Software Foundations

Benjamin C. Pierce

Arthur Azevedo de Amorim

Chris Casinghino

Marco Gaboardi

Michael Greenberg

Cătălin Hriţcu

Vilhelm Sjöberg

Brent Yorgey

with Loris D'Antoni, Andrew W. Appel, Arthur Charguéraud, Anthony Cowley, Jeffrey Foster, Dmitri Garbuzov, Michael Hicks, Ranjit Jhala, Greg Morrisett, Jennifer Paykin, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, Andrew Tolmach, Stephanie Weirich and Steve Zdancewic.

Idris translation by Eric Bailey, Alex Gryzlov and Erlend Hamberg.

Contents

CHAPTER 1

Preface

1. Welcome

This electronic book is a course on *Software Foundations*, the mathematical underpinnings of reliable software. Topics include basic concepts of logic, computer-assisted theorem proving, the Idris programming language, functional programming, operational semantics, Hoare logic, and static type systems. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful.

The principal novelty of the course is that it is one hundred percent formalized and machine-checked: the entire text is Literate Idris. It is intended to be read alongside an interactive session with Idris. All the details in the text are fully formalized in Idris, and the exercises are designed to be worked using Idris.

The files are organized into a sequence of core chapters, covering about one semester's worth of material and organized into a coherent linear narrative, plus a number of "appendices" covering additional topics. All the core chapters are suitable for both upper-level undergraduate and graduate students.

2. Overview

Building reliable software is hard. The scale and complexity of modern systems, the number of people involved in building them, and the range of demands placed on them render it extremely difficult to build software that is even more-or-less correct, much less 100% correct. At the same time, the increasing degree to which information processing is woven into every aspect of society continually amplifies the cost of bugs and insecurities.

Computer scientists and software engineers have responded to these challenges by developing a whole host of techniques for improving software reliability, ranging from recommendations about managing software projects and organizing programming teams (e.g., extreme programming) to design philosophies for libraries (e.g., model-view-controller, publish-subscribe, etc.) and programming languages (e.g., object-oriented programming, aspect-oriented programming, functional programming, …) to mathematical techniques for specifying and reasoning about properties of software and tools for helping validate these properties.

The present course is focused on this last set of techniques. The text weaves together five conceptual threads:

1. basic tools from *logic* for making and justifying precise claims about programs;
2. the use of *proof assistants* to construct rigorous logical arguments;
3. the idea of *functional programming*, both as a method of programming that simplifies reasoning about programs and as a bridge between programming and logic;
4. formal techniques for *reasoning about the properties of specific programs* (e.g., the fact that a sorting function or a compiler obeys some formal specification); and
5. the use of *type systems* for establishing well-behavedness guarantees for *all* programs in a given programming language (e.g., the fact that well-typed Java programs cannot be subverted at runtime).

Each of these topics is easily rich enough to fill a whole course in its own right, so tackling all of them together naturally means that much will be left unsaid. Nevertheless, we hope readers will find that the themes illuminate and amplify each other and that bringing them together creates a foundation from which it will be easy to dig into any of them more deeply. Some suggestions for further reading can be found in the [Postscript] chapter. Bibliographic information for all cited works can be found in the [Bib] chapter.

2.1. Logic. Logic is the field of study whose subject matter is *proofs* – unassailable arguments for the truth of particular propositions. Volumes have been written about the central role of logic in computer science. Manna and Waldinger called it "the calculus of computer science," while Halpern et al.'s paper *On the Unusual Effectiveness of Logic in Computer Science* catalogs scores of ways in which logic offers critical tools and insights. Indeed, they observe that "As a matter of fact, logic has turned out to be significantly more effective in computer science than it has been in mathematics. This is quite remarkable, especially since much of the impetus for the development of logic during the past one hundred years came from mathematics."

In particular, the fundamental notion of inductive proofs is ubiquitous in all of computer science. You have surely seen them before, in contexts from discrete math to analysis of algorithms, but in this course we will examine them much more deeply than you have probably done so far.

2.2. Proof Assistants. The flow of ideas between logic and computer science has not been in just one direction: CS has also made important contributions to logic. One of these has been the development of software tools for helping construct proofs of logical propositions. These tools fall into two broad categories:

- *Automated theorem provers* provide "push-button" operation: you give them a proposition and they return either *true*, *false*, or *ran out of time*.

Although their capabilities are limited to fairly specific sorts of reasoning, they have matured tremendously in recent years and are used now in a huge variety of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.

- *Proof assistants* are hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, Coq, and Idris among many others.

This course is based around Coq, a proof assistant that has been under development, mostly in France, since 1983 and that in recent years has attracted a large community of users in both research and industry. Coq provides a rich environment for interactive development of machine-checked formal reasoning. The kernel of the Coq system is a simple proof-checker, which guarantees that only correct deduction steps are performed. On top of this kernel, the Coq environment provides high-level facilities for proof development, including powerful tactics for constructing complex proofs semi-automatically, and a large library of common definitions and lemmas.

Coq has been a critical enabler for a huge variety of work across computer science and mathematics:

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C.

- As an *environment for developing formally certified software*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms.

- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot project at Harvard embedded "relational Hoare reasoning" (an extension of the *Hoare Logic* we will see later in this course) in Coq.

- As a *proof assistant for higher-order logic*, it has been used to validate a number of important results in mathematics. For example, its ability to include complex computations inside proofs made it possible to develop the first formally verified proof of the 4-color theorem. This proof had previously been controversial among mathematicians because part of it included checking a large number of configurations using a program. In the Coq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led

to a Coq formalization of the Feit-Thompson Theorem – the first major step in the classification of finite simple groups.

By the way, in case you're wondering about the name, here's what the official Coq web site says: "Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, 'coq' means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based." The rooster is also the national symbol of France, and C-o-q are the first three letters of the name of Thierry Coquand, one of Coq's early developers.

2.3. Functional Programming. The term *functional programming* refers both to a collection of programming idioms that can be used in almost any programming language and to a family of programming languages designed to emphasize these idioms, including Haskell, OCaml, Standard ML, F#, Scala, Scheme, Racket, Common Lisp, Clojure, Erlang, and Coq.

Functional programming has been developed over many decades – indeed, its roots go back to Church's lambda-calculus, which was invented in the 1930s, before there were even any computers! But since the early '90s it has enjoyed a surge of interest among industrial engineers and language designers, playing a key role in high-value systems at companies like Jane St. Capital, Microsoft, Facebook, and Ericsson.

The most basic tenet of functional programming is that, as much as possible, computation should be *pure*, in the sense that the only effect of execution should be to produce a result: the computation should be free from *side effects* such as I/O, assignments to mutable variables, redirecting pointers, etc. For example, whereas an *imperative* sorting function might take a list of numbers and rearrange its pointers to put the list in order, a pure sorting function would take the original list and return a *new* list containing the same numbers in sorted order.

One significant benefit of this style of programming is that it makes programs easier to understand and reason about. If every operation on a data structure yields a new data structure, leaving the old one intact, then there is no need to worry about how that structure is being shared and whether a change by one part of the program might break an invariant that another part of the program relies on. These considerations are particularly critical in concurrent programs, where every piece of mutable state that is shared between threads is a potential source of pernicious bugs. Indeed, a large part of the recent interest in functional programming in industry is due to its simpler behavior in the presence of concurrency.

Another reason for the current excitement about functional programming is related to the first: functional programs are often much easier to parallelize than their imperative counterparts. If running a computation has no effect other than producing a result, then it does not matter *where* it is run. Similarly, if a data structure is never modified destructively, then it can be copied freely, across cores or across the network. Indeed, the "Map-Reduce" idiom, which lies at the heart of massively distributed query processors like Hadoop and is used by Google to index the entire web is a classic example of functional programming.

For this course, functional programming has yet another significant attraction: it serves as a bridge between logic and computer science. Indeed, Coq itself can be viewed as a combination of a small but extremely expressive functional programming language plus with a set of tools for stating and proving logical assertions. Moreover, when we come to look more closely, we find that these two sides of Coq are actually aspects of the very same underlying machinery – i.e., *proofs are programs*.

2.4. Program Verification. Approximately the first third of the book is devoted to developing the conceptual framework of logic and functional programming and gaining enough fluency with Coq to use it for modeling and reasoning about nontrivial artifacts. From this point on, we increasingly turn our attention to two broad topics of critical importance to the enterprise of building reliable software (and hardware): techniques for proving specific properties of particular *programs* and for proving general properties of whole programming *languages*.

