Logic and Mechanized Reasoning

Release 0.1

Jeremy Avigad
Seul Baek
Marijn J. H. Heule
Wojciech Nawrocki
Emre Yolcu

Sep 13, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Historical background

In the thirteenth century, Ramon Lull, an eccentric Franciscan monk on the Island of Majorca, wrote a work called the *Ars Magna*, which contains mechanical devices designed to aid reasoning. For example, Lull claimed that the reason that infidels did not accept the Christian god was that they failed to appreciate the multiplicity of God's attributes, so he designed nested paper circles with letters and words signifying these attributes. By rotating the circles, one obtained various combinations of the words and symbols, signifying compound attributes. For example, in one configuration, one might read off the result that God's greatness is true and good, and that his power is wise and great. Other devices and diagrams would assist in reasoning about virtues and vices, and so on.

Today, this sounds silly, but the work was based on three fundamental ideas that are still of central importance.

- First, we can use symbols, or tokens, to stand for ideas or concepts.
- Second, compound ideas and concepts are formed by putting together simpler ones.
- And, third, mechanical devices—even as simple a concentric rotating wheels—can serve as aids to reasoning.

The first two ideas go back to ancient times and can be found in the work of Aristotle. The third, however, is usually attributed to Lull, marking his work as the origin of mechanized reasoning.

Four centuries later, Lull's work resonated with Gottfied Leibniz, who invented calculus around the same time that Isaac Newton did so independently. Leibniz was also impressed by the possibility of symbolic representations of concepts and rules for reasoning. He spoke of developing a *characteristica universalis*—a universal language of thought—and a *calculus ratiocinator*—a calculus for reasoning. In a famous passage, he wrote:

> If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other (and if they so wish also to a friend called to help): Let us calculate.

The last phrase—*calculemus!* in the original Latin—has become a motto of computer scientists and computationally-minded mathematicians today.

The development of modern logic in the late eighteenth and early nineteen centuries began to bring Leibniz' vision to fruition. In 1931, Kurt Gödel wrote:

> The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

It is notable that the use of the word "mechanical" here—*mechanischen* in the original German—predates the modern computer by a decade or so.

What logicians from the time of Aristotle to the present day have in common is that they are all at least slightly crazy. They are driven by the view that knowledge is rooted in language and that the key to knowledge lies in having just the right symbolic representations of language and rules of use. But often it's the crazy people that change the world. The logical view of language and knowledge lies at the heart of computer science and provides the foundation for some of our most valuable technologies today, including programming languages, automated reasoning and AI, and databases.

That's what this course is about: the logician's view of the world, the power of symbolic representations of language, and the way those representations facilitate the mechanization of reasoning and the acquisition of knowledge.

The logicians' view complements the view from statistics and machine learning, where representations of knowledge tend to be very large, approximate, and hard to represent is succinct symbolic terms. Such methods have had stunning successes in recent years, but there are still branches of computer science and AI where symbolic methods are paramount. It is an important open question as to the best way to combine logical, statistical, and machine learning methods in the years to come.

1.2 An overview of this course

This course is designed to teach you the mathematical theory behind symbolic logic, with an eye towards putting it to good use. An interesting aspect of the course is that it develops three interacting strands in parallel:

- *Theory.* We will teach you the syntax and semantics of propositional and first-order logic. If time allows, we will give you a brief overview of related topics, like simple type theory and higher-order logic.
- *Implementation.* We will teach you how to implement logical syntax—terms and formulas—in a functional programming language called *Lean*. We will also teach you how to carry out fundamental operations and transformations on these objects.
- *Application.* We will show you how to use logic-based automated reasoning tools to solve interesting and difficult problems. In particular, we will use a SAT solver called CaDiCaL, an SMT solver called Z3, and a first-order theorem prover called Vampire (and by then you will understand what all these terms mean).

The first strand will be an instance of pure mathematics. We will build on the skills you have learned in Mathematical Foundations of Computer Science (15-151). The goal is to teach you to think about and talk about logic in a mathematically rigorous way.

The second strand will give you an opportunity to code up some of what you have learned and put it to good use. Our goal is to provide a foundation for you to use logic-based computational methods in the future, whether you choose to make use of them in small or large ways. In the third strand, for illustrative purposes, we will focus mainly on solving puzzles and combinatorial problems. This will give you a sense of how the tools can also be used on proof and constraint satisfaction problems that come up in fields like program verification, discrete optimization, and AI.

CHAPTER

TWO

MATHEMATICAL BACKGROUND

2.1 Induction and recursion on the natural numbers

In its most basic form, the principle of induction on the natural numbers says that if you want to prove that every natural number has some property, it suffices to show that zero has the property, and that whenever some number n has the property, so does $n + 1$. Here is an example.

Theorem

For every natural number n, $\sum_{i \le n} i = n(n + 1)/2$.

Proof

Use induction on n. In the base case, we have $\sum_{i \le 0} i = 0 = 0(0 + 1)/2$. For the induction step, assuming $\sum_{i \le n} i = n(n + 1)/2$, we have

$$\begin{aligned} \sum_{i \le n+1} i &= \sum_{i \le n} i + (n + 1) \\ &= n(n + 1)/2 + 2(n + 1)/2 \\ &= (n + 1)(n + 2)/2 \end{aligned}$$

The story is often told that Gauss, as a schoolchild, discovered this formula by writing

$$\begin{aligned} S &= 1 + \ldots + n \\ S &= n + \ldots + 1 \end{aligned}$$

and then adding the two rows and dividing by two. The proof by induction doesn't provide insight as to how one might *discover* the theorem, but once you have guessed it, it provides a short and effective means for establishing that it is true.

In a similar vein, you might notice that an initial segment of the odd numbers yields a perfect square. For example, we have $1 + 3 + 5 + 7 + 9 = 25$. Here is a proof of the general fact:

Theorem

For every natural number n, $\sum_{i \le n} (2i + 1) = (n + 1)^2$.

Proof

The base case is easy, and assuming the inductive hypothesis, we have

$$\begin{aligned}\sum_{i \le n+1} (2i+1) &= \sum_{i \le n} (2i+1) + 2(n+1) + 1 \\ &= (n+1)^2 + 2n + 3 \\ &= n^2 + 4n + 4 \\ &= (n+2)^2.\end{aligned}$$

A close companion to induction is the principle of *recursion*. Recursion enables us to define functions on the natural numbers, and induction allows us to prove things about them. For example, let $g : \mathbb{N} \to \mathbb{N}$ be the function defined by

$$\begin{aligned}g(0) &= 1 \\ g(n+1) &= (n+1) \cdot g(n)\end{aligned}$$

Then g is what is known as the *factorial* function, whereby $g(n)$ is conventionally written $n!$. The point is that if you don't know what the factorial function is, the two equations above provide a complete specification. There is exactly one function, defined on the natural numbers, that meets that description.

Here is an identity involving the factorial function:

Theorem

$\sum_{i \le n} i \cdot i! = (n+1)! - 1.$

Proof

The base case is easy. Assuming the claim holds for n, we have

$$\begin{aligned}\sum_{i \le n+1} i \cdot i! &= \sum_{i \le n} i \cdot i! + (n+1) \cdot (n+1)! \\ &= (n+1)! + (n+1) \cdot (n+1)! - 1 \\ &= (n+1)!(1 + (n+1)) - 1 \\ &= (n+2)! - 1\end{aligned}$$

This is a pattern found throughout mathematics and computer science: define functions and operations using recursion, and then use induction to prove things about them.

The *Towers of Hanoi* puzzle provides a textbook example of a problem that can be solved recursively. The puzzle consists of three pegs and disks of different diameters that slide onto the pegs. The initial configuration has n disks stacked on one of the pegs in decreasing order, with the largest one at the bottom and the smallest one at the top. Suppose the pegs are numbered 1, 2, and 3, with the disks starting on peg 1. The required task is to move all the disks from peg 1 to peg 2, one at a time, with the constraint that a larger disk is never placed on top of a smaller one.

```
To move n disks from peg A to peg B with auxiliary peg C:
    if n = 0
        return
    else
        move n - 1 disks from peg A to peg C using auxiliary peg B
        move 1 disk from peg A to peg B
        move n - 1 disks from peg C to peg B using auxiliary peg A
```

We will show in class that this requires $2^n - 1$ moves. The exercises below ask you to show that *any* solution requires at least this many moves.

2.2 Complete induction

As we have described it, the principle of induction is pretty rigid: in the inductive step, to show that $n + 1$ has some property, we can only use the corresponding property of n. The principle of *complete* induction is much more flexible.

Principle of complete induction

To show that every natural number n has some property, show that n has that property whenever all smaller numbers do.

As an exercise, we ask you to prove the principle of complete induction using the ordinary principle of induction. Remember that a natural number greater than or equal to 2 is *composite* if it can be written as a product of two smaller numbers, and *prime* otherwise.

Theorem

Every number greater than two can be factored into primes.

Proof

Let n be any natural number greater than or equal to 2. If n is prime, we are done. Otherwise, write $n = m \cdot k$, where m and k are smaller than n (and hence greater than 1). By the inductive hypothesis, m and k can each be factored into prime numbers, and combining these yields a factorization of n.

Here is another example we will discuss in class:

Theorem

For any $n \geq 3$, the sum of the angles in any n-gon is $180(n - 2)$.

The companion to complete induction on the natural numbers is a form of recursion known as course-of-values recursion, which allows you to define a function f by giving the value of $f(n)$ in terms of the value of f at arbitrary smaller values of n. For example, we can define the sequence of Fibonacci numbers as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{aligned}$$

The fibonacci numbers satisfy lots of interesting identities, some of which are given in the exercises.

In fact, you can define a function by recursion as long as *some* associated measure decreases with each recursive call. Define a function $f(n, k)$ for $k \leq n$ by

$$f(n,k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ f(n-1,k) + f(n-1,k-1) & \text{otherwise} \end{cases}$$

Here it is the first argument that decreases. In class, we'll discuss a proof that this defines the function

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

which is simultaneously equal to number of ways of choosing k objects out of n without repetition.

Finally, here is a recursive description of the greatest common divisor of two nonnegative integers:

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, mod(x, y)) & \text{otherwise} \end{cases}$$

where $mod(x, y)$ is the remainder when dividing x by y.

