.

Lexing

We start by classifying characters into broad groups: spaces, punctuation, symbolic, alphanumeric, etc. We treat the underscore and prime characters as alphanumeric, in deference to the usual conventions in computing (‘`x_1`’) and mathematics (‘f'’). The following OCaml predicates tell us whether a character (actually, one-character string) belongs to a certain class:[†]

```
let matches s = let chars = explode s in fun c -> mem c chars;;

let space = matches " \t\n\r"
and punctuation = matches "()[]{},"
and symbolic = matches "~`!@#$%^&*-+=|\\:;<>.?/"
and numeric = matches "0123456789"
and alphanumeric = matches
  "abcdefghijklmnopqrstuvwxyz_'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";;
```

A token will be either a sequence of adjacent alphanumeric characters (like ‘`x`’ or ‘`size1`’), a sequence of adjacent symbolic characters (‘`+`’, ‘`<=`’), or a single punctuation character (‘`(`’).[‡] Lexical analysis, scanning left-to-right, will assume that a token is the longest possible, for instance that ‘`x1`’ is a single token, not two. We treat punctuation characters differently from other symbols just to avoid some counterintuitive effects of the ‘longest possible token’ rule, such as the detection of a token ‘`((`’ in the string ‘`((x + y) + z)`’.

Next we will define an auxiliary function `lexwhile` that takes a property `prop` of characters, such as one of the classifying predicates above, and a list of input characters, separating off as a string the longest initial sequence of that list of characters satisfying `prop`:

```
let rec lexwhile prop inp =
  match inp with
    c::cs when prop c -> let tok,rest = lexwhile prop cs in c^tok,rest
  | _ -> "",inp;;
```

[†] Of course, this is a very inefficient procedure. However, we care even less than usual about efficiency in these routines since parsing is not usually a critical component in overall runtime.

[‡] In the present example, the only meaningful symbolic tokens consist of a single character, like ‘`+`’. However, by allowing longer symbolic tokens we will be able to re-use this lexical analyzer unchanged in later work.

The lexical analyzer itself maps a list of input characters into a list of token strings. First any initial spaces are separated and thrown away, using `lexwhile space`. If the resulting list of characters is nonempty, we classify the first character and use `lexwhile` to separate the longest string of characters of the same class; for punctuation (or other unexpected) characters we give `lexwhile` an always-false property so it stops at once. Then we add the first character back on to the token and recursively analyze the rest of the input.

```
let rec lex inp =
  match snd(lexwhile space inp) with
    [] -> []
  | c::cs -> let prop = if alphanumeric(c) then alphanumeric
                        else if symbolic(c) then symbolic
                        else fun c -> false in
             let toktl,rest = lexwhile prop cs in
             (c^toktl)::lex rest;;
```

We can try the lexer on a typical input string, and another example reminiscent of C syntax to illustrate longer symbolic tokens.

```
# lex(explode "2*((var_1 + x') + 11)");;
- : string list =
["2"; "*"; "("; "("; "var_1"; "+"; "x'"; ")"; "+"; "11"; ")"]
# lex(explode "if (*p1-- == *p2++) then f() else g()");;
- : string list =
["if"; "("; "*"; "p1"; "--"; "=="; "*"; "p2"; "++"; ")"; "then"; "f";
 "("; ")"; "else"; "g"; "("; ")"]
```

Parsing

Now we want to transform a sequence of tokens into an abstract syntax tree. We can reflect the higher precedence of multiplication over addition by considering an expression like $2 * w + 3 * (x + y) + z$ to be a sequence of 'product expressions' (here '$2 * w$', '$3 * (x + y)$' and 'z') separated by '$+$'. In turn each product expression, say $2 * w$, is a sequence of 'atomic expressions' (here '2' and 'w') separated by '$*$'. Finally, an atomic expression is either a constant, a variable, or an arbitrary expression enclosed in brackets; note that we require *parentheses* (round brackets), though we could if we chose allow square brackets and/or braces as well. We can invent names for these three categories, say 'expression', 'product' and 'atom', and illustrate how each is built up from the others by a series of rules often called a 'BNF[†]