For both of these, the first thing we need is a way of representing programs as mathematical objects, so we can talk about them precisely, together with ways of describing their behavior in terms of mathematical functions or relations. Our tools for these tasks are *abstract syntax* and *operational semantics*, a method of specifying programming languages by writing abstract interpreters. At the beginning, we work with operational semantics in the so-called "big-step" style, which leads to somewhat simpler and more readable definitions when it is applicable. Later on, we switch to a more detailed "small-step" style, which helps make some useful distinctions between different sorts of "nonterminating" program behaviors and is applicable to a broader range of language features, including concurrency.

The first programming language we consider in detail is *Imp*, a tiny toy language capturing the core features of conventional imperative programming: variables, assignment, conditionals, and loops. We study two different ways of reasoning about the properties of Imp programs.

First, we consider what it means to say that two Imp programs are *equivalent* in the intuitive sense that they yield the same behavior when started in any initial memory state. This notion of equivalence then becomes a criterion for judging the correctness of *metaprograms* – programs that manipulate other programs, such as compilers and optimizers. We build a simple optimizer for Imp and prove that it is correct.

Second, we develop a methodology for proving that particular Imp programs satisfy formal specifications of their behavior. We introduce the notion of *Hoare triples* – Imp programs annotated with pre- and post-conditions describing what should be true about the memory in which they are started and what they promise to make true about the memory in which they terminate – and the reasoning principles of *Hoare Logic*, a "domain-specific logic" specialized for convenient compositional reasoning about imperative programs, with concepts like "loop invariant" built in.

This part of the course is intended to give readers a taste of the key ideas and mathematical tools used in a wide variety of real-world software and hardware verification tasks.

2.5. Type Systems. Our final major topic, covering approximately the last third of the course, is *type systems*, a powerful set of tools for establishing properties of *all* programs in a given language.

Type systems are the best established and most popular example of a highly successful class of formal verification techniques known as *lightweight formal methods*. These are reasoning techniques of modest power – modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underlying theories. Other examples of lightweight formal methods include hardware and software model checkers, contract checkers, and run-time property monitoring techniques for detecting when some component of a system is not behaving according to specification.

This topic brings us full circle: the language whose properties we study in this part, the *simply typed lambda-calculus*, is essentially a simplified model of the core of Coq itself!

2.6. Further Reading. This text is intended to be self contained, but readers looking for a deeper treatment of a particular topic will find suggestions for further reading in the [Postscript] chapter.

3. Practicalities

3.1. Chapter Dependencies. A diagram of the dependencies between chapters and some suggested paths through the material can be found in the file [deps.html].

3.2. System Requirements. Coq runs on Windows, Linux, and OS X. You will need:

- A current installation of Coq, available from the Coq home page. Everything should work with version 8.4. (Version 8.5 will *not* work, due to a few incompatible changes in Coq between 8.4 and 8.5.)
- An IDE for interacting with Coq. Currently, there are two choices:
  - Proof General is an Emacs-based IDE. It tends to be preferred by users who are already comfortable with Emacs. It requires a separate installation (google "Proof General").
  - CoqIDE is a simpler stand-alone IDE. It is distributed with Coq, so it should "just work" once you have Coq installed. It can also be compiled from scratch, but on some platforms this may involve installing additional packages for GUI libraries and such.

3.3. Exercises. Each chapter includes numerous exercises. Each is marked with a "star rating," which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Also, some exercises are marked "advanced", and some are marked "optional." Doing just the non-optional, non-advanced exercises should provide good coverage of the core material. Optional exercises provide a bit of extra practice with key concepts and introduce secondary themes that may be of interest to some readers. Advanced exercises are for readers who want an extra challenge (and, in return, a deeper contact with the material).

Please do not post solutions to the exercises in any public place: Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. The authors especially request that readers not post solutions to the exercises anyplace where they can be found by search engines.

3.4. Downloading the Coq Files. A tar file containing the full sources for the "release version" of these notes (as a collection of Coq scripts and HTML files) is available here:

```
http://www.cis.upenn.edu/~bcpierce/sf
```

If you are using the notes as part of a class, you may be given access to a locally extended version of the files, which you should use instead of the release version.

4. Note for Instructors

If you intend to use these materials in your own course, you will undoubtedly find things you'd like to change, improve, or add. Your contributions are welcome!

To keep the legalities of the situation clean and to have a single point of responsibility in case the need should ever arise to adjust the license terms, sublicense, etc., we ask all contributors (i.e., everyone with access to the developers' repository) to assign copyright in their contributions to the appropriate "author of record," as follows:

```
I hereby assign copyright in my past and future contributions
to the Software Foundations project to the Author of Record of
each volume or component, to be licensed under the same terms
as the rest of Software Foundations.  I understand that, at
present, the Authors of Record are as follows: For Volumes 1
```

```
and 2, known until 2016 as "Software Foundations" and from
2016 as (respectively) "Logical Foundations" and "Programming
Foundations," the Author of Record is Benjamin Pierce.  For
Volume 3, "Verified Functional Algorithms", the Author of
Record is Andrew W. Appel. For components outside of
designated Volumes (e.g., typesetting and grading tools and
other software infrastructure), the Author of Record is
Benjamin Pierce.
```

To get started, please send an email to Benjamin Pierce, describing yourself and how you plan to use the materials and including 1. the above copyright transfer text and 2. the result of doing "htpasswd -s -n NAME" where NAME is your preferred user name.

We'll set you up with access to the subversion repository and developers' mailing lists. In the repository you'll find a file [INSTRUCTORS] with further instructions.

5. Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can now be enjoyed in Japanese at [http://proofcafe.org/sf]. A Chinese translation is underway.

CHAPTER 2

Basics

REMINDER:

```
########################################################
###  PLEASE DO NOT DISTRIBUTE SOLUTIONS PUBLICLY  ###
########################################################

                (See the Preface for why.)
```

```
||| Basics: Functional Programming in Idris
module Basics

%access public export
```

`postulate` is Idris's "escape hatch" that says accept this definition without proof. Instead of using it to mark the holes, similar to Coq's `Admitted`, we use Idris's holes directly. In practice, holes (and `postulate`) are useful when you're incrementally developing large proofs.

1. Introduction

The functional programming style brings programming closer to simple, everyday mathematics: If a procedure or method has no side effects, then (ignoring efficiency) all we need to understand about it is how it maps inputs to outputs – that is, we can think of it as just a concrete method for computing a mathematical function. This is one sense of the word "functional" in "functional programming." The direct connection between programs and simple mathematical objects supports both formal correctness proofs and sound informal reasoning about program behavior.

The other sense in which functional programming is "functional" is that it emphasizes the use of functions (or methods) as *first-class* values – i.e., values that can be passed as arguments to other functions, returned as results, included in data structures, etc. The recognition that functions can be treated as data in this way enables a host of useful and powerful idioms.

Other common features of functional languages include *algebraic data types* and *pattern matching*, which make it easy to construct and manipulate rich data structures, and sophisticated *polymorphic type systems* supporting abstraction and code reuse. Idris shares all of these features.

The first half of this chapter introduces the most essential elements of Idris's functional programming language. The second half introduces some basic *tactics* that can be used to prove simple properties of Idris programs.

2. Enumerated Types

One unusual aspect of Idris, similar to Coq, is that its set of built-in features (see the `base` package in the Idris distribution) is *extremely* small. For example, instead of providing the usual palette of atomic data types (booleans, integers, strings, etc.), Idris offers a powerful mechanism for defining new data types from scratch, from which all these familiar types arise as instances.

Naturally, the Idris distribution comes with extensive standard libraries providing definitions of booleans, numbers, and many common data structures like lists and hash tables (see the `prelude` and `contrib` packages), as well as the means to write type-safe effectful code (see the `effects` package) and `pruvlioj`, a toolkit for proof automation and program construction. But there is nothing magic or primitive about these library definitions. To illustrate this, we will explicitly recapitulate all the definitions we need in this course, rather than just getting them implicitly from the library.

To see how this definition mechanism works, let's start with a very simple example.

2.1. Days of the Week. The following declaration tells Idris that we are defining a new set of data values – a *type*.

```
namespace Days

  ||| Days of the week.
  data Day = ||| `Monday` is a `Day`.
             Monday
           | ||| `Tuesday` is a `Day`.
             Tuesday
           | ||| `Wednesday` is a `Day`.
             Wednesday
           | ||| `Thursday` is a `Day`.
             Thursday
           | ||| `Friday` is a `Day`.
             Friday
           | ||| `Saturday` is a `Day`.
             Saturday
           | ||| `Sunday` is a `Day`.
             Sunday
```

The type is called `Day`, and its members are `Monday`, `Tuesday`, etc. The right hand side of the definition can be read "`Monday` is a `Day`, `Tuesday` is a `Day`, etc."