2.3 Generalized induction and recursion

The natural numbers are characterized inductively by the following clauses:

- 0 is a natural number.
- If x is a natural number, so is $succ(x)$.

Here the function $succ(x)$ is known as the *successor* function, namely, the function that, given any number, returns the next one in the sequence. The natural numbers structure is also sometimes said to be *freely generated* by this data. The fact that it is *generated* by 0 and $succ(x)$ means that it is the *smallest* set that contains 0 and is closed under $succ(x)$; in other words, any set of natural numbers that contains 0 and is closed under $succ(x)$ contains all of them. This is just the principle of induction in disguise. The fact that it is generated *freely* by these elements means that there is no confusion between them: 0 is not a successor, and if $succ(x) = succ(y)$, then $x = y$. Intuitively, being generated by 0 and $succ(x)$ means that any number can be represented by an expression built up from these, and being generated freely means that the representation is unique.

The natural numbers are an example of an *inductively defined structure*. These come up often in logic and computer science. It is often useful to define functions by recursion on such structures, and to use induction to prove things about them. We will describe the general schema here with some examples that often come up in computer science.

Let α be any data type. The set of all *lists* of elements of α, which we will write as $List(\alpha)$, is defined inductively as follows:

- The element *nil* is an element of $List(\alpha)$.
- If a is an element of α and ℓ is an element of $List(\alpha)$, then the element $cons(a, \ell)$ is an element of $List(\alpha)$.

Here *nil* is intended to describe the empty list, $[]$, and $cons(a, \ell)$ is intended to describe the result of adding a to the beginning of ℓ. So, for example, the list of natural numbers $[1, 2, 3]$ would be written $cons(1, cons(2, cons(3, nil)))$. Think of $List(\alpha)$ as having a constructor $cons(a, \cdot)$ for each a. Then, in the terminology above, $List(\alpha)$ is generated inductively by *nil* and those constructors.

Henceforth, for clarity, we'll use the notation $[]$ for *nil* and $a :: \ell$ for $cons(a, \ell)$. More generally, we can take $[a, b, c, \ldots]$ to be an abbreviation for $a :: (b :: (c :: \ldots []))$.

Saying that $List(\alpha)$ is inductively defined means that we principles of recursion and induction on it. For example, the following concatenates two lists:

$$\begin{aligned} append([], m) &= m \\ append((a :: \ell), m) &= a :: (append(\ell, m)) \end{aligned}$$

Here the recursion is on the first argument. As with the natural numbers, the recursive definition specifies what to do for each of the constructors. We'll use the notation $\ell +\!\!+\, m$ for $append(\ell, m)$, and with this notation, the two defining clauses read as follows:

$$\begin{aligned} [] +\!\!+\, m &= m \\ (a :: \ell) +\!\!+\, m &= a :: (\ell +\!\!+\, m) \end{aligned}$$

From the definition, we have $[] +\!\!+\, \ell = \ell$ for every ℓ, but $m +\!\!+\, [] = m$ is something we have to prove.

Proposition

For every m, we have $m +\!\!+ [] = m$.

Proof

We use induction on m. In the base case, we have $[] +\!\!+ [] = []$ from the definition of *append*. For the induction step, suppose we have $m +\!\!+ [] = m$. Then we also have

$$\begin{aligned}(a :: m) +\!\!+ [] &= a :: (m +\!\!+ []) \\ &= a :: m.\end{aligned}$$

The definition of the append function is an example of *structural recursion*, called that because the definition proceeds by recursion on the structure of the inductively defined type. In particular, there is a clause of the definition corresponding to each constructor. The proof we have just seen is an instance of *structural induction*, called that because, once again, there is part of the proof for each constructor. The base case, for *nil*, is straightforward, because that constructor has no arguments. The inductive step, for *cons*, comes with an inductive hypothesis because the *cons* constructor has a recursive argument. In class, we'll do a similar proof that the *append* operation is associative.

The following function (sometimes called *snoc*) appends a single element at the end:

$$\begin{aligned}\mathit{append1}([], a) &= \mathit{cons}(a, \mathit{nil}) \\ \mathit{append1}(\mathit{cons}(b, \ell), a) &= \mathit{cons}(b, \mathit{append1}(\ell, a))\end{aligned}$$

An easy induction on ℓ shows that, as you would expect, $\mathit{append1}(\ell, a)$ is equal to $\ell +\!\!+ [a]$.

The following function reverses a list:

$$\begin{aligned}\mathit{reverse}([]) &= [] \\ \mathit{reverse}(\mathit{cons}(a, \ell)) &= \mathit{append1}(\mathit{reverse}(\ell), a)\end{aligned}$$

In class, or for homework, we'll work through proofs that that the following holds for every pair of lists ℓ and m:

$$\mathit{reverse}(\ell +\!\!+ m) = \mathit{reverse}(m) +\!\!+ \mathit{reverse}(\ell)$$

Here is another example of a property that can be proved by induction:

$$\mathit{reverse}(\mathit{reverse}(\ell)) = \ell$$

From a mathematical point of view, this definition of the reverse function above is as good as any other, since it specifies the function we want unambiguously. But in Chapter 3 we will see that such a definition can also be interpreted as executable code in a functional programming language such as Lean. In this case, the execution is quadratic in the length of the list (think about why). The following definition is more efficient in that sense:

$$\begin{aligned}\mathit{reverseAux}([], m) &= m \\ \mathit{reverseAux}((a :: \ell), m) &= \mathit{reverseAux}(\ell, (a :: m))\end{aligned}$$

$$\mathit{reverse}'(\ell) = \mathit{reverseAux}(\ell, [])$$

The idea is that *reverseAux* adds all the elements of the first argument to the second one in reverse order. So the second arguments acts as an *accumulator*. In fact, because it is a tail recursive description, the code generated by Lean is quite efficient. In class, we'll discuss an inductive proof that $\mathit{reverse}(\ell) = \mathit{reverse}'(\ell)$ for every ℓ.

It is worth mentioning that structural induction is not the only way to prove things about lists, and structural recursion is not the only way to define functions by recursion. Generally speaking, we can assign any complexity measure to a data type, and do induction on complexity, as long as the measure is well founded. (This will be the case, for example, for measures that take values in the natural numbers, with the usual ordering on size.) For example, we can define the length of a list as follows:

$$\begin{aligned} \mathit{length}([]) &= 0 \\ \mathit{length}(a :: \ell) &= \mathit{length}(\ell) + 1 \end{aligned}$$

Then we can define a function f on lists by giving the value of $f(\ell)$ in terms of the value of f on smaller lists, and we can prove a property of lists using the fact that the property holds of all smaller lists as an inductive hypothesis. These are ordinary instances of recursion and induction on the natural numbers.

As another example, we consider the type of finite binary trees, defined inductively as follows:

- The element *empty* is a binary tree.
- If s and t are finite binary trees, so is the $\mathit{node}(s, t)$.

In this definition, *empty* is intended to denote the empty tree, and $\mathit{node}(s, t)$ is intended to denote the binary tree that consists of a node at the top and has s and t as the left and right subtrees, respectively.

Be careful: it is more common to take the set of binary trees to consist of only the *nonempty* trees, in which case, what we have defined here are called the *extended* binary trees. Adding the empty tree results in a nice inductive characterization. If we started with a one-node tree as the base case, we would have to allow for three types of compound tree: one type with a node and a subtree to the left, one with a node and a subtree to the right, and one with a node with both left and right subtrees.

We can count the number of nodes in an extended binary tree with the following recursive definition:

$$\begin{aligned} \mathit{size}(\mathit{empty}) &= 0 \\ \mathit{size}(\mathit{node}(s, t)) &= 1 + \mathit{size}(s) + \mathit{size}(t) \end{aligned}$$

We can compute the depth of an extended binary tree as follows:

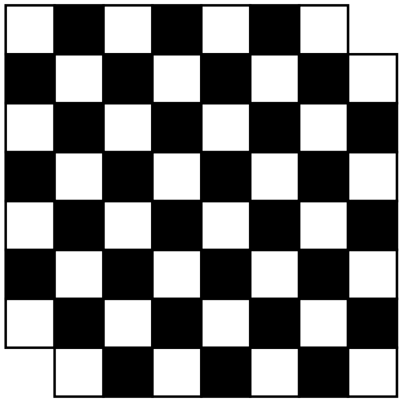
$$\begin{aligned} \mathit{depth}(\mathit{empty}) &= 0 \\ \mathit{depth}(\mathit{node}(s, t)) &= 1 + \max(\mathit{depth}(s), \mathit{depth}(t)) \end{aligned}$$

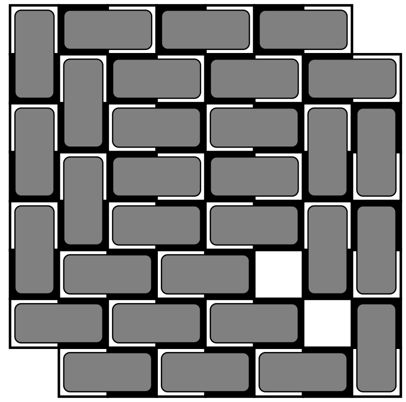
Again, be careful: many authors take the depth of a tree to be the length of the longest path from the root to a leaf, in which case, what we have defined here computes the depth *plus one* for nonempty trees.

2.4 Invariants

The *mutilated chessboard* problems involves an 8×8 chessboard with the top right and bottom left corners removed. Imagine you are given a set of dominoes, each of which can cover exactly two squares. It is possible to cover all the squares of the mutilated chessboard using dominoes, so that each square is covered by exactly one domino?

A moment's reflection shows that the answer is no. If you imagine the chessboard squares colored white and black in the usual way, you'll notice that the two squares we removed have the same color, say, black. That means that there are more white squares than black squares. On the other hand, every domino covers exactly one square of each color. So no matter how many dominoes we put down, we'll never have them color more white squares than black squares.





The fact that any way of putting down dominoes covers the same number of white and black squares is an instance of an *invariant*, which is a powerful idea in both mathematics and computer science. An invariant is something—a quantity, or a property—that doesn't change as something else does (in this case, the number of dominoes).