[†] BNF stands for 'Backus–Naur form', honouring two computer scientists who used this technique to describe the syntax of the programming language ALGOL. Similar grammars are used in formal language theory.

grammar'; read '$\longrightarrow$' as 'may be of the form' and '$|$' as 'or'.

$$\begin{array}{rcl}
\text{expression} & \longrightarrow & \text{product} + \cdots + \text{product} \\
\text{product} & \longrightarrow & \text{atom} * \cdots * \text{atom} \\
\text{atom} & \longrightarrow & (\text{expression}) \\
 & | & \text{constant} \\
 & | & \text{variable}
\end{array}$$

Since the grammar is already recursive ('expression' is defined in terms of itself, via the intermediate categories), we might as well use recursion to replace the repetitions:

$$\begin{array}{rcl}
\text{expression} & \longrightarrow & \text{product} \\
 & | & \text{product} + \text{expression} \\
\text{product} & \longrightarrow & \text{atom} \\
 & | & \text{atom} * \text{product} \\
\text{atom} & \longrightarrow & (\text{expression}) \\
 & | & \text{constant} \\
 & | & \text{variable}
\end{array}$$

This gives rise to a very direct way of parsing the input using three mutually recursive functions for the three different categories of expression, an approach known as *recursive descent parsing*. Each parsing function is given a list of tokens and returns a pair consisting of the parsed expression tree together with any unparsed input. Note that the pattern of recursion exactly matches the above grammar and simply examines tokens when necessary to decide which of several alternatives to take. For example, to parse an expression, we first parse a product, and then test whether the first unparsed character is '+'; if it is, then we make a recursive call to parse the rest and compose the results accordingly.

```
let rec parse_expression i =
  match parse_product i with
    e1,"+"::i1 -> let e2,i2 = parse_expression i1 in Add(e1,e2),i2
  | e1,i1 -> e1,i1
```

A product works similarly in terms of a parser for atoms:

```
and parse_product i =
  match parse_atom i with
    e1,"*"::i1 -> let e2,i2 = parse_product i1 in Mul(e1,e2),i2
  | e1,i1 -> e1,i1
```

and an atom parser handles the most basic expressions, including an arbitrary expression in brackets:

```
and parse_atom i =
  match i with
    [] -> failwith "Expected an expression at end of input"
  | "("::i1 -> (match parse_expression i1 with
                  e2,")"::i2 -> e2,i2
                | _ -> failwith "Expected closing bracket")
  | tok::i1 -> if forall numeric (explode tok)
               then Const(int_of_string tok),i1
               else Var(tok),i1;;
```

The 'right-recursive' formulation of the grammar means that we interpret repeated operations that lack disambiguating brackets as right-associative, e.g. $x+y+z$ as $x+(y+z)$. Had we instead defined a 'left-recursive' grammar:

$$\begin{array}{rcl} \text{expression} & \longrightarrow & \text{product} \\ & | & \text{expression} + \text{product} \end{array}$$

then $x+y+z$ would have been interpreted as $(x+y)+z$. For an associative operation like '+' it doesn't matter that much, since at least the meanings are the same, but for '$-$' this latter policy is clearly more appropriate.[†]

Finally, we define the overall parser via a wrapper function that explodes the input string, lexically analyzes it, parses the sequence of tokens and then finally checks that no input remains unparsed. We define a generic function for this, applicable to any core parser `pfn`, since it will be useful again later:

```
let make_parser pfn s =
  let expr,rest = pfn (lex(explode s)) in
  if rest = [] then expr else failwith "Unparsed input";;
```

We call our parser `default_parser`, and test it on a simple example:

```
# let default_parser = make_parser parse_expression;;
val default_parser : string -> expression = <fun>
# default_parser "x + 1";;
- : expression = Add (Var "x", Const 1)
```

But we don't even need to invoke the parser explicitly. Our setup exploits OCaml's quotation facility so that any French-style ≪quotation≫ will automatically have its body passed as a string to the function `default_parser`:[‡]

[†] Translating such a left-recursive grammar naively into recursive parsing functions would cause an infinite loop since `parse_expression` would just call itself directly right at the beginning and never get started on useful work. However, a small modification copes with this difficulty – see the definition of `parse_left_infix` in Appendix 3.