Using the `%name` directive, we can tell Idris how to choose default variable names for a particular *type*.

```
%name Day day, day1, day2
```

Now, if Idris needs to choose a name for a variable of type `Day`, it will choose `day` by default, followed by `day1` and `day2` if any of its predecessors are already in scope.

Having defined `Day`, we can write functions that operate on days.

Type the following:

```
nextWeekday : Day → Day
```

Then with the point on `nextWeekday`, call `idris-add-clause`.

```
nextWeekday : Day → Day
nextWeekday day = ?nextWeekday_rhs
```

With the point on `day`, call `idris-case-split`.

```
nextWeekday : Day → Day
nextWeekday Monday = ?nextWeekday_rhs_1
nextWeekday Tuesday = ?nextWeekday_rhs_2
nextWeekday Wednesday = ?nextWeekday_rhs_3
nextWeekday Thursday = ?nextWeekday_rhs_4
nextWeekday Friday = ?nextWeekday_rhs_5
nextWeekday Saturday = ?nextWeekday_rhs_6
nextWeekday Sunday = ?nextWeekday_rhs_7
```

Fill in the proper `Day` constructors and align whitespace as you like.

```
||| Determine the next weekday after a day.
nextWeekday : Day → Day
nextWeekday Monday    = Tuesday
nextWeekday Tuesday   = Wednesday
nextWeekday Wednesday = Thursday
nextWeekday Thursday  = Friday
nextWeekday Friday    = Monday
nextWeekday Saturday  = Monday
nextWeekday Sunday    = Monday
```

Call `idris-load-file` to load the `Basics` module with the finished `nextWeekday` definition.

Having defined a function, we should check that it works on some examples. There are actually three different ways to do this in Idris.

First, we can evaluate an expression involving `nextWeekday` in a REPL.

```
λΠ> nextWeekday Friday
Monday : Day

λΠ> nextWeekday (nextWeekday Saturday)
Tuesday : Day
```

Mention other editors? Discuss idris-mode?

We show Idris's responses in comments, but, if you have a computer handy, this would be an excellent moment to fire up the Idris interpreter under your favorite Idris-friendly text editor – such as Emacs or Vim – and try this for and try this for yourself. Load this file, `Basics.lidr` from the book's accompanying Idris sources, find the above example, submit it to the Idris REPL, and observe the result.

Second, we can record what we *expect* the result to be in the form of a proof.

```
|||  The second weekday after `Saturday` is `Tuesday`.
testNextWeekday :
  (nextWeekday (nextWeekday Saturday)) = Tuesday
```

This declaration does two things: it makes an assertion (that the second weekday after `Saturday` is `Tuesday`) and it gives the assertion a name that can be used to refer to it later.

Having made the assertion, we can also ask Idris to verify it, like this:

```
testNextWeekday = Refl
```

Edit this

The details are not important for now (we'll come back to them in a bit), but essentially this can be read as "The assertion we've just made can be proved by observing that both sides of the equality evaluate to the same thing, after some simplification."

(For simple proofs like this, you can call `idris-add-clause` with the point on the name (`testNextWeekday`) in the type signature and then call `idris-proof-search` (`M-RET p`) with the point on the resultant hole to have Idris solve the proof for you.)

Verify the "main uses" claim.

Third, we can ask Idris to *generate*, from our definition, a program in some other, more conventional, programming (C, JavaScript and Node are bundled with Idris) with a high-performance compiler. This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. Indeed, this is one of the main uses for which Idris was developed. We'll come back to this topic in later chapters.

3. Booleans

```
namespace Booleans
```

In a similar way, we can define the standard type `Bool` of booleans, with members `False` and `True`.

```
|||  Boolean Data Type
data Bool = True | False
```

This definition is written in the simplified style, similar to `Day`. It can also be written in the verbose style:

```
data Bool : Type where
     True : Bool
    False : Bool
```

The verbose style is more powerful because it allows us to assign precise types to individual constructors. This will become very useful later on.

Although we are rolling our own booleans here for the sake of building up everything from scratch, Idris does, of course, provide a default implementation of the booleans in its standard library, together with a multitude of useful functions and lemmas. (Take a look at `Prelude` in the Idris library documentation if you're interested.) Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
negb : (b : Bool) → Bool
negb True  = False
negb False = True

andb : (b1 : Bool) → (b2 : Bool) → Bool
andb True  b2 = b2
andb False b2 = False

orb : (b1 : Bool) → (b2 : Bool) → Bool
orb True  b2 = True
orb False b2 = b2
```

The last two illustrate Idris's syntax for multi-argument function definitions. The corresponding multi-argument application syntax is illustrated by the following four "unit tests," which constitute a complete specification – a truth table – for the `orb` function:

```
testOrb1 : (orb True  False) = True
testOrb1 = Refl

testOrb2 : (orb False False) = False
testOrb2 = Refl

testOrb3 : (orb False True)  = True
testOrb3 = Refl

testOrb4 : (orb True  True)  = True
testOrb4 = Refl
```

Edit this.

We can also introduce some familiar syntax for the boolean operations we have just defined. The `syntax` command defines a new symbolic notation for an existing definition, and `infixl` specifies left-associative fixity.

```
infixl 4 /\, \/

(/\) : Bool → Bool → Bool
(/\) = andb

(\/) : Bool → Bool → Bool
(\/) = orb

testOrb5 : False \/ False \/ True = True
testOrb5 = Refl
```

3.1. Exercise: 1 star (nandb). Fill in the hole `?nandb_rhs` and complete the following function; then make sure that the assertions below can each be verified by Idris. (Fill in each of the holes, following the model of the `orb` tests above.) The function should return *True* if either or both of its inputs are *False*.

```
nandb : (b1 : Bool) → (b2 : Bool) → Bool
nandb b1 b2 = ?nandb_rhs

test_nandb1 : (nandb True  False) = True
test_nandb1 = ?test_nandb1_rhs

test_nandb2 : (nandb False False) = True
test_nandb2 = ?test_nandb2_rhs

test_nandb3 : (nandb False True)  = True
test_nandb3 = ?test_nandb3_rhs

test_nandb4 : (nandb True  True)  = False
test_nandb4 = ?test_nandb4_rhs
```

□

3.2. Exercise: 1 star (andb3). Do the same for the `andb3` function below. This function should return *True* when all of its inputs are *True*, and *False* otherwise.

```
andb3 : (b1 : Bool) → (b2 : Bool) → (b3 : Bool) → Bool
andb3 b1 b2 b3 = ?andb3_rhs

test_andb31 : (andb3 True  True  True)  = True
test_andb31 = ?test_andb31_rhs

test_andb32 : (andb3 False True  True)  = False
test_andb32 = ?test_andb32_rhs

test_andb33 : (andb3 True  False True)  = False
test_andb33 = ?test_andb33_rhs
```

```
test_andb34 : (andb3 True  True  False) = False
test_andb34 = ?test_andb34_rhs
```

□

4. Function Types

Every expression in Idris has a type, describing what sort of thing it computes. The `:type` (or `:t`) REPL command asks Idris to print the type of an expression.

For example, the type of `negb True` is `Bool`.

```
λΠ> :type True
-- True : Bool
λΠ> :t negb True
-- negb True : Bool
```

Confirm the "function types" wording.

Functions like `negb` itself are also data values, just like `True` and `False`. Their types are called *function types*, and they are written with arrows.

```
λΠ> :t negb
-- negb : Bool → Bool
```

The type of `negb`, written `Bool → Bool` and pronounced "`Bool` arrow `Bool`," can be read, "Given an input of type `Bool`, this function produces an output of type `Bool`." Similarly, the type of `andb`, written `Bool → Bool → Bool`, can be read, "Given two inputs, both of type `Bool`, this function produces an output of type `Bool`."

5. Modules

Flesh this out and discuss namespaces

Idris provides a *module system*, to aid in organizing large developments.

6. Numbers

```
namespace Numbers
```

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finite set of elements. A More interesting way of defining a type is to give a collection of *inductive rules* describing its elements. For example, we can define the natural numbers as follows:

```
  data Nat : Type where
       Z : Nat
       S : Nat → Nat
```

The clauses of this definition can be read: - `Z` is a natural number. - `S` is a "constructor" that takes a natural number and yields another one – that is, if `n` is a natural number, then `S n` is too.

Let's look at this in a little more detail.

Every inductively defined set (`Day`, `Nat`, `Bool`, etc.) is actually a set of *expressions*. The definition of `Nat` says how expressions in the set `Nat` can be constructed:

- the expression `Z` belongs to the set `Nat`;
- if `n` is an expression belonging to the set `Nat`, then `S n` is also an expression belonging to the set `Nat`; and
- expression formed in these two ways are the only ones belonging to the set `Nat`.

The same rules apply for our definitions of `Day` and `Bool`. The annotations we used for their constructors are analogous to the one for the `Z` constructor, indicating that they don't take any arguments.

These three conditions are the precise force of inductive declarations. They imply that the expression `Z`, the expression `S Z`, the expression `S (S Z)`, the expression `S (S (S Z))` and so on all belong to the set `Nat`, while other expressions like `True`, `andb True False`, and `S (S False)` do not.

We can write simple functions that pattern match on natural numbers just as we did above – for example, the predecessor function:

```
pred : (n : Nat) → Nat
pred  Z     = Z
pred (S k) = k
```

The second branch can be read: "if `n` has the form `S k` for some `k`, then return `k`."