Often the natural way to establish an invariant uses induction. In this case, it is obvious that putting down one domino doesn't change the difference between the number of white and black squares covered, since each domino covers one of each. By induction on n, putting down n dominoes doesn't change the difference either.

The following puzzle, called the *MU puzzle*, comes from the book *Gödel, Escher, Bach* by Douglas Hofstadter. It concerns strings consisting of the letters *M*, *I*, and *U*. Starting with the string *MI*, we are allowed to apply any of the following rules:

1. Replace *sI* by *sIU*, that is, add a *U* to the end of any string that ends with *I*.
2. Replace *Ms* by *Mss*, that is, double the string after the initial *M*.
3. Replace *sIIIt* by *sUt*, that is, replace any three consecutive *Is* with a *U*.
4. Replace *sUUt* by *st*, that is, delete any consecutive pair of *Us*.

The puzzle asks whether it is possible to derive the string *MU*. The answer is no: it turns out that a string is derivable if and only if it consists of an *M* followed by any number of *Is* and *Us*, as long as the number of *Is* is not divisible by 3. In class, we'll prove the "only if" part of this equivalence. Try the "if" part if you like a challenge.

As a final example, in class we'll discuss the Golomb *tromino theorem*. A *tromino* is an L-shaped configuration of three squares. Golomb's theorem says that any $2^n \times 2^n$ chessboard with one square removed can be tiled with trominoes. We'll prove this together in class.

2.5 Exercises

1. Prove the formula for the sum of a geometric series:

$$\sum_{i<n} ar^i = \frac{a(r^n - 1)}{r - 1}$$

2. Prove that for every $n > 4$, $n! > 2^n$.
3. Show that the solution to the towers of Hanoi given in Section 2.1 is optimal: for every n, it takes at least *2^n - 1* moves to move all the disks from one peg to another.
4. Consider the variation on the towers of Hanoi problem in which you can only move a disk to an *adjacent* peg. In other words, you can move a disk from peg 1 to peg 2, from peg 2 to peg 1, from peg 2 to peg 3, or from peg 3 to peg 2, but not from peg 1 to peg 3 or from peg 3 to peg 1.

Describe a recursive procedure for solving this problem, and show that is requires $3^n - 1$ moves. If you are ambitious, show that this is optimal, and that it goes through all the 3^n valid positions.

5. Consider the variation on the towers of Hanoi in which pegs can be moved cyclicly: from peg 1 to peg 2, from peg 2 to peg 3, or from peg 3 to peg 1. Describe a recursive procedure to solve this problem.
6. Use the ordinary principle of induction to prove the principle of complete induction.
7. Let $F_0, F_1, F_2, \ldots$ be the sequence of Fibonacci numbers.
    1. Let α and β be the two roots of the equation $x^2 = x + 1$. Show that for every n, $F_n = (\alpha^n - \beta^n)/\sqrt{5}$.
    2. Show $\sum_{i<n} F_i = F_{n+1} - 1$.
    3. Show $\sum_{i \leq n} F_i^2 = F_n F_{n+1}$.
8. Show that with *n* straight lines we can divide the plane into at most $n^2 + n + 2$ regions, and that this is sharp.
9. Show that the recursive description of $\mathit{gcd}(x, y)$ presented in Section 2.3 correctly computes the greatest common divisor of x and y, where we define $\mathit{gcd}(0, 0)$ equal to 0. You can restrict attention to nonnegative values of x and y. (Hint: you can use the fact that for for every y not equal to 0, we can write $x = \mathit{div}(x, y) \cdot y + \mathit{mod}(x, y)$, where $\mathit{div}(x, y)$ is the integer part of x divided by y. First show that for every k, $\mathit{gcd}(x, y) = \mathit{gcd}(x + ky, y)$, and use that fact.
10. Use structural induction to prove

$$\mathit{reverse}(\ell +\!\!+\, m) = \mathit{reverse}(m) +\!\!+\, \mathit{reverse}(\ell).$$

11. Use structural induction to prove

$$\mathit{reverse}(\mathit{reverse}(\ell)) = \ell.$$

12. Prove that for every ℓ we have

$$\mathit{reverse}'(\ell) = \mathit{reverse}(\ell).$$

13. Prove that for every ℓ and m we have

$$\mathit{length}(\ell +\!\!+\, m) = \mathit{length}(\ell) + \mathit{length}(m).$$

14. How many binary trees of depth n are there? Prove your answer is correct.
15. Show that a string is derivable in the MU puzzle if and only if it consists of an M followed by any number of Is and Us, as long as the number of Is is not divisible by 3.

CHAPTER

THREE

USING LEAN

3.1 About Lean

Lean is a new programming language and interactive proof assistant being developed at Microsoft Research. It is currently in an experimental, development stage, which makes it a risky choice for this course. But in many ways it is an ideal system for working with logical syntax and putting logic to use. Lean is an exciting project, and the system fun to use. So please bear with us. Using Lean puts us out on the frontier, but if you adopt a pioneering attitude, you will be in a good position to enjoy all the cool things that Lean has to offer.

You can learn more about Lean on the Lean home page, on the Lean community home page, and by asking questions on the Lean Zulip chat, which you are heartily encouraged to join. To be more precise, there are currently two versions of Lean:

- Lean 3 is reasonably stable, and primarily an interactive proof assistant. It has a very large mathematical library, known as mathlib.
- Lean 4 is being designed as a performant programming language, and it is still under development. It can also be used as a proof assistant, though it does not yet have a substantial library. Its language and syntax are similar to that of Lean 3, but it is not backward compatible.

In this course, we will use Lean 4, even though it is still under development. It has the rough beginnings of a user manual and there is a tutorial on the underlying foundation. As we will see, Lean has a lot of features that make that worthwhile. In particular, Lean 4 is designed to be an ideal language for implementing powerful logic-based systems, as evidenced by the fact that most of Lean 4 is implemented in Lean 4 itself.

The goal of this section is to give you a better sense of what Lean is, how it can possibly be a programming language and proof assistant at the same time, and why that makes sense. The rest of the introduction will give you a quick tour of some of its features, and we will learn more about them as the course progresses.

At the core, Lean is an implementation of a formal logical foundation known as *type theory*. More specifically, it is an implementation of *dependent type theory*, and even more specifically than that, it implements a version of the *Calculus of Inductive Constructions*. Saying that it implements a formal logic foundation means that there is a precise grammar for writing expressions, and precise rules for using them. In Lean, every well-formed expression has a type.

```
#check 2 + 2
#check -5
#check [1, 2, 3]
#check #[1, 2, 3]
#check (1, 2, 3)
#check "hello world"
#check true
#check fun x => x + 1
#check fun x => if x = 1 then "yes" else "no"
```

You can find this example in the file *using_lean/examples1.lean* in the *LAMR/Examples* folder of the course repository. We recommend copying that entire folder in the *User* folder, so you can edit the files and try examples of your own. You can always find the original file in the folder *LAMR/Examples*, which you should not edit.

If you hover over the *#check* statements or move your cursor to one of these lines and check the information window, Lean reports the result of the command. It tells you that *2 + 2* has type *Nat*, *-5* has type *Int*, and so on. In fact, in the formal foundation, types are expressions as well. The types of all the expressions above are listed below:

```
#check Nat
#check Int
#check List Nat
#check Array Nat
#check Nat × Nat × Nat
#check String
#check Bool
#check Nat → Nat
#check Nat → String
```

Now Lean tells you each of these has type *Type*, indicating that they are all data types. If you know the type of an expression, you can ask Lean to confirm it:

```
#check (2 + 2 : Nat)
#check ([1, 2, 3] : List Nat)
```

Lean will report an error if it cannot construe the expression as having the indicated type.

In Lean, you can define new objects with the *def* command. The new definition becomes part of the *environment*: the defined expression is associated with the identifier that appears after the word *def*.

```
def four : Nat := 2 + 2

def isOne (x : Nat) : String := if x = 1 then "yes" else "no"

#check four
#print four

#check isOne
#print isOne
```

The type annotations indicate the intended types of the arguments and the result, but they can be omitted when Lean can infer them from the context:

```
def four' := 2 + 2

def isOne' x := if x = 1 then "yes" else "no"
```

So far, so good: in Lean, we can define expressions and check their types. What makes Lean into a programming language is that the logical foundation has a computational semantics, under which expressions can be *evaluated*.

```
#eval four
#eval isOne 3
#eval isOne 1
```

The *#eval* command evaluates the expression and then displays the return value. Evaluation can also have *side effects*, which are generally related to system IO. For example, displaying the string "Hello, world!" is a side effect of the following evaluation:

```
#eval IO.println "Hello, world!"
```

Theoretical computer scientists are used to thinking about programs as expressions and identifying the act of running the program with the act of evaluating the expression. In Lean, this view is made manifest, and the expressions are defined in a formal system with a precise specification.

But what makes Lean into a proof assistant? To start with, some expressions in the proof system express propositions:

```
#check 2 + 2 = 4
#check 2 + 2 < 5
#check isOne 3 = "no"
#check 2 + 2 < 5 ∧ isOne 3 = "no"
```

Lean confirms that each of these is a proposition by reporting that each of them has type *Prop*. Notice that they do not all express *true* propositions; theorem proving is about certifying the ones that are. But the language of Lean is flexible enough to express just about any meaningful mathematical statement at all. For example, here is the statement of Fermat's last theorem:

```
def Fermat_statement : Prop :=
∀ a b c n : Nat, a * b * c ≠ 0 ∧ n > 2 → a^n + b^n ≠ c^n
```

In Lean's formal system, data types are expressions of type *Type*, and if *T* is a type, an expression of type *T* denotes an object of that type. We have also seen that propositions are expressions of type *Prop*. In the formal system, if *P* is a proposition, a proof of *P* is just an expression of type *P*. This is the final piece of the puzzle: we use Lean as a proof assistant by writing down a proposition *P*, writing down an expression *p*, and asking Lean to confirm that *p* has type *P*. The fact that *2 + 2 = 4* has an easy proof, that we will explain later:

```
theorem two_plus_two_is_four : 2 + 2 = 4 := rfl
```

In contrast, proving Fermat's last theorem is considerably harder.

```
theorem Fermat_last_theorem : Fermat_statement := sorry
```

Lean knows that *sorry* is not a real proof, and it flags a warning there. If you manage to replace *sorry* by a real Lean expression, please let us know. We will be very impressed.