[‡] OCaml's treatment of quotations is programmable; our action of feeding the string to `default_parser` is set up in the file `Quotexpander.ml`.

```
# <<(x1 + x2 + x3) * (1 + 2 + 3 * x + y)>>;;
- : expression =
Mul (Add (Var "x1", Add (Var "x2", Var "x3")),
 Add (Const 1, Add (Const 2, Add (Mul (Const 3, Var "x"), Var "y"))))
```

The process by which parsing functions were constructed from the grammar is almost mechanical, and indeed there are tools to produce parsers automatically from slightly augmented grammars. However, we thought it worthwhile to be explicit about this programming task, which is not really so difficult and provides a good example of programming with recursive functions.

1.8 Prettyprinting

For presentation to the user we need the reverse transformation, from abstract to concrete syntax. A crude but adequate solution is the following:

```
let rec string_of_exp e =
  match e with
    Var s -> s
  | Const n -> string_of_int n
  | Add(e1,e2) -> "("^(string_of_exp e1)^" + "^(string_of_exp e2)^")"
  | Mul(e1,e2) -> "("^(string_of_exp e1)^" * "^(string_of_exp e2)^")";;
```

Brackets are necessary in general to reflect the groupings in the abstract syntax, otherwise we could mistakenly print, say '$6\times(x+y)$' as '$6\times x+y$'. Our function puts brackets uniformly round each instance of a binary operator, which is perfectly correct but sometimes looks cumbersome to a human:

```
# string_of_exp <<x + 3 * y>>;;
- : string = "(x + (3 * y))"
```

We would (probably) prefer to omit the outermost brackets, and others that are implicit in rules for precedence or associativity. So let's give `string_of_exp` an additional argument for the 'precedence level' of the operator of which the expression is an immediate subexpression. Now, brackets are only needed if the current expression has a top-level operator with lower precedence than this 'outer precedence' argument.

We arbitrarily allocate precedence 2 to addition, 4 to multiplication, and use 0 at the outermost level. Moreover, we treat the operators asymmetrically to reflect right-associativity, so the left-hand recursive subcall is given a slightly higher outer precedence to force brackets if iterated instances of the same operation are left-associated.

```
let rec string_of_exp pr e =
  match e with
    Var s -> s
  | Const n -> string_of_int n
  | Add(e1,e2) ->
        let s = (string_of_exp 3 e1)^" + "^(string_of_exp 2 e2) in
        if 2 < pr then "("^s^")" else s
  | Mul(e1,e2) ->
        let s = (string_of_exp 5 e1)^" * "^(string_of_exp 4 e2) in
        if 4 < pr then "("^s^")" else s;;
```

Our overall printing function will print with starting precedence level 0 and surround the result with the kind of quotation marks we use for input:

```
let print_exp e = Format.print_string ("<<"^string_of_exp 0 e^">>");;
```

As with the parser, we can set up the printer to be invoked automatically on any result of the appropriate type, using the following magic incantation (the hash is part of the directive that is entered, not the OCaml prompt):

```
#install_printer print_exp;;
```

Now we get output quite close to the concrete syntax we would naturally type in:

```
# <<x + 3 * y>>;;
- : expression = <<x + 3 * y>>
# <<(x + 3) * y>>;;
- : expression = <<(x + 3) * y>>
# <<1 + 2 + 3>>;;
- : expression = <<1 + 2 + 3>>
# <<((1 + 2) + 3) + 4>>;;
- : expression = <<((1 + 2) + 3) + 4>>
```

The main rough edge remaining is that expressions too large to fit on one line are not split up in an intelligent way to reflect the structure via the line breaks, as in the following example. The printers we use later (see Appendix 3) make a somewhat better job of this by employing a special OCaml library `Format`.

```
# <<(x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10) *
    (y1 + y2 + y3 + y4 + y5 + y6 + y7 + y8 + y9 + y10)>>;;
- : expression =
<<(x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10) * (y1 + y2 + y3 +
y4 + y5 + y6 + y7 + y8 + y9 + y10)>>
```

Having demonstrated the basic programming needed to support symbolic computation, we will end this chapter and move on to the serious study of logic and automated reasoning.