```
minusTwo : (n : Nat) → Nat
minusTwo  Z         = Z
minusTwo (S  Z)     = Z
minusTwo (S (S k)) = k
```

Because natural numbers are such a pervasive form of data, Idris provides a tiny bit of built-in magic for parsing and printing them: ordinary Arabic numerals can be used as an alternative to the "unary" notation defined by the constructors `S` and `Z`. Idris prints numbers in Arabic form by default:

```
λΠ> S (S (S (S Z)))
-- 4 : Nat
λΠ> minusTwo 4
-- 2 : Nat
```

The constructor `S` has the type `Nat → Nat`, just like the functions `minusTwo` and `pred`:

```
λΠ> :t S
λΠ> :t pred
λΠ> :t minusTwo
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between the first one and the other two: functions like `pred` and `minusTwo` come with *computation rules* – e.g., the definition of `pred` says that `pred 2` can be simplified to `1` – while the definition of `S` has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not *do* anything at all!

For most function definitions over numbers, just pattern matching is not enough: we also need recursion. For example, to check that a number `n` is even, we may need to recursively check whether `n-2` is even.

```
||| Determine whether a number is even.
||| @n a number
evenb : (n : Nat) → Bool
evenb  Z        = True
evenb (S  Z)    = False
evenb (S (S k)) = evenb k
```

We can define `oddb` by a similar recursive declaration, but here is a simpler definition that is a bit easier to work with:

```
||| Determine whether a number is odd.
||| @n a number
oddb : (n : Nat) → Bool
oddb n = negb (evenb n)

testOddb1 : oddb 1 = True
testOddb1 = Refl

testOddb2 : oddb 4 = False
testOddb2 = Refl
```

Naturally, we can also define multi-argument functions by recursion.

```
namespace Playground2

  plus : (n : Nat) → (m : Nat) → Nat
  plus  Z    m = m
  plus (S k) m = S (Playground2.plus k m)
```

Adding three to two now gives us five, as we'd expect.

```
λΠ> plus 3 2
```

The simplification that Idris performs to reach this conclusion can be visualized as follows:

```
plus (S (S (S Z))) (S (S Z))
```

↪ `S (plus (S (S Z)) (S (S Z)))` by the second clause of `plus`

↪ `S (S (plus (S Z) (S (S Z))))` by the second clause of `plus`

↪ `S (S (S (plus Z (S (S Z)))))` by the second clause of `plus`

↪ `S (S (S (S (S Z))))` by the first clause of `plus`

As a notational convenience, if two or more arguments have the same type, they can be written together. In the following definition, `(n, m : Nat)` means just the same as if we had written `(n : Nat) → (m : Nat)`.

```
mult : (n, m : Nat) → Nat
mult  Z    = Z
mult (S k) = plus m (mult k m)

testMult1 : (mult 3 3) = 9
testMult1 = Refl
```

You can match two expressions at once:

```
minus : (n, m : Nat) → Nat
minus  Z     _    = Z
minus  n     Z    = n
minus (S k) (S j) = minus k j
```

Verify this.

The `_` in the first line is a *wildcard pattern*. Writing `_` in a pattern is the same as writing some variable that doesn't get used on the right-hand side. This avoids the need to invent a bogus variable name.

```
exp : (base, power : Nat) → Nat
exp base  Z    = S Z
exp base (S p) = mult base (exp base p)
```

6.1. Exercise: 1 star (factorial). Recall the standard mathematical factorial function:

$$factorial(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times factorial(n-1), & \text{otherwise} \end{cases}$$

Translate this into Idris.

```
factorial : (n : Nat) → Nat
factorial n = ?factorial_rhs

testFactorial1 : factorial 3 = 6
testFactorial1 = ?testFactorial1_rhs

testFactorial2 : factorial 5 = mult 10 12
testFactorial2 = ?testFactorial2_rhs
```

□

We can make numerical expressions a little easier to read and write by introducing `syntax` for addition, multiplication, and subtraction.

```
syntax [x] "+" [y] = plus  x y
syntax [x] "-" [y] = minus x y
syntax [x] "*" [y] = mult  x y
```

```
λΠ> :t (0 + 1) + 1
```

(The details are not important, but interested readers can refer to the optional "More on Syntax" section at the end of this chapter.)

Note that these do not change the definitions we've already made: they are simply instructions to the Idris parser to accept `x + y` in place of `plus x y` and, conversely, to the Idris pretty-printer to display `plus x y` as `x + y`.

The `beq_nat` function tests *Nat*ural numbers for `equality`, yielding a `boolean`.

```
||| Test natural numbers for equality.
beq_nat : (n, m : Nat) → Bool
beq_nat  Z     Z     = True
beq_nat  Z    (S j)  = False
beq_nat (S k)  Z     = False
beq_nat (S k) (S j)  = beq_nat k j
```

The `leb` function tests whether its first argument is less than or equal to its second argument, yielding a boolean.

```
||| Test whether a number is less than or equal to another.
leb : (n, m : Nat) → Bool
leb  Z     m     = True
leb (S k)  Z     = False
leb (S k) (S j)  = leb k j

testLeb1 : leb 2 2 = True
testLeb1 = Refl

testLeb2 : leb 2 4 = True
testLeb2 = Refl

testLeb3 : leb 4 2 = False
testLeb3 = Refl
```

6.2. Exercise: 1 star (blt_nat). The `blt_nat` function tests *Nat*ural numbers for `less-than`, yielding a `boolean`. Instead of making up a new recursive function for this one, define it in terms of a previously defined function.

```
blt_nat : (n, m : Nat) → Bool
blt_nat n m = ?blt_nat_rhs

test_blt_nat_1 : blt_nat 2 2 = False
```

```
test_blt_nat_1 = ?test_blt_nat_1_rhs

test_blt_nat_2 : blt_nat 2 4 = True
test_blt_nat_2 = ?test_blt_nat_2_rhs

test_blt_nat_3 : blt_nat 4 2 = False
test_blt_nat_3 = ?test_blt_nat_3_rhs
```

□

7. Proof by Simplification

Now that we've defined a few datatypes and functions, let's turn to stating and proving properties of their behavior. Actually, we've already started doing this: each of the functions beginning with `test` in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same: use `Refl` to check that both sides contain identical values.

The same sort of "proof by simplification" can be used to prove more interesting properties as well. For example, the fact that `0` is a "neutral element" for `+` on the left can be proved just by observing that `0 + n` reduces to `n` no matter what `n` is, a fact that can be read directly off the definition of `plus`.

```
plus_Z_n : (n : Nat) → 0 + n = n
plus_Z_n n = Refl
```

It will be useful later to know that [reflexivity] does some simplification – for example, it tries "unfolding" defined terms, replacing them with their right-hand sides. The reason for this is that, if reflexivity succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions `Refl` has created by all this simplification and unfolding.

Other similar theorems can be proved with the same pattern.

```
plus_1_l : (n : Nat) → 1 + n = S n
plus_1_l n = Refl

mult_0_l : (n : Nat) → 0 * n = 0
mult_0_l n = Refl
```

The `_l` suffix in the names of these theorems is pronounced "on the left."

Although simplification is powerful enough to prove some fairly general facts, there are many statements that cannot be handled by simplification alone. For instance, we cannot use it to prove that `0` is also a neutral element for `+` *on the right*.

```
plus_n_Z : (n : Nat) → n = n + 0
plus_n_Z n = Refl
```

```
When checking right hand side of plus_n_Z with expected type
             n = n + 0
```

```
Type mismatch between
        plus n 0 = plus n 0 (Type of Refl)
and
        n = plus n 0 (Expected type)

Specifically:
        Type mismatch between
                plus n 0
        and
                n
```

(Can you explain why this happens?)