So, in Lean, one can write programs and execute them, and one can state propositions and prove them. In fact, one can state propositions about programs and then prove those statements as well. This is known as *software verification*; it is a means of obtaining a strong guarantee that a computer program behaves as intended, something that is important, say, if you are using the software to control a nuclear reactor or fly an airplane.

This course is not about software verification. We will be using Lean 4 primarily as a programming language, one in which we can easily define logical expressions and manipulate them. To a small extent, we will also write some simple proofs in Lean. This will help us think about proof systems and rules, and understand how they work. Taken together, these two activities embody the general vision that animates this course: knowing how to work with formally specified expressions and rules opens up a world of opportunity. It is the key to unlocking the secrets of the universe.

3.2 Using Lean as a functional programming language

There is a preliminary user's manual for Lean, still a work in progress, here. The fact that Lean is a functional programming language means that instead of presenting a program as a list of instructions, you simply *define* functions and ask Lean to evaluate them.

```
def foo n := 3 * n + 7

#eval foo 3
#eval foo (foo 3)

def bar n := foo (foo n) + 3

#eval bar 3
#eval bar (bar 3)
```

There is no global state: any value a function can act on is passed as an explicit argument and is never changed. For that reason, functional programming languages are amenable to parallelization.

Nonetheless, Lean can do handle system IO using the *IO monad*, and can accommodate an imperative style of programming using *do notation*.

```
def printExample : IO Unit:= do
  IO.println "hello"
  IO.println "world"

#eval printExample
```

Recursive definitions are built into Lean.

```
def factorial : Nat → Nat
  | 0       => 1
  | (n + 1) => (n + 1) * factorial n

#eval factorial 10
#eval factorial 100
```

Here is a solution to the Towers of Hanoi problem:

```
def hanoi (numPegs start finish aux : Nat) : IO Unit :=
  match numPegs with
  | 0     => pure ()
  | n + 1 => do
      hanoi n start aux finish
      IO.println s!"Move disk {n + 1} from peg {start} to peg {finish}"
      hanoi n aux finish start

#eval hanoi 7 1 2 3
```

You can also define things by recursion on lists:

```
def addNums : List Nat → Nat
  | []    => 0
  | a::as => a + addNums as

#eval addNums [0, 1, 2, 3, 4, 5, 6]
```

In fact, there are a number of useful functions built into Lean's library. The function *List.range n* returns the list *[0, 1, …, n-1]*, and the functions *List.map* and *List.foldl* and *List.foldr* implement the usual map and fold functions for lists. By opening the *List* namespace, we can refer to these as *range*, *map*, *foldl*, and *foldr*. In the examples below, the dollar sign has the same effect as putting parentheses around everything that appears afterward.

```
#eval List.range 7

section
open List

#eval range 7
#eval addNums $ range 7
#eval map (fun x => x + 3) $ range 7
#eval foldl (. + .) 0 $ range 7

end
```

The scope of the *open* command is limited to the section, and the cryptic inscription *(. + .)* is notation for the addition function. Lean also supports projection notation that is useful when the corresponding namespace is not open:

```
def myRange := List.range 7
#eval myRange.map fun x => x + 3
```

Because *myRange* has type *List Nat*, Lean interprets *myrange.map fun x => x + 3* as *List.map (fun x => x + 3) myrange*. In other words, it automatically interprets *map* as being in the *List* namespace, and then it interprets *myrange* as the first *List* argument.

This course assumes you have some familiarity with functional programming. There is a free online textbook, Learn You a Haskell for Great Good that you might find helpful; porting some of the examples there to Lean is a good exercise. We will all suffer from the fact that documentation for Lean 4 barely exists at the moment, but we will do our best to provide you with enough examples for you to be able to figure out how to do what you need to do. One trick is to nose around the Lean code base itself. If you ctrl-click on the name of a function in the Lean library, the editor will jump to the definition, and you can look around and see what else is there. Another strategy is simply to ask us, ask each other, or ask questions on the Lean Zulip chat. We are all in this together.

When working with a functional programming language, there are often clever tricks for doing things that you may be more comfortable doing in an imperative programming language. For example, as explained in Section 2.3, here are Lean's definitions of the *reverse* and *append* functions for lists:

```
namespace hidden

def reverseAux : List α → List α → List α
  | [],   r => r
  | a::l, r => reverseAux l (a::r)

def reverse (as : List α) :List α :=
  reverseAux as []

protected def append (as bs : List α) : List α :=
  reverseAux as.reverse bs

end hidden
```

The function *reverseAux l r* reverses the elements of list *l* and adds them to the front of *r*. When called from *reverse l*, the argument *r* acts as an *accumulator*, storing the partial result. Because *reverseAux* is tail recursive, Lean's compiler can implement it efficiently as a loop rather than a recursive function. We have defined these functions in a namespace named *hidden* so that they don't conflict with the ones in Lean's library if you open the *List* namespace.

In Lean's foundation, every function is totally defined. In particular, every function that Lean computes has to terminates (in principle) on every input. Lean 4 will eventually support arbitrary recursive definitions in which the arguments in a recursive call decrease by some measure, but some work is needed to justify these calls in the underlying foundation. In the meanwhile, we can always cheat by using the *partial* keyword, which will let us perform arbitrary recursive calls.

```
partial def gcd m n :=
  if n = 0 then m else gcd n (m % n)

#eval gcd 45 30
#eval gcd 37252 49824
```

Using *partial* takes us outside the formal foundation; Lean will not let us prove anything about *gcd* when we define it this way. Using *partial* also makes it easy for us to shoot ourselves in the foot:

```
partial def bad (n : Nat) : Nat := bad (n + 1)
```

On homework exercises, you should try to use structural recursion when you can, but don't hesitate to use *partial* whenever Lean complains about a recursive definition. We will not penalize you for it.

The following definition of the Fibonacci numbers does not require the *partial* keyword:

```
def fib' : Nat → Nat
  | 0 => 0
  | 1 => 1
  | n + 2 => fib' (n + 1) + fib' n
```

But it is inefficient; you should convince yourself that the natural evaluation strategy requires exponential time. The following definition avoids that.

```
def fibAux : Nat → Nat × Nat
  | 0     => (0, 1)
  | n + 1 => let p := fibAux n
             (p.2, p.1 + p.2)

def fib n := (fibAux n).1

#eval (List.range 20).map fib
```

Producing a *list* of Fibonacci numbers, however, as we have done here is inefficient; you should convince yourself that the running time is quadratic. In the exercises, we ask you to define a function that computes a list of Fibonacci values with running time linear in the length of the list.

3.3 Inductive data types in Lean

One reason that computer scientists and logicians tend to like functional programming languages is that they often provide good support for defining inductive data types and then using structural recursion on such types. For example, here is a Lean definition of the extended binary trees that we defined in mathematical terms in Section 2.3:

```
import Init

inductive BinTree
  | empty : BinTree
  | node  : BinTree → BinTree → BinTree

open BinTree
```

The command *import Init* imports a part of the initial library for us to use. The command *open BinTree* allows us to write *empty* and *node* instead of *BinTree.empty* and *BinTree.node*. Note the Lean convention of capitalizing the names of data types.

We can now define the functions *size* and *depth* by structural recursion:

```
def size : BinTree → Nat
  | empty     => 0
  | node a b => 1 + size a + size b

def depth : BinTree → Nat
  | empty     => 0
  | node a b => 1 + Nat.max (depth a) (depth b)

def example_tree := node (node empty empty) (node empty (node empty empty))

#eval size example_tree
#eval depth example_tree
```

In fact, the *List* data type is also inductively defined.

```
#print List
```

You should try writing the inductive definition on your own. Call it *MyList*, and then try *#print MyList* to see how it compares.

3.4 Using Lean as a proof assistant

A feature of working with a system like Lean, which is based on a formal logical foundation, is that you can not only define data types and functions but also prove things about them. The goal of this section is to give you a flavor of using Lean as a proof assistant. It isn't easy: Lean syntax is finicky and its error messages are often inscrutable. In class, we'll try to give you some pointers as to how to interact with Lean to construct proofs. The examples in this section will serve as a basis for discussion.

Remember that Lean's core library defines the *List* data type and notation for it. In the example below, we import the library, open the namespace, declare some variables, and try out the notation.

```
import Init

open List

variable {α : Type}
variable (as bs cs : List α)
variable (a b c : α)

#check a :: as
#check as ++ bs

example : [] ++ as = as := nil_append as

example : (a :: as) ++ bs = a :: (as ++ bs) := cons_append a as bs
```

The *variable* command does not do anything substantive. It tells Lean that when the corresponding identifiers are used in definitions and theorems that follow, they should be interpreted as arguments to those theorems and proofs, with the indicated types. The curly brackets around the declaration α *: Type* indicate that that argument is meant to be *implicit*, which is to say, users do not have to write it explicitly. Rather, Lean is expected to infer it from the context.

The library proves the theorems *[] ++ as* and *(a :: as) ++ bs = a :: (as ++ bs)* under the names *nil_append* and *cons_append*, respectively. You can see them by writing *#check nil_append* and *#check cons_append*. Remember that we took these to be the defining equations for the *append* function in Section 2.3. Although Lean uses a different definition of the append function, for illustrative purposes we will treat them as the defining equations and base our subsequent proofs on that.

Lean's library also proves *as ++ []* under the name *append_nil*, but to illustrate how proofs like this go, we will prove it again under the name *append_nil'*.

```
theorem append_nil' : as ++ [] = as := by
  induction as with
  | nil => rw [nil_append]
  | cons a as ih => rw [cons_append, ih]
```

In class, we will help you make sense of this. The *by* command tell Lean that we are going to write a *tactic* proof. In other words, instead of writing the proof as an expression, we are going to give Lean a list of instructions that tell it how to prove the theorem. At the start of the tactic proof, the theorem in question is our *goal*. At each step, tactics act on one more more of the remaining goals; when no more goals remain, the theorem is proved.