The next chapter will introduce *induction*, a powerful technique that can be used for proving this goal. For the moment, though, let's look at a few more simple tactics.

8. Proof by Rewriting

This theorem is a bit more interesting than the others we've seen:

```
plus_id_example : (n, m : Nat) → (n = m) → n + n = m + m
```

Instead of making a universal claim about all numbers `n` and `m`, it talks about a more specialized property that only holds when `n = m`. The arrow symbol is pronounced "implies."

As before, we need to be able to reason by assuming the existence of some numbers `n` and `m`. We also need to assume the hypothesis `n = m`.

Edit.

The `intros` tactic will serve to move all three of these from the goal into assumptions in the current context.

Since `n` and `m` are arbitrary numbers, we can't just use simplification to prove this theorem. Instead, we prove it by observing that, if we are assuming `n = m`, then we can replace `n` with `m` in the goal statement and obtain an equality with the same expression on both sides. The tactic that tells Idris to perform this replacement is called `rewrite`.

```
plus_id_example n m prf = rewrite prf in Refl
```

The first two variables on the left side move the universally quantified variables `n` and `m` into the context. The third moves the hypothesis `n = m` into the context and gives it the name `prf`. The right side tells Idris to rewrite the current goal (`n + n = m + m`) by replacing the left side of the equality hypothesis `prf` with the right side.

8.1. Exercise: 1 star (plus_id_exercise). Fill in the proof.

```
plus_id_exercise : (n, m, o : Nat) → (n = m) → (m = o) → n + m = m + o
plus_id_exercise n m o prf prf1 = ?plus_id_exercise_rhs
```

□

The prefix `?` on the right-hand side of an equation tells Idris that we want to skip trying to prove this theorem and just leave a hole. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use holes to delay defining them for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped.

Decide whether or not to discuss `postulate`.
"'idris – Be careful, though: every time you say 'postulate' you are leaving a door open – for total nonsense to enter Idris's nice, rigorous, formally checked world! "'

We can also use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context. If the statement of the previously proved theorem involves quantified variables, as in the example below, Idris tries to instantiate them by matching with the current goal.

```
mult_0_plus : (n, m : Nat) → (0 + n) * m = n * (0 + m)
mult_0_plus n m = Refl
```

Unlike in Coq, we don't need to perform such a rewrite for `mult_0_plus` in Idris and can just use `Refl` instead.

8.2. Exercise: 2 starts (mult_S_1).

```
mult_S_1 : (n, m : Nat) → (m = S n) → m * (1 + n) = m * m
mult_S_1 n m prf = ?mult_S_1_rhs
```

□

9. Proof by Case Analysis

Of course, not everything can be proved by simple calculation and rewriting: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can block simplification. For example, if we try to prove the following fact using the `Refl` tactic as above, we get stuck.

```
plus_1_neq_0_firsttry : (n : Nat) → beq_nat (n + 1) 0 = False
plus_1_neq_0_firsttry n = Refl -- does nothing!
```

The reason for this is that the definitions of both `beq_nat` and `+` begin by performing a `match` on their first argument. But here, the first argument to `+` is the unknown number `n` and the argument to `beq_nat` is the compound expression `n + 1`; neither can be simplified.

To make progress, we need to consider the possible forms of `n` separately. If `n` is `Z`, then we can calculate the final result of `beq_nat (n + 1) 0` and check that it is, indeed, `False`. And if `n = S k` for some `k`, then, although we don't know exactly what number `n + 1` yields, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `beq_nat (n + 1) 0` will yield `False`.

To tell Idris to consider, separately, the cases where `n = Z` and where `n = S k`, simply case split on `n`.

Mention case splitting interactively in Emacs, Atom, etc.

```
plus_1_neq_0 : (n : Nat) → beq_nat (n + 1) 0 = False
plus_1_neq_0  Z    = Refl
plus_1_neq_0 (S k) = Refl
```

Case splitting on `n` generates *two* holes, which we must then prove, separately, in order to get Idris to accept the theorem.

In this example, each of the holes is easily filled by a single use of `Refl`, which itself performs some simplification – e.g., the first one simplifies `beq_nat (S k + 1) 0` to `False` by first rewriting `(S k + 1)` to `S (k + 1)`, then unfolding `beq_nat`, simplifying its pattern matching.

There are no hard and fast rules for how proofs should be formatted in Idris. However, if the places where multiple holes are generated are lifted to lemmas, then the proof will be readable almost no matter what choices are made about other aspects of layout.

This is also a good place to mention one other piece of somewhat obvious advice about line lengths. Beginning Idris users sometimes tend to the extremes, either writing each tactic on its own line or writing entire proofs on one line. Good style lies somewhere in the middle. One reasonable convention is to limit yourself to 80-character lines.

The case splitting strategy can be used with any inductively defined datatype. For example, we use it next to prove that boolean negation is involutive – i.e., that negation is its own inverse.

```
||| A proof that boolean negation is involutive.
negb_involutive : (b : Bool) → negb (negb b) = b
negb_involutive True  = Refl
negb_involutive False = Refl
```

Note that the case splitting here doesn't introduce any variables because none of the subcases of the patterns need to bind any, so there is no need to specify any names.

It is sometimes useful to case split on more than one parameter, generating yet more proof obligations. For example:

```
andb_commutative : (b, c : Bool) → andb b c = andb c b
andb_commutative True  True  = Refl
```

```
andb_commutative True  False = Refl
andb_commutative False True  = Refl
andb_commutative False False = Refl
```

In more complex proofs, it is often better to lift subgoals to lemmas:

```
andb_commutative'_rhs_1 : (c : Bool) → c = andb c True
andb_commutative'_rhs_1 True  = Refl
andb_commutative'_rhs_1 False = Refl

andb_commutative'_rhs_2 : (c : Bool) → False = andb c False
andb_commutative'_rhs_2 True  = Refl
andb_commutative'_rhs_2 False = Refl

andb_commutative' : (b, c : Bool) → andb b c = andb c b
andb_commutative' True  = andb_commutative'_rhs_1
andb_commutative' False = andb_commutative'_rhs_2
```

9.1. Exercise: 2 stars (andb_true_elim2). Prove the following claim, lift cases (and subcases) to lemmas when case split.

```
andb_true_elim_2 : (b, c : Bool) → (andb b c = True) → c = True
andb_true_elim_2 b c prf = ?andb_true_elim_2_rhs
```

□

9.2. Exercise: 1 star (zero_nbeq_plus_1).

```
zero_nbeq_plus_1 : (n : Nat) → beq_nat 0 (n + 1) = False
zero_nbeq_plus_1 n = ?zero_nbeq_plus_1_rhs
```

□

Discuss associativity.

10. Structural Recursion (Optional)

Here is a copy of the definition of addition:

```
plus' : Nat → Nat → Nat
plus'  Z       right = right
plus' (S left) right = S (plus' left right)
```

When Idris checks this definition, it notes that `plus'` is "decreasing on 1st argument." What this means is that we are performing a *structural recursion* over the argument `left` – i.e., that we make recursive calls only on strictly smaller values of `left`. This implies that all calls to `plus'` will eventually terminate. Idris demands that some argument of *every* recursive definition is "decreasing."

This requirement is a fundamental feature of Idris's design: In particular, it guarantees that every function that can be defined in Idris will terminate on all inputs.

However, because Idris's "decreasing analysis" is not very sophisticated, it is sometimes necessary to write functions in slightly unnatural ways.

Verify the previous claims.

Add decreasing exercise.

11. More Exercises

11.1. Exercise: 2 stars (boolean_functions). Use the tactics you have learned so far to prove the following theorem about boolean functions.

```
identity_fn_applied_twice : (f : Bool → Bool) →
                            ((x : Bool) → f x = x) →
                            (b : Bool) → f (f b) = b
identity_fn_applied_twice f g b = ?identity_fn_applied_twice_rhs
```

Now state and prove a theorem `negation_fn_applied_twice` similar to the previous one but where the second hypothesis says that the function `f` has the property that `f x = negb x`.

```
-- FILL IN HERE
```

□

11.2. Exercise: 2 start (andb_eq_orb). Prove the following theorem. (You may want to first prove a subsidiary lemma or two. Alternatively, remember that you do not have to introduce all hypotheses at the same time.)

```
andb_eq_orb : (b, c : Bool) → (andb b c = orb b c) → b = c
andb_eq_orb b c prf = ?andb_eq_orb_rhs
```

□

11.3. Exercise: 3 stars (binary). Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either

- zero,
- twice a binary number, or
- one more than twice a binary number.

(a) First, write an inductive definition of the type `Bin` corresponding to this description of binary numbers.

(Hint: Recall that the definition of `Nat` from class,

```
data Nat : Type where
       Z : Nat
       S : Nat → Nat
```

says nothing about what `Z` and `S` "mean." It just says "`Z` is in the set called `Nat`, and if `n` is in the set then so is `S n`." The interpretation of `Z` as zero and `S` as successor/plus one comes from the way that we *use* `Nat` values, by writing functions to do things with them, proving things about them, and so on. Your definition of `Bin` should be correspondingly simple; it is the functions you will write next that will give it mathematical meaning.)

(b) Next, write an increment function `incr` for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.

(c) Write five unit tests `test_bin_incr_1`, `test_bin_incr_2`, etc. for your increment and binary-to-unary functions. Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

```
-- FILL IN HERE
```

□

CHAPTER 3

Induction: Proof by Induction

```
module Induction
```

First, we import all of our definitions from the previous chapter.

```
import Basics
```

Next, we import the following `Prelude` modules, since we'll be dealing with natural numbers.

```
import Prelude.Interfaces
import Prelude.Nat
```

For `import Basics` to work, you first need to use `idris` to compile `Basics.lidr` into `Basics.ibc`. This is like making a .class file from a .java file, or a .o file from a .c file. There are at least two ways to do it:

- In your editor with an Idris plugin, e.g. Emacs:

  Open `Basics.lidr`. Evaluate `idris-load-file`.

  There exists similar support for Vim, Sublime Text and Visual Studio Code as well.

- From the command line:

  Run `idris --check --total --noprelude src/Basics.lidr`.

  Refer to the Idris man page (or `idris --help` for descriptions of the flags.

1. Proof by Induction

We proved in the last chapter that `0` is a neutral element for `+` on the left using an easy argument based on simplification. The fact that it is also a neutral element on the *right*...

```
Theorem plus_n_O_firsttry : forall n:nat,
  n = n + 0.
```

... cannot be proved in the same simple way in Coq, but as we saw in `Basics`, Idris's `Refl` just works.

To prove interesting facts about numbers, lists, and other inductively defined sets, we usually need a more powerful reasoning principle: *induction*.

Recall (from high school, a discrete math course, etc.) the principle of induction over natural numbers: If `p n` is some proposition involving a natural number `n` and we want to show that `p` holds for *all* numbers `n`, we can reason like this:

- show that `p Z` holds;
- show that, for any `k`, if `p k` holds, then so does `p (S k)`;
- conclude that `p n` holds for all `n`.

In Idris, the steps are the same and can often be written as function clauses by case splitting. Here's how this works for the theorem at hand.

```
plus_n_Z : (n : Nat) → n = n + 0
plus_n_Z  Z    = Refl
plus_n_Z (S k) =
  let inductiveHypothesis = plus_n_Z k in
    rewrite inductiveHypothesis in Refl
```

In the first clause, `n` is replaced by `Z` and the goal becomes `0 = 0`, which follows by `Refl`exivity. In the second, `n` is replaced by `S k` and the goal becomes `S k = S (plus k 0)`. Then we define the inductive hypothesis, `k = k + 0`, which can be written as `plus_n_Z k`, and the goal follows from it.

```
minus_diag : (n : Nat) → minus n n = 0
minus_diag  Z    = Refl
minus_diag (S k) = minus_diag k
```

1.1. Exercise: 2 stars, recommended (basic_induction). Prove the following using induction. You might need previously proven results.

```
mult_0_r : (n : Nat) → n * 0 = 0
mult_0_r n = ?mult_0_r_rhs

plus_n_Sm : (n, m : Nat) → S (n + m) = n + (S m)
plus_n_Sm n m = ?plus_n_Sm_rhs

plus_comm : (n, m : Nat) → n + m = m + n
plus_comm n m = ?plus_comm_rhs

plus_assoc : (n, m, p : Nat) → n + (m + p) = (n + m) + p
plus_assoc n m p = ?plus_assoc_rhs
```

□

1.2. Exercise: 2 stars (double_plus). Consider the following function, which doubles its argument:

```
double : (n : Nat) → Nat
double  Z    = Z
double (S k) = S (S (double k))
```

Use induction to prove this simple fact about `double`:

```
double_plus : (n : Nat) → double n = n + n
double_plus n = ?double_plus_rhs
```

□

1.3. Exercise: 2 stars, optional (evenb_S). One inconvenient aspect of our definition of `evenb n` is that it may need to perform a recursive call on `n - 2`. This makes proofs about `evenb n` harder when done by induction on `n`, since we may need an induction hypothesis about `n - 2`. The following lemma gives a better characterization of `evenb (S n)`:

```
evenb_S : (n : Nat) → evenb (S n) = negb (evenb n)
evenb_S n = ?evenb_S_rhs
```

□

2. Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are often broken into a sequence of theorems, with later proofs referring to earlier theorems. But sometimes a proof will require some miscellaneous fact that is too trivial and of too little general interest to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed "sub-theorem" right at the point where it is used. The `assert` tactic allows us to do this. For example, our earlier proof of the `mult_0_plus` theorem referred to a previous theorem named `plus_Z_n`. We could instead use `assert` to state and prove `plus_Z_n` in-line:

```
mult_0_plus' : (n, m : Nat) → (0 + n) * m = n * m
mult_0_plus' n m = Refl
```

The `assert` tactic introduces two sub-goals. The first is the assertion itself; by prefixing it with `H:` we name the assertion `H`. (We can also name the assertion with `as` just as we did above with `destruct` and `induction`, i.e., `assert (0 + n = n) as H`.) Note that we surround the proof of this assertion with curly braces `{ ... }`, both for readability and so that, when using Coq interactively, we can see more easily when we have finished this sub-proof. The second goal is the same as the one at the point where we invoke `assert` except that, in the context, we now have the assumption `H` that `0 + n = n`. That is, `assert` generates one subgoal where we must prove the asserted fact and a second subgoal where we can use the asserted fact to make progress on whatever we were trying to prove in the first place.

The `assert` tactic is handy in many sorts of situations. For example, suppose we want to prove that `(n + m) + (p + q) = (m + n) + (p + q)`. The only difference between the two sides of the `=` is that the arguments `m` and `n` to the first inner `+` are swapped, so it seems we should be able to use the commutativity of addition (`plus_comm`) to rewrite one into the other. However, the `rewrite` tactic is a little stupid about *where* it applies the rewrite. There are three uses of `+` here, and it turns out that doing `rewrite → plus_comm` will affect only the *outer* one…

```
plus_rearrange_firsttry : (n, m, p, q : Nat) →
                          (n + m) + (p + q) = (m + n) + (p + q)
plus_rearrange_firsttry n m p q = rewrite plus_comm in Refl
```

```
When checking right hand side of plus_rearrange_firsttry with expected type
        n + m + (p + q) = m + n + (p + q)

_ does not have an equality type ((n1 : Nat) →
(n1 : Nat) → plus n1 m1 = plus m1 n1)
```

To get `plus_comm` to apply at the point where we want it to, we can introduce a local lemma using the `where` keyword stating that `n + m = m + n` (for the particular `m` and `n` that we are talking about here), prove this lemma using `plus_comm`, and then use it to do the desired rewrite.

```
plus_rearrange : (n, m, p, q : Nat) →
                 (n + m) + (p + q) = (m + n) + (p + q)
plus_rearrange n m p q = rewrite plus_rearrange_lemma n m in Refl
  where
  plus_rearrange_lemma : (n, m : Nat) → n + m = m + n
  plus_rearrange_lemma = plus_comm
```

3. More Exercises

3.1. Exercise: 3 stars, recommended (mult_comm). Use `rewrite` to help prove this theorem. You shouldn't need to use induction on `plus_swap`.

```
plus_swap : (n, m, p : Nat) → n + (m + p) = m + (n + p)
plus_swap n m p = ?plus_swap_rhs
```

Now prove commutativity of multiplication. (You will probably need to define and prove a separate subsidiary theorem to be used in the proof of this one. You may find that `plus_swap` comes in handy.)

```
mult_comm : (m, n : Nat) → m * n = n * m
mult_comm m n = ?mult_comm_rhs
```