In this case, there are only two tactics that are needed. The *induction* tactic, as the name suggests, sets up a proof by induction, and the *rw* tactic *rewrites* the goal using given equations. Moving the cursor around in the editor windows shows you the goals at the corresponding state of the proof.

```
theorem append_assoc' : as ++ bs ++ cs = as ++ (bs ++ cs) := by
  induction as with
  | nil => rw [nil_append, nil_append]
  | cons a as ih => rw [cons_append, cons_append, ih, ←cons_append]
```

Here is a similar proof of the associativity of the *append* function. Note that the left arrow in the expression *←cons_append* tell Lean that we want to use the equation from right to left instead of from left to right.

Now let us consider Lean's definition of the *reverse* function:

```
theorem reverse_def : reverse as = reverseAux as [] := rfl

theorem reverseAux_nil : reverseAux [] as = as := rfl

theorem reverseAux_cons : reverseAux (a :: as) bs = reverseAux as (a :: bs) := rfl
```

We will use these identities in the proofs that follow. Let's think about what it would take to prove the identity *reverse (as ++ bs) = reverse bs ++ reverse as*. Since *reverse* is defined in terms of *reverseAux*, we should expect to have to prove something about *reverseAux*. And since the identity mentions the *append* function, it is natural to try to characterize the way that *reverseAux* interacts with *append*. These are the two identities we need:

```
theorem reverseAux_append : reverseAux (as ++ bs) cs = reverseAux bs (reverseAux as␣
↪cs) := by
  induction as generalizing cs with
  | nil => rw [nil_append, reverseAux_nil]
  | cons a as ih => rw [cons_append, reverseAux_cons, reverseAux_cons, ih]

theorem reverseAux_append' : reverseAux as (bs ++ cs) = reverseAux as bs ++ cs := by
  induction as generalizing bs with
  | nil => rw [reverseAux_nil, reverseAux_nil]
  | cons a as ih => rw [reverseAux_cons, reverseAux_cons, ←cons_append, ih]
```

Note the *generalizing* clause in the induction. What it means is that what we are proving by induction on *as* is that the identity holds *for every choice of bs*. This means that, when we apply the inductive hypothesis, we can apply it to any choice of the parameter *bs*. You should try deleting the *generalizing* clause to see what goes wrong when we omit it.

With those facts in hand, we have the identity we are after:

```
theorem reverse_append : reverse (as ++ bs) = reverse bs ++ reverse as := by
  rw [reverse_def, reverseAux_append, reverse_def, ←reverseAux_append', nil_append,
      reverse_def]
```

3.5 Using Lean as an imperative programming language

The fact that Lean is a functional programming language means that there is no global notion of *state*. Functions take values as input and return values as output; there are no global or even local variables that are changed by the result of a function call.

But one of the interesting features of Lean is a functional programming language is that it incorporates features that make it *feel* like an imperative programming language. The following example shows how to print out, for each value i less than 100, the the sum of the numbers up to i.

```
def showSums : IO Unit := do
  let mut sum := 0
  for i in [0:100] do
    sum := sum + i
    IO.println s!"i: {i}, sum: {sum}"

#eval showSums
```

You can use a loop not just to print values, but also to compute values. The following is a boolean test for primality:

```
def isPrime (n : Nat) : Bool := do
  if n < 2 then false else
    for i in [2:n] do
      if n % i = 0 then
        return false
      if i * i > n then
        return true
    true
```

You can use such a function with the list primitives to construct a list of the first 10,000 prime numbers.

```
#eval (List.range 10000).filter isPrime
```

But you can also use it with Lean's support for *arrays*. Within the formal foundation these are modeled as lists, but the compiler implements them as dynamic arrays, and for efficiency it will modify values rather than copy them whenever the old value is not referred to by another part of an expression.

```
def primes (n : Nat) : Array Nat := do
  let mut result := #[]
  for i in [2:n] do
    if isPrime i then
      result := result.push i
  result

#eval (primes 10000).size
```

Notice the notation: *#[]* denotes a fresh array (Lean infers the type from context), and the *Array.push* function adds a new element at the end of the array.

The following example shows how to compute a two-dimensional array and print it out.

```
def mulTable (n : Nat) : Array (Array Nat) := do
  let mut table := #[]
  for i in [:n] do
    let mut row := #[]
    for j in [:n] do
      row := row.push ((i + 1) * (j + 1))
    table := table.push row
  table

#eval mulTable 10

def printMulTable (n : Nat) : IO Unit := do
  let t := mulTable n
  for i in [:n] do
    for j in [:n] do
      IO.print s!"{t[i][j]} "
    IO.println ""

#eval printMulTable 10
```

3.6 Exercises

1. Using operations on *List*, write a Lean function that for every n returns the list of all the divisors of n that are less than n.
2. A natural number n is *perfect* if it is equal to the sum of the divisors less than n. Write a Lean function (with return type *Bool*) that determines whether a number n is perfect. Use it to find all the perfect numbers less than 1,000.
3. Define a recursive function $\mathit{sublists}(\ell)$ that, for every list ℓ, returns a list of all the sublists of ℓ. For example, given the list $[1, 2, 3]$, it should compute the list

   $$[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]].$$

   The elements need not be listed in that same order.
4. Prove in Lean that the length of $\mathit{sublists}(\ell)$ is $2^{\mathit{length}(\ell)}$.
5. Define a function $\mathit{permutations}(\ell)$ that returns a list of all the permutations of ℓ.
6. Prove in Lean that the length of $\mathit{permutations}(\ell)$ is $\mathit{factorial}(\mathit{length}(\ell))$.
7. Define in Lean a function that, assuming ℓ is a list of lists representing an $n \times n$ array, returns a list of lists representing the transpose of that array.
8. Write a program that solves the Tower of Hanoi problem with n disks on the assumption that disks can only be moved to an *adjacent* peg. (See Section 2.5.)
9. Write a program that solves the Tower of Hanoi problem with n disks on the assumption that disks can only be moved clockwise. (See Section 2.5.)
10. Define a Lean data type of binary trees in which every node is numbered by a label. Define a Lean function to compute the sum of the nodes in such a tree. Also write functions to list the elements in a preorder, postorder, and inorder traversal.

CHAPTER

FOUR

PROPOSITIONAL LOGIC

We are finally ready to turn to the proper subject matter of this course, logic. We will see that although propositional logic has limited expressive power, it can be used to carry out useful combinatorial reasoning in a wide range of applications.

4.1 Syntax

We start with a stock of variables $p_0, p_1, p_2, \ldots$ that we take to range over propositions, like "the sky is blue" or "2 + 2 = 5". We'll make the interpretation of propositional logic explicit in the next section, but, intuitively, propositions are things that can be either true or false. (More precisely, this is the *classical* interpretation of propositional logic, which is the one we will focus on in this course.) Each propositional variable is a *formula*, and we also include symbols $\top$ and $\bot$ for "true" and "false" respectively. We also provide means for building new formulas from old ones. The following is a paradigm instance of an inductive definition.

Definition

The set of propositional formulas is generated inductively as follows:

- Each variable p_i is a formula.
- $\top$ and $\bot$ are formulas.
- If A is a formula, so is $\neg A$ ("not A").
- If A and B are formulas, so are
  - $A \wedge B$ ("A and B"),
  - $A \vee B$ ("A or B"),
  - $A \to B$ ("A implies B"), and
  - $A \leftrightarrow B$ ("A if and only if B").

We will see later that there is some redundancy here; we could get by with fewer connectives and define the others in terms of those. Conversely, there are other connectives that can be defined in terms of these. But the ones we have included form a *complete* set of connectives, which is to say, any conceivable connective can be defined in terms of these, in a sense we will clarify later.

The fact that the set is generated inductively means that we can use recursion to define functions on the set of propositional formulas, as follows:

$$
\begin{aligned}
\mathit{complexity}(p_i) &= 0 \\
\mathit{complexity}(\top) &= 0 \\
\mathit{complexity}(\bot) &= 0 \\
\mathit{complexity}(\neg A) &= \mathit{complexity}(A) + 1 \\
\mathit{complexity}(A \wedge B) &= \mathit{complexity}(A) + \mathit{complexity}(B) + 1 \\
\mathit{complexity}(A \vee B) &= \mathit{complexity}(A) + \mathit{complexity}(B) + 1 \\
\mathit{complexity}(A \to B) &= \mathit{complexity}(A) + \mathit{complexity}(B) + 1 \\
\mathit{complexity}(A \leftrightarrow B) &= \mathit{complexity}(A) + \mathit{complexity}(B) + 1
\end{aligned}
$$

The function $\mathit{complexity}(A)$ counts the number of connectives. The function $\mathit{depth}(A)$, defined in a similar way, computes the depth of the parse tree.

$$
\begin{aligned}
\mathit{depth}(p_i) &= 0 \\
\mathit{depth}(\top) &= 0 \\
\mathit{depth}(\bot) &= 0 \\
\mathit{depth}(\neg A) &= \mathit{depth}(A) + 1 \\
\mathit{depth}(A \wedge B) &= \max(\mathit{depth}(A), \mathit{depth}(B)) + 1 \\
\mathit{depth}(A \vee B) &= \max(\mathit{depth}(A), \mathit{depth}(B)) + 1 \\
\mathit{depth}(A \to B) &= \max(\mathit{depth}(A), \mathit{depth}(B)) + 1 \\
\mathit{depth}(A \leftrightarrow B) &= \max(\mathit{depth}(A), \mathit{depth}(B)) + 1
\end{aligned}
$$

Here's an example of a proof by induction:

Theorem

For every formula A, we have $\mathit{complexity}(A) \leq 2^{\mathit{depth}(A)} - 1$.