□

3.2. Exercise: 3 stars, optional (more_exercises). Take a piece of paper. For each of the following theorems, first *think* about whether (a) it can be proved using only simplification and rewriting, (b) it also requires case analysis (`destruct`), or (c) it also requires induction. Write down your prediction. Then fill in the proof. (There is no need to turn in your piece of paper; this is just to encourage you to reflect before you hack!)

```
leb_refl : (n : Nat) → True = leb n n
leb_refl n = ?leb_refl_rhs
```

```
zero_nbeq_S : (n : Nat) → beq_nat 0 (S n) = False
zero_nbeq_S n = ?zero_nbeq_S_rhs
```

```
andb_false_r : (b : Bool) → andb b False = False
andb_false_r b = ?andb_false_r_rhs

plus_ble_compat_l : (n, m, p : Nat) →
                    leb n m = True → leb (p + n) (p + m) = True
plus_ble_compat_l n m p prf = ?plus_ble_compat_l_rhs

S_nbeq_0 : (n : Nat) → beq_nat (S n) 0 = False
S_nbeq_0 n = ?S_nbeq_0_rhs

mult_1_l : (n : Nat) → 1 * n = n
mult_1_l n = ?mult_1_l_rhs

all3_spec : (b, c : Bool) →
            orb (andb b c)
                (orb (negb b)
                     (negb c))
            = True
all3_spec b c = ?all3_spec_rhs

mult_plus_distr_r : (n, m, p : Nat) → (n + m) * p = (n * p) + (m * p)
mult_plus_distr_r n m p = ?mult_plus_distr_r_rhs

mult_assoc : (n, m, p : Nat) → n * (m * p) = (n * m) * p
mult_assoc n m p = ?mult_assoc_rhs
```

□

3.3. Exercise: 2 stars, optional (beq_nat_refl). Prove the following theorem. (Putting the `True` on the left-hand side of the equality may look odd, but this is how the theorem is stated in the Coq standard library, so we follow suit. Rewriting works equally well in either direction, so we will have no problem using the theorem no matter which way we state it.)

```
beq_nat_refl : (n : Nat) → True = beq_nat n n
beq_nat_refl n = ?beq_nat_refl_rhs
```

□

3.4. Exercise: 2 stars, optional (plus_swap'). The `replace` tactic allows you to specify a particular subterm to rewrite and what you want it rewritten to: `replace (t) with (u)` replaces (all copies of) expression `t` in the goal by expression `u`, and generates `t = u` as an additional subgoal. This is often useful when a plain `rewrite` acts on the wrong part of the goal.

Use the `replace` tactic to do a proof of `plus_swap'`, just like `plus_swap` but without needing `assert (n + m = m + n)`.

```
plus_swap' : (n, m, p : Nat) → n + (m + p) = m + (n + p)
plus_swap' n m p = ?plus_swap'_rhs
```

□

3.5. Exercise: 3 stars, recommended (binary_commute). Recall the `incr` and `bin_to_nat` functions that you wrote for the `binary` exercise in the `Basics` chapter. Prove that the following diagram commutes:

```
      bin --------- incr ------> bin
       |                          |
   bin_to_nat                 bin_to_nat
       |                          |
       v                          v
      nat ---------- S --------> nat
```

That is, incrementing a binary number and then converting it to a (unary) natural number yields the same result as first converting it to a natural number and then incrementing. Name your theorem `bin_to_nat_pres_incr` ("pres" for "preserves").

Before you start working on this exercise, please copy the definitions from your solution to the `binary` exercise here so that this file can be graded on its own. If you find yourself wanting to change your original definitions to make the property easier to prove, feel free to do so!

□

3.6. Exercise: 5 stars, advanced (binary_inverse). This exercise is a continuation of the previous exercise about binary numbers. You will need your definitions and theorems from there to complete this one.

(a) First, write a function to convert natural numbers to binary numbers. Then prove that starting with any natural number, converting to binary, then converting back yields the same natural number you started with.

(b) You might naturally think that we should also prove the opposite direction: that starting with a binary number, converting to a natural, and then back to binary yields the same number we started with. However, this is not true! Explain what the problem is.

(c) Define a "direct" normalization function – i.e., a function `normalize` from binary numbers to binary numbers such that, for any binary number b, converting to a natural and then back to binary yields `(normalize b)`. Prove it. (Warning: This part is tricky!)

Again, feel free to change your earlier definitions if this helps here.

□

CHAPTER 4

Lists: Working with Structured Data

```
module Lists

import Basics

%hide Prelude.Basics.fst
%hide Prelude.Basics.snd
%hide Prelude.Nat.pred

%default total
```

1. Pairs of Numbers

In an inductive type definition, each constructor can take any number of arguments – none (as with `true` and `Z`), one (as with `S`), or more than one, as here:

```
data NatProd : Type where
  Pair : Nat → Nat → NatProd
```

This declaration can be read: "There is just one way to construct a pair of numbers: by applying the constructor `Pair` to two arguments of type `Nat`."

```
λΠ> :t Pair 3 5
```

Here are two simple functions for extracting the first and second components of a pair. The definitions also illustrate how to do pattern matching on two-argument constructors.

```
fst : (p : NatProd) → Nat
fst (Pair x y) = x

snd : (p : NatProd) → Nat
snd (Pair x y) = y

λΠ> fst (Pair 3 5)
-- 3 : Nat
```

Since pairs are used quite a bit, it is nice to be able to write them with the standard mathematical notation `(x,y)` instead of `Pair x y`. We can tell Idris to allow this with a `syntax` declaration.

```
syntax "(" [x] "," [y] ")" = Pair x y
```

The new pair notation can be used both in expressions and in pattern matches (indeed, we've actually seen this already in the previous chapter, in the definition of the `minus` function – this works because the pair notation is also provided as part of the standard library):

```
λΠ> fst (3,5)
-- 3 : Nat

fst' : (p : NatProd) → Nat
fst' (x,y) = x

snd' : (p : NatProd) → Nat
snd' (x,y) = y

swap_pair : (p : NatProd) → NatProd
swap_pair (x,y) = (y,x)
```

Let's try to prove a few simple facts about pairs.

If we state things in a particular (and slightly peculiar) way, we can complete proofs with just reflexivity (and its built-in simplification):

```
surjective_pairing' : (n,m : Nat) → (n,m) = (fst (n,m), snd (n,m))
surjective_pairing' n m = Refl
```

But `Refl` is not enough if we state the lemma in a more natural way:

```
surjective_pairing_stuck : (p : NatProd) → p = (fst p, snd p)
surjective_pairing_stuck p = Refl

When checking right hand side of
  surjective_pairing_stuck with expected type p = Pair (fst p) (snd p)
...
Type mismatch between p and Pair (fst p) (snd p)
```

We have to expose the structure of `p` so that Idris can perform the pattern match in `fst` and `snd`. We can do this with `case`.

```
surjective_pairing : (p : NatProd) → p = (fst p, snd p)
surjective_pairing p = case p of (n,m) ⇒ Refl
```

Notice that `case` matches just one pattern here. That's because `NatProd`s can only be constructed in one way.

1.1. Exercise: 1 star (snd_fst_is_swap).

```
snd_fst_is_swap : (p : NatProd) → (snd p, fst p) = swap_pair p
snd_fst_is_swap p = ?snd_fst_is_swap_rhs
```

□

1.2. Exercise: 1 star, optional (fst_swap_is_snd).

```
fst_swap_is_snd : (p : NatProd) → fst (swap_pair p) = snd p
fst_swap_is_snd p = ?fst_swap_is_snd_rhs
```

□

2. Lists of Numbers

Generalizing the definition of pairs, we can describe the type of *lists* of numbers like this: "A list is either the empty list or else a pair of a number and another list."

```
data NatList : Type where
  Nil : NatList
  Cons : Nat → NatList → NatList
```

For example, here is a three-element list:

```
mylist : NatList
mylist = Cons 1 (Cons 2 (Cons 3 Nil))
```

As with pairs, it is more convenient to write lists in familiar programming notation. The following declarations allow us to use :: as an infix Cons operator and square brackets as an "outfix" notation for constructing lists.

```
syntax [x] "::" [l] = Cons x l
syntax "[ ]" = Nil
```

Figure out 'syntax' command for this and edit the section

Notation "[x ; .. ; y]" := (cons x .. (cons y nil) ..).

It is not necessary to understand the details of these declarations, but in case you are interested, here is roughly what's going on. The right associativity annotation tells Coq how to parenthesize expressions involving several uses of :: so that, for example, the next three declarations mean exactly the same thing:

```
mylist1 : NatList
mylist1 = 1 :: (2 :: (3 :: Nil))

mylist2 : NatList
mylist2 = 1::2::3::Nil
```

Definition mylist3 := [1;2;3].

The at level 60 part tells Coq how to parenthesize expressions that involve both :: and some other infix operator. For example, since we defined + as infix notation for the plus function at level 50,

Notation "x + y" := (plus x y) (at level 50, left associativity).

the + operator will bind tighter than :: , so 1 + 2 :: [3] will be parsed, as we'd expect, as (1 + 2) :: [3] rather than 1 + (2 :: [3]) .