Proof

In the base case, we have

$$\mathit{complexity}(p_i) = 0 = 2^0 - 1 = 2^{\mathit{depth}(p_i)} - 1,$$

and similarly for $\top$ and $\bot$. In the case for negation, assuming the claim holds of A, we have

$$
\begin{aligned}
\mathit{complexity}(\neg A) &= \mathit{complexity}(A) + 1 \\
&\leq 2^{\mathit{depth}(A)} - 1 + 1 \\
&\leq 2^{\mathit{depth}(A)} + 2^{\mathit{depth}(A)} - 1 \\
&\leq 2^{\mathit{depth}(A)+1} - 1 \\
&= 2^{\mathit{depth}(\neg A)} - 1.
\end{aligned}
$$

Finally, assuming the claim holds of A and B, we have

$$
\begin{aligned}
\mathit{complexity}(A \wedge B) &= \mathit{complexity}(A) + \mathit{complexity}(B) + 1 \\
&\leq 2^{\mathit{depth}(A)} - 1 + 2^{\mathit{depth}(B)} - 1 + 1 \\
&\leq 2 \cdot 2^{\max(\mathit{depth}(A), \mathit{depth}(B))} - 1 \\
&= 2^{\max(\mathit{depth}(A), \mathit{depth}(B))+1} - 1 \\
&= 2^{\mathit{depth}(A \wedge B)} - 1,
\end{aligned}
$$

and similarly for the other binary connectives.

In our metatheory, we will use variables $p, q, r, \dots$ to range over propositional variables, and $A, B, C, \dots$ to range over propositional formulas. The formulas p_i, $\top$, and $\bot$ are called *atomic* formulas. If A is a formula, B is a *subformula* of A if B occurs somewhere in A. We can make this precise by defining the set of subformulas of any formula A inductively as follows:

$$\begin{aligned} \mathit{subformulas}(A) &= \{A\} \quad \text{if } A \text{ is atomic} \\ \mathit{subformulas}(\neg A) &= \{\neg A\} \cup \mathit{subformulas}(A) \\ \mathit{subformulas}(A \star B) &= \{A \star B\} \cup \mathit{subformulas}(A) \cup \mathit{subformulas}(B) \end{aligned}$$

In the last clause, the star is supposed to represent any binary connective.

If A and B are formulas and p is a propositional variable, the notation $A[B/p]$ denotes the result of substituting B for p in A. Beware: the notation for this varies widely; $A[p \mapsto B]$ is also becoming common in computer science. The meaning is once again given by a recursive definition:

$$\begin{aligned} p_i[B/p] &= \begin{cases} B & \text{if } p \text{ is } p_i \\ p_i & \text{otherwise} \end{cases} \\ (\neg C)[B/p] &= \neg(C[B/p]) \\ (C \star D)[B/p] &= C[B/p] \star D[B/p] \end{aligned}$$

4.2 Semantics

Consider the formula $p \wedge (\neg q \vee r)$. Is it true? Well, that depends on the propositions p, q, and r. More precisely, it depends on whether they are true — and, in fact, that is all it depends on. In other words, once we specify which of p, q, and r are true and which are false, the truth value of $p \wedge (\neg q \vee r)$ is completely determined.

To make this last claim precise, we will use the set $\{\top, \bot\}$ to represent the truth values *true* and *false*. It doesn't really matter what sorts of mathematical objects those are, as long as they are distinct. You can take them to be the corresponding propositional formulas, or you can take $\top$ to be the number 1 and $\bot$ to be the number 0. A *truth assignment* is a function from propositional variables to the set $\{\top, \bot\}$, that is, a function which assigns a value of true or false to each propositional variable. Any truth assignment v extends to a function $\bar{v}$ that assigns a value of $\top$ or $\bot$ to any propositional formula. It is defined recursively as follows:

$$\begin{aligned} \bar{v}(p_i) &= v(p_i) \\ \bar{v}(\top) &= \top \\ \bar{v}(\bot) &= \bot \\ \bar{v}(\neg A) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bot \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \wedge B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \top \text{ and } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \vee B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \top \text{ or } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \to B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bot \text{ or } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \leftrightarrow B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bar{v}(B) \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

It is common to write $[\![A]\!]_v$ instead of $\bar{v}(A)$. Double-square brackets like these are often used to denote a semantic value that is assigned to a syntactic object. Think of $[\![A]\!]_v$ as giving the *meaning* of A with respect to the interpretation given by v. In this case, variables are interpreted as standing for truth values and the meaning of the formula is the resulting truth value, but in the chapters to come we will come across other semantic interpretations of this sort.

The following definitions are now fundamental to logic. Make sure you are clear on the terminology and know how to use it. If you can use these terms correctly, you can pass as a logician. If you get the terminology wrong, you'll be frowned upon.

- If $[\![A]\!]_v = \top$, we say that A is *satisfied* by v, or that v is a *satisfying assignment* for A. We also sometimes write $\models_v A$.
- A formula A is *satisfiable* if there is some truth assignment that satisfies it. A formula A is *unsatisfiable* if it is not satisfiable.
- A formula A is *valid*, or a *tautology* if it is satisfied by *every* truth assignment. In other words, A is valid if $[\![A]\!]_v = \top$ for every truth assignment v.
- If Γ is a set of propositional formulas, we say that Γ is *satisfied by* v if every formula in Γ is satisfied by v. In other words, Γ is satisfied by v if $[\![A]\!]_v = \top$ for every A in Γ.
- A set of formulas Γ is *satisfiable* if it is satisfied by some truth assignment v. Otherwise, it is *unsatisfiable*.
- If Γ is a set of propositional formulas and A is a propositional formula, we say Γ entails A if every truth assignment that satisfies Γ also satisfies A. Roughly speaking, this says that whenever the formulas in Γ are true, then A is also true. In this case, A is also said to be a *logical consequence* of Γ.
- Two formulas A and B are *logically equivalent* if each one entails the other, that is, we have $\{A\} \models B$ and $\{B\} \models A$. When that happens, we write $A \equiv B$.

There is a lot to digest here, but it is important that you become comfortable with these definitions. The mathematical analysis of truth and logical consequence is one of the crowning achievements of modern logic, and this basic framework for reasoning about expressions and their meaning has been applied to countless other settings in logic and computer science.

You should also get used to using semantic notions in proofs. For example:

Theorem

A propositional formula A is valid if and only if $\neg A$ is unsatisfiable.

Proof

A is valid if and only if $[\![A]\!]_v = \top$ for every truth assignment v. By the definition of $[\![\neg A]\!]_v$, this happens if and only if $[\![\neg A]\!]_v = \bot$ for every v, which is the same as saying that $\neg A$ is unsatisfiable.

4.3 Calculating with propositions

Remember that Leibniz imagined that one day we would be able to calculate with propositions. What he had noticed is that propositions, like numbers, obey algebraic laws. Here are some of them:

- $A \vee \neg A \equiv \top$
- $A \wedge \neg A \equiv \bot$
- $\neg\neg A \equiv A$

- $A \vee A \equiv A$
- $A \wedge A \equiv A$
- $A \vee \bot \equiv A$
- $A \wedge \bot \equiv \bot$
- $A \vee \top \equiv \top$
- $A \wedge \top \equiv A$
- $A \vee B \equiv B \vee A$
- $A \wedge B \equiv B \wedge A$
- $(A \vee B) \vee C \equiv A \vee (B \vee C)$
- $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$
- $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
- $A \wedge (A \vee B) \equiv A$
- $A \vee (A \wedge B) \equiv A$

The equivalences $\neg(A \wedge B) \equiv \neg A \vee \neg B$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$ are known as *De Morgan's laws*. It is not hard to show that all the logical connectives respect equivalence, and hence substituting equivalent formulas for a variable in a formula results in equivalent formulas. This means that, as Leibniz imagined, we can prove that a Boolean formula is valid by calculating to show that it is equivalent to $\top$. Here is an example.

Theorem

For any propositional formulas A and B, we have $(A \wedge \neg B) \vee B \equiv A \vee B$.

Proof

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

Mathematicians have a trick, called *quotienting*, for turning an equivalence relation into an equality. If we interpret A, B, and C as *equivalence classes* of formulas instead of formulas, the equivalences listed above become identities. The resulting algebraic structure is known as a *Boolean algebra*, and we can view the preceding proof as establishing an identity that holds in any Boolean algebra. The same trick is used, for example, to interpret an equivalence between numbers modulo 12, like $5 + 9 \equiv 2$ as an identity on the structure $\mathbb{Z}/12\mathbb{Z}$.

4.4 Complete sets of connectives

You may have noticed that our choice of connectives is redundant. For example, the following equivalences show that we can get by with $\neg$, $\vee$, and $\bot$ alone:

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

We can even define $\bot$ as $P \wedge \neg P$ for any propositional variable P, though that has the sometimes annoying consequence that we cannot express the constants $\top$ and $\bot$ without using a propositional variable.

Let $f(x_0, \ldots, x_{n-1})$ be a function that takes n truth values and returns a truth value. A formula A with variables $p_0, \ldots, p_{n-1}$ is said to *represent* f if for every truth assignment *v*,

$$[\![A]\!]_v = f(v(p_0), \ldots, v(p_{n-1})).$$

If you think of f as a truth table, this says that the truth table of A is f.

A set of connectives is said to be *complete* if every function f is represented by some formula A involving those connectives. In class, we'll discuss how to prove that $\{\wedge, \neg\}$ is a complete set of connectives in that sense.

It is now straightforward to show that a certain set of connectives is complete: just show how to define $\vee$ and $\neg$ in terms of them. Showing that a set of connectives is *not* complete typically requires some more ingenuity. One idea, as suggested in Section 2.4, is to look for some invariant property of the formulas that *are* represented.

4.5 Normal forms

For both theoretical reasons and practical reasons, it is often useful to know that formulas can be expressed in particularly simple or convenient forms.

Definition

An *atomic* formula is a variable or one of the constants $\top$ or $\bot$. A *literal* is an atomic formula or a negated atomic formula.

Definition

The set of propositional formulas in *negation normal form* (NNF) is generated inductively as follows:

- Each literal is in negation normal form.
- If A and B are in negation normal form, then so are $A \wedge B$ and $A \vee B$.

More concisely, the set of formulas in negation normal form is the smallest set of formulas containing the literals and closed under conjunction and disjunction. If we identify $\top$ with $\neg\bot$ and $\neg\top$ with $\bot$, we can alternatively characterize the formulas in negation normal form as the smallest set of formulas containing $\top$, $\bot$, variables, and their negations, and closed under conjunction and disjunction.

Proposition

Every propositional formula is equivalent to one in negation normal form.

Proof

First use the identities $A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$ and $A \to B \equiv \neg A \vee B$ to get rid of $\leftrightarrow$ and $\to$. Then use De Morgan's laws together with $\neg\neg A \equiv A$ to push negations down to the atomic formulas.

More formally, we can prove by induction on propositional formulas A that both A and $\neg A$ are equivalent to formulas in negation normal form. (You should try to write that proof down carefully.) Putting a formula in negation normal form is reasonably efficient. You should convince yourself that if A is in negation normal form, then putting $\neg A$ in negation normal form amounts to switching all the following in A:

- $\top$ with $\bot$
- variables p_i with their negations $\neg p_i$
- $\wedge$ with $\vee$.

We will see that *conjunctive normal form* (CNF) and *disjunctive normal form* (DNF) are also important representations of propositional formulas. A formula is in conjunctive normal form if it can be written as conjunction of disjunctions of literals, in other words, if it can be written as a big "and" of "or" expressions:

$$\bigwedge_{i<n} \left(\bigvee_{j<m_i} \pm \ell_j \right).$$

where each ℓ_j is a literal. Here is an example:

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

We can think of $\bot$ as the empty disjunction (because a disjunction is true only when one of the disjuncts is true) and we can think of $\top$ as the empty conjunction (because a conjunction is true when all of its conjuncts are true, which happens trivially when there aren't any).

Dually, a formula is in disjunctive normal form if it is an "or" and "and" expressions:

$$\bigvee_{i<n} \left(\bigwedge_{j<m_i} \pm \ell_j \right).$$

If you switch $\wedge$ and $\vee$ in the previous example, you have a formula in disjunctive normal form.

It is pretty clear that if you take the conjunction of two formulas in CNF the result is a CNF formula (modulo associating parentheses), and, similarly, the disjunction of two formulas in DNF is again DNF. The following is less obvious:

Lemma

The disjunction of two CNF formulas is equivalent to a CNF formula, and dually for DNF formulas.

Proof

For the first claim, we use the equivalence $A \vee (B \wedge C) \equiv (A \wedge B) \vee (A \wedge C)$. By induction on n, we have that for every sequence of formulas $B_0, \ldots, B_{n-1}$ we have $A \vee \bigwedge_{i<n} B_i \equiv \bigwedge_{i<n} (A \vee B_i)$. Then by induction on n' we have $\bigwedge_{i'<n'} A'_i \vee \bigwedge_{i<n} B_i \equiv \bigwedge_{i'<n'} \bigwedge_{i<n} (A'_i \vee B_i)$. Since each $A_{i'}$ and each B_i is a disjunction of literals, this yields the result.

The second claim is proved similarly, switching $\wedge$ and $\vee$.

Proposition

Every propositional formula is equivalent to one in conjunctive normal form, and also to one in disjunctive normal form.

Proof

Since we already know that every formula is equivalent to one in negation normal form, we can use induction on that set of formulas. The claim is clearly true of $\top$, $\bot$, p_i, and $\neg p_i$. By the previous lemma, whenever it is true of A and B, it is also true of $A \wedge B$ and $A \vee B$.

In contrast to putting formulas in negation normal form, the exercises below show that the smallest CNF or DNF equivalent of a formula A can be exponentially longer than A.

We will see that conjunctive normal form is commonly used in automated reasoning. Notice that if a disjunction of literals contains a duplicated literal, deleting the duplicate results in an equivalent formula. We can similarly delete any occurrence of $\bot$. A disjunction of literals is called a *clause*. Since the order of the disjuncts and repetitions don't matter, we generally identify clauses with the corresponding set of literals; for example, the clause $p \vee \neg q \vee r$ is associated with the set $\{p, \neg q, r\}$. If a clause contains a pair p_i and $\neg p_i$, or if it contains $\top$, it is equivalent to $\top$. If Γ is a set of clauses, we think of Γ as saying that all the clauses in Γ are true. With this identification, every formula in conjunctive normal form is equivalent to a set of clauses. An empty clause corresponds to $\bot$, and an empty set of clauses corresponds to $\top$.

The dual notion to a clause is a conjunction like $\neg p \wedge q \wedge \neg r$. If each variable occurs at most once (either positively or negatively), we can think of this as a *partial truth assignment*. In this example, any truth assignment that satisfies the formula has to set p false, q true, and r false.

4.6 Exercises

1. Prove that if A is a subformula of B and B is a subformula of C then A is a subformula of C. (Hint: prove by induction on C that for every $B \in \mathit{subformulas}(C)$, every subformula of B is a subformula of C.)
2. Prove that for every A, B, and p, $\mathit{depth}(A[B/p]) \leq \mathit{depth}(A) + \mathit{depth}(B)$.
3. Prove that A and B are logically equivalent if and only if the formula $A \leftrightarrow B$ is valid.
4. Use algebraic calculations to show that all of the following are tautologies:
   - $((A \wedge \neg B) \vee B) \leftrightarrow (A \vee B)$
   - $(A \to \neg A) \to \neg A$
   - $(A \to B) \leftrightarrow (\neg B \to \neg A)$
   - $A \to (B \to A \wedge B)$
5. The *Sheffer stroke* $A \mid B$, also known as "nand," says that A and B are not both true. Show that $\{\mid\}$ is a complete set of connectives. Do the same for "nor," that is, the binary connective that holds if neither A nor B is true.
6. Show that $\{\wedge, \neg\}$ and $\{\to, \bot\}$ are complete sets of connectives.
7. Show that $\{\to, \vee, \wedge\}$ is not a complete set of connectives. Conclude that $\{\to, \vee, \wedge, \leftrightarrow, \top\}$ is not a complete set of connectives.
8. Show that $\{\bot, \leftrightarrow\}$ is not a complete set of connectives. Conclude that $\{\bot, \top, \neg, \leftrightarrow, \oplus\}$ is not complete. Here $A \oplus B$ is the "exclusive or," which is to say, $A \oplus B$ is true if one of A or B is true but not both.
9. Using the property $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ and the dual statement with $\wedge$ and $\vee$ switched, put $(p_1 \wedge p_2) \vee (q_1 \wedge q_2) \vee (r_1 \wedge r_2)$ in conjunctive normal form.

10. The boolean function *parity*$(x_0, x_1, \ldots, x_{n-1})$ holds if and only if an odd number of the x_i s are true. It is represented by the formula $p_0 \oplus p_1 \oplus \cdots \oplus p_{n-1}$. Show that any CNF formula representing the parity function has to have at least 2^n clauses.

CHAPTER

FIVE

IMPLEMENTING PROPOSITIONAL LOGIC

5.1 Propositional formulas

We have seen that the set of propositional formulas can be defined inductively, and we have seen that Lean makes it easy to specify inductively defined types. It's a match made in heaven! Here is the definition of the type of propositional formulas that we will use in this course:

```
namespace hidden

inductive PropForm
  | tr     : PropForm
  | fls    : PropForm
  | var    : String → PropForm
  | conj   : PropForm → PropForm → PropForm
  | disj   : PropForm → PropForm → PropForm
  | impl   : PropForm → PropForm → PropForm
  | neg    : PropForm → PropForm
  | biImpl : PropForm → PropForm → PropForm
  deriving Repr, DecidableEq

end hidden

#print PropForm

open PropForm

#check (impl (conj (var "p") (var "q")) (var "r"))
```

You can find this example in the file *implementing_propositional_logic/examples.lean* in the *User* folder of the course repository.

The command *import LAMR.Util.Propositional* at the top of the file imports the part of the library with functions that we provide for you to deal with propositional logic. We will often put a copy of a definition from the library in an examples file so you can experiment with it. Here we have put it in a namespace called *hidden* so that our copy's full name is *hidden.PropForm*, which won't conflict with the one in the library. Outside the *hidden* namespace, the command *#print PropForm* refers to the real one, that is, the one in the library. The command *open PropForm* means that we can write, for example, *tr* for the first constructor instead of *PropForm.tr*. Try writing some propositional formulas of your own. There should be squiggly blue lines under the the *#print* and *#check* commands in VSCode, indicating that there is Lean output associated with these. You can see it by hovering over the commands, or by moving the caret to the command and checking the *Lean infoview* window.

The phrase *deriving Repr, DecidableEq* tells Lean to automatically define functions to be used to test equality of two expressions of type *PropForm* and to display the result of an *#eval*. We'll generally leave these out of the display from now on. You an always use *#check* and *#print* to learn more about a definition in the library. If you hold down *ctrl* and click on an identifier, the VSCode Lean extension will take you to the definition in the library. Simply holding down *ctrl* and hovering over it will show you the definition in a pop-up window. Try taking a look at the definition of *PropForm* in the library.

Writing propositional formulas using constructors can be a pain in the neck. In the library, we have used Lean's mechanisms for defining new syntax to implement nicer syntax.

```
#check prop!{p ∧ q → (r ∨ ¬ p) → q}
#check prop!{p ∧ q ∧ r → p}

def propExample := prop!{p ∧ q → r ∧ p ∨ ¬ s1 → s2 }

#print propExample
#eval propExample

#eval toString propExample
```

You can get the symbols by typing *\and*, *\to*, *\or*, *\not*, and *\iff* in VS Code. And, in general, when you see a symbol in VSCode, hovering over it with the mouse shows you how to type it. Once again, try typing some examples of your own. The library defines the function *PropForm.toString* that produces a more readable version of a propositional formula, one that, when inserted within the *prop!{...}* brackets, should produce the same result.

Because *PropForm* is inductively defined, we can easily define functions using structural recursion.

```
namespace PropForm

def complexity : PropForm → Nat
  | var _ => 0
  | tr => 0
  | fls => 0
  | neg A => complexity A + 1
  | conj A B => complexity A + complexity B + 1
  | disj A B => complexity A + complexity B + 1
  | impl A B => complexity A + complexity B + 1
  | biImpl A B => complexity A + complexity B + 1

def depth : PropForm → Nat
  | var _ => 0
  | tr => 0
  | fls => 0
  | neg A => depth A + 1
  | conj A B => Nat.max (depth A) (depth B) + 1
  | disj A B => Nat.max (depth A) (depth B) + 1
  | impl A B => Nat.max (depth A) (depth B) + 1
  | biImpl A B => Nat.max (depth A) (depth B) + 1

def vars : PropForm → List String
  | var s => [s]
  | tr => []
  | fls => []
  | neg A => vars A
  | conj A B => (vars A).union (vars B)
  | disj A B => (vars A).union (vars B)
  | impl A B => (vars A).union (vars B)
  | biImpl A B => (vars A).union (vars B)

#eval complexity propExample
#eval depth propExample
```

(continues on next page)

(continued from previous page)

```
#eval vars propExample

end PropForm

#eval PropForm.complexity propExample
#eval propExample.complexity
```

The function *List.union* returns concatenation of the two lists with duplicates removed, assuming that the original two lists had no duplicate elements.

5.2 Semantics

The course library defines the type *PropAssignment* to be *String* → *Bool*. If *v* has type *PropAssignment*, you should think of the expression *v s* as assigning a truth value to the variable named *s*. The following function then evaluates the truth value of any propositional formula under assignment *v*:

```
def PropForm.eval (v : PropAssignment) : PropForm → Bool
  | var s => v s
  | tr => true
  | fls => false
  | neg A => !(eval v A)
  | conj A B => (eval v A) && (eval v B)
  | disj A B => (eval v A) || (eval v B)
  | impl A B => !(eval v A) || (eval v B)
  | biImpl A B => (!(eval v A) || (eval v B)) && (!(eval v B) || (eval v A))

-- try it out
#eval let v := fun s => if s ∈ ["p", "q", "r"] then true else false
      propExample.eval v
```

The example at the end defines *v* to be the assignment that assigns the value *true* to the strings *"p"*, *"q"*, and *"r"* and false to all the others. This is a reasonably convenient way to describe truth assignments manually, so the library provides a function *mkPropAssignment* and notation *propassign!{...}* to support that.

```
#check PropAssignment.mk ["p", "q", "r"]
#check propassign!{p, q, r}

#eval propExample.eval propassign!{p, q, r}
```

You should think about how the next function manages to compute a list of all the sublists of a given list. It is analogous to the power set operation in set theory.

```
def allSublists : List α → List (List α)
  | [] => [[]]
  | (a :: as) =>
      let recval := allSublists as
      recval.map (a :: .) ++ recval

#eval allSublists propExample.vars
```

With that in hand, here is a function that computes the truth table of a propositional formula. The value of *truthTable A* is a list of pairs: the first element of the pair is the list of *true*/*false* values assigned to the elements of *vars A*, and the second element is the truth value of *A* under that assignment.

```
def truthTable (A : PropForm) : List (List Bool × Bool) :=
  let vars := A.vars
  let assignments := (allSublists vars).map PropAssignment.mk
  let evalLine := fun v : PropAssignment => (vars.map v, A.eval v)
  assignments.map evalLine

#eval truthTable propExample
```

We can now use the list operation *List.all* to test whether a formula is valid, and we can use *List.some* to test whether it is satisfiable.

```
def PropForm.isValid (A : PropForm) : Bool := List.all (truthTable A) Prod.snd
def PropForm.isSat(A : PropForm) : Bool := List.any (truthTable A) Prod.snd

#eval propExample.isValid
#eval propExample.isSat
```

5.3 Normal Forms

The library defines an inductive type of negation-normal form formulas:

```
inductive Lit
  | tr  : Lit
  | fls : Lit
  | pos : String → Lit
  | neg : String → Lit

inductive NnfForm :=
  | lit  (l : Lit)       : NnfForm
  | conj (p q : NnfForm) : NnfForm
  | disj (p q : NnfForm) : NnfForm
```

It is then straightforward to define the negation operation for formulas in negation normal form, and a translation from propositional formulas to formulas in negation normal form.

```
def Lit.negate : Lit → Lit
  | tr   => fls
  | fls  => tr
  | pos s => neg s
  | neg s => pos s

def NnfForm.neg : NnfForm → NnfForm
  | lit l    => lit l.negate
  | conj p q => disj (neg p) (neg q)
  | disj p q => conj (neg p) (neg q)

namespace PropForm

def toNnfForm : PropForm → NnfForm
  | tr         => NnfForm.lit Lit.tr
  | fls        => NnfForm.lit Lit.fls
  | var n      => NnfForm.lit (Lit.pos n)
  | neg p      => p.toNnfForm.neg
  | conj p q   => NnfForm.conj p.toNnfForm q.toNnfForm
```

(continues on next page)

(continued from previous page)

```
  | disj p q   => NnfForm.disj p.toNnfForm q.toNnfForm
  | impl p q   => NnfForm.disj p.toNnfForm.neg q.toNnfForm
  | biImpl p q => NnfForm.conj (NnfForm.disj p.toNnfForm.neg q.toNnfForm)
                              (NnfForm.disj q.toNnfForm.neg p.toNnfForm)

end PropForm
```

Putting the first in the namespace *NnfForm* has the effect that given *A : NnfForm*, we can write *A.neg* instead of *NnfForm.neg A*. Similarly, putting the second definition in the namespace *PropForm* means we can write *A.toNnfForm* to put a propositional formula in negation normal form.

We can try them out on the example defined above:

```
#eval propExample.toNnfForm
#eval toString propExample.toNnfForm
```

To handle conjunctive normal form, the library defines a type *Lit* of literals. A *Clause* is then a list of literals, and a *CnfForm* is a list of clauses.

```
def Clause := List Lit

def CnfForm := List Clause
```

As usual, you can use *#check* and *#print* to find information about them, and ctrl-click to see the definitions in the library. Since, as usual, defining things using constructors can be annoying, the library defines syntax for writing expressions of these types.

```
def exLit0 := lit!{ p }
def exLit1 := lit!{ -q }

#print exLit0
#print exLit1

def exClause0 := clause!{ p }
def exClause1 := clause!{ p -q r }
def exClause2 := clause!{ r -s }

#print exClause0
#print exClause1
#print exClause2

def exCnf0 := cnf!{
  p,
  -p q -r,
  -p q
}

def exCnf1 := cnf!{
  p -q,
  p q,
  -p -r,
  -p r
}

def exCnf2 := cnf!{
  p q,
```

(continues on next page)

(continued from previous page)

```
  -p,
  -q
}

#print exCnf0
#print exCnf1
#print exCnf2

#eval toString exClause1
#eval toString exCnf2
```

Let us now consider what is needed to put an arbitrary propositional formula in conjunctive normal form. In Section 4.5, we saw that the key is to show that the disjunction of two CNF formulas is again CNF. Lean's library has a function *List.insert*, which adds an element to a list; if the element already appears in the list, it does nothing. It has a function *List.union* that will form the union of two lists; if the original two lists have no duplicates, the union won't either. Finally, we have a function *List.Union* which takes the union of a list of lists. Since clauses are lists, we can use them on clauses:

```
#eval List.insert lit!{ r } exClause0

#eval exClause0.union exClause1

#eval List.Union [exClause0, exClause1, exClause2]
```

We can now take the disjunction of a single clause and a CNF formula by taking the union of the clause with each element of the CNF formula. We can implement that with the function *List.map*:

```
#eval exCnf1.map exClause0.union
```

This applied the function "take the union with *exClause0*" to each element of *exCnf1*, and returns the resulting list. We can now define the disjunction of two CNF formulas by taking all the clauses in the first, taking the disjunction of each clause with the second CNF, and then taking the union of all of those, corresponding to the conjunctions of the CNFs. Here is the library definition, and an example:

```
def CnfForm.disj (cnf1 cnf2 : CnfForm) : CnfForm :=
(cnf1.map (fun cls => cnf2.map cls.union)).Union

#eval cnf!{p, q, u -v}.disj cnf!{r1 r2, s1 s2, t1 t2 t3}
#eval toString $ cnf!{p, q, u -v}.disj cnf!{r1 r2, s1 s2, t1 t2 t3}
```

Functional programmers like this sort of definition; it's short, clever, and inscrutable. You should think about defining the disjunction of two CNF formulas by hand, using recursions over clauses and CNF formulas. Your solution will most likely reconstruct the effect of the instance *map* and *Union* in the library definition, and that will help you understand why they make sense.

In any case, with this in hand, it is easy to define the translation from negation normal form formulas and arbitrary propositional formulas to CNF.

```
def NnfForm.toCnfForm : NnfForm → CnfForm
  | NnfForm.lit (Lit.pos s) => [ [Lit.pos s] ]
  | NnfForm.lit (Lit.neg s) => [ [Lit.neg s] ]
  | NnfForm.lit Lit.tr      => []
  | NnfForm.lit Lit.fls     => [ [] ]
  | NnfForm.conj A B        => A.toCnfForm.conj B.toCnfForm
  | NnfForm.disj A B        => A.toCnfForm.disj B.toCnfForm

def PropForm.toCnfForm (A : PropForm) : CnfForm := A.toNnfForm.toCnfForm
```

We can try them out:

```
#eval propExample.toCnfForm

#eval prop!{(p1 ∧ p2) ∨ (q1 ∧ q2)}.toCnfForm.toString

#eval prop!{(p1 ∧ p2) ∨ (q1 ∧ q2) ∨ (r1 ∧ r2) ∨ (s1 ∧ s2)}.toCnfForm.toString
```

5.4 Exercises

1. Write a Lean function that, given any element of *PropForm*, outputs a list of all the subformulas.
2. Write a Lean function that, given a list of propositional formulas and another propositional formula, determines whether the second is a logical consequence of the first.
3. Write a Lean function that, given a clause, tests whether any literal *Lit.pos p* appears together with its negation *Lit.neg p*. Write another Lean function that, given a formula in conjunctive normal form, deletes all these clauses.

CHAPTER

SIX

USING SAT SOLVERS

CHAPTER

SEVEN

DEDUCTION AND SEARCH

CHAPTER

EIGHT

TERMS AND FORMULAS

CHAPTER

NINE

IMPLEMENTING TERMS AND FORMULAS

CHAPTER TEN

USING SMT SOLVERS

CHAPTER

ELEVEN

FIRST-ORDER LOGIC

CHAPTER

TWELVE

IMPLEMENTING FIRST-ORDER LOGIC

CHAPTER THIRTEEN

USING FIRST-ORDER THEOREM PROVERS

CHAPTER

FOURTEEN

SIMPLE TYPE THEORY