(Expressions like "1 + 2 :: [3]" can be a little confusing when you read them in a .v file. The inner brackets, around 3, indicate a list, but the outer brackets, which are invisible in the HTML rendering, are there to instruct the "coqdoc" tool that the bracketed part should be displayed as Coq code rather than running text.)

The second and third Notation declarations above introduce the standard square-bracket notation for lists; the right-hand side of the third one illustrates Coq's syntax for declaring n-ary notations and translating them to nested sequences of binary constructors.

2.1. Repeat. A number of functions are useful for manipulating lists. For example, the `repeat` function takes a number `n` and a `count` and returns a list of length `count` where every element is `n`.

```
repeat : (n, count : Nat) → NatList
repeat n Z = Nil
repeat n (S k) = n :: repeat n k
```

2.2. Length. The `length` function calculates the length of a list.

```
length : (l : NatList) → Nat
length Nil = Z
length (h :: t) = S (length t)
```

2.3. Append. The `app` function concatenates (appends) two lists.

```
app : (l1, l2 : NatList) → NatList
app Nil l2 = l2
app (h :: t) l2 = h :: app t l2
```

Actually, `app` will be used a lot in some parts of what follows, so it is convenient to have an infix operator for it.

```
syntax [x] "++" [y] = app x y

test_app1 : ((1::2::3::[]) ++ (4::5::[])) = (1::2::3::4::5::[])
test_app1 = Refl

test_app2 : ([] ++ (4::5::[])) = (4::5::[])
test_app2 = Refl

test_app3 : ((1::2::3::[]) ++ []) = (1::2::3::[])
test_app3 = Refl
```

2.4. Head (with default) and Tail. Here are two smaller examples of programming with lists. The `hd` function returns the first element (the "head") of the list, while `tl` returns everything but the first element (the "tail"). Of course, the empty list has no first element, so we must pass a default value to be returned in that case.

```
hd : (default : Nat) → (l : NatList) → Nat
hd default Nil = default
hd default (h :: t) = h

tl : (l : NatList) → NatList
tl Nil = Nil
tl (h :: t) = t
```

```
test_hd1 : hd 0 (1::2::3::[]) = 1
test_hd1 = Refl

test_hd2 : hd 0 [] = 0
test_hd2 = Refl

test_tl : tl (1::2::3::[]) = (2::3::[])
test_tl = Refl
```

2.5. Exercises.

2.5.1. *Exercise: 2 stars, recommended (list_funs).* Complete the definitions of `nonzeros`, `oddmembers` and `countoddmembers` below. Have a look at the tests to understand what these functions should do.

```
nonzeros : (l : NatList) → NatList
nonzeros l = ?nonzeros_rhs

test_nonzeros : nonzeros (0::1::0::2::3::0::0::[]) = (1::2::3::[])
test_nonzeros = ?test_nonzeros_rhs

oddmembers : (l : NatList) → NatList
oddmembers l = ?oddmembers_rhs

test_oddmembers : oddmembers (0::1::0::2::3::0::0::[]) = (1::3::[])
test_oddmembers = ?test_oddmembers_rhs

countoddmembers : (l : NatList) → Nat
countoddmembers l = ?countoddmembers_rhs

test_countoddmembers1 : countoddmembers (1::0::3::1::4::5::[]) = 4
test_countoddmembers1 = ?test_countoddmembers1_rhs
```

□

2.5.2. *Exercise: 3 stars, advanced (alternate).* Complete the definition of `alternate`, which "zips up" two lists into one, alternating between elements taken from the first list and elements from the second. See the tests below for more specific examples.

Note: one natural and elegant way of writing `alternate` will fail to satisfy Idris's requirement that all function definitions be "obviously terminating." If you find yourself in this rut, look for a slightly more verbose solution that considers elements of both lists at the same time. (One possible solution requires defining a new kind of pairs, but this is not the only way.)

```
alternate : (l1, l2 : NatList) → NatList
alternate l1 l2 = ?alternate_rhs

test_alternate1 : alternate (1::2::3::[]) (4::5::6::[]) =
                            (1::4::2::5::3::6::[])
test_alternate1 = ?test_alternate1_rhs
```

```
test_alternate2 : alternate (1::[]) (4::5::6::[]) = (1::4::5::6::[])
test_alternate2 = ?test_alternate2_rhs

test_alternate3 : alternate (1::2::3::[]) (4::[]) = (1::4::2::3::[])
test_alternate3 = ?test_alternate3_rhs

test_alternate4 : alternate [] (20::30::[]) = (20::30::[])
test_alternate4 = ?test_alternate4_rhs
```

□

2.6. Bags via Lists. A `bag` (or `multiset`) is like a set, except that each element can appear multiple times rather than just once. One possible implementation is to represent a bag of numbers as a list.

```
Bag : Type
Bag = NatList
```

2.6.1. *Exercise: 3 stars, recommended (bag_functions).* Complete the following definitions for the functions `count`, `sum`, `add`, and `member` for bags.

```
count : (v : Nat) → (s : Bag) → Nat
count v s = ?count_rhs
```

All these proofs can be done just by `Refl`.

```
test_count1 : count 1 (1::2::3::1::4::1::[]) = 3
test_count1 = ?test_count1_rhs

test_count2 : count 6 (1::2::3::1::4::1::[]) = 0
test_count2 = ?test_count2_rhs
```

Multiset `sum` is similar to set `union`: `sum a b` contains all the elements of `a` and of `b`. (Mathematicians usually define union on multisets a little bit differently, which is why we don't use that name for this operation.)

How to forbid recursion here? Edit

For `sum` we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword Definition instead of Fixpoint, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether sum can be implemented in another way – perhaps by using functions that have already been defined.

```
sum : Bag → Bag → Bag
sum x y = ?sum_rhs

test_sum1 : count 1 (sum (1::2::3::[]) (1::4::1::[])) = 3
test_sum1 = ?test_sum1_rhs

add : (v : Nat) → (s : Bag) → Bag
add v s = ?add_rhs
```

```
test_add1 : count 1 (add 1 (1::4::1::[])) = 3
test_add1 = ?test_add1_rhs

test_add2 : count 5 (add 1 (1::4::1::[])) = 0
test_add2 = ?test_add2_rhs

member : (v : Nat) → (s : Bag) → Bool
member v s = ?member_rhs

test_member1 : member 1 (1::4::1::[]) = True
test_member1 = ?test_member1_rhs

test_member2 : member 2 (1::4::1::[]) = False
test_member2 = ?test_member2_rhs
```

□

2.6.2. *Exercise: 3 stars, optional (bag_more_functions).* Here are some more bag functions for you to practice with.

When `remove_one` is applied to a bag without the number to remove, it should return the same bag unchanged.

```
remove_one : (v : Nat) → (s : Bag) → Bag
remove_one v s = ?remove_one_rhs

test_remove_one1 : count 5 (remove_one 5 (2::1::5::4::1::[])) = 0
test_remove_one1 = ?test_remove_one1_rhs

test_remove_one2 : count 5 (remove_one 5 (2::1::4::1::[])) = 0
test_remove_one2 = ?test_remove_one2_rhs

test_remove_one3 : count 4 (remove_one 5 (2::1::5::4::1::4::[])) = 2
test_remove_one3 = ?test_remove_one3_rhs

test_remove_one4 : count 5 (remove_one 5 (2::1::5::4::5::1::4::[])) = 1
test_remove_one4 = ?test_remove_one4_rhs

remove_all : (v : Nat) → (s : Bag) → Bag
remove_all v s = ?remove_all_rhs

test_remove_all1 : count 5 (remove_all 5 (2::1::5::4::1::[])) = 0
test_remove_all1 = ?test_remove_all1_rhs

test_remove_all2 : count 5 (remove_all 5 (2::1::4::1::[])) = 0
test_remove_all2 = ?test_remove_all2_rhs

test_remove_all3 : count 4 (remove_all 5 (2::1::5::4::1::4::[])) = 2
test_remove_all3 = ?test_remove_all3_rhs

test_remove_all4 : count 5
                   (remove_all 5 (2::1::5::4::5::1::4::5::1::4::[])) = 0
test_remove_all4 = ?test_remove_all4_rhs

subset : (s1 : Bag) → (s2 : Bag) → Bool
subset s1 s2 = ?subset_rhs
```

```
test_subset1 : subset (1::2::[]) (2::1::4::1::[]) = True
test_subset1 = ?test_subset1_rhs

test_subset2 : subset (1::2::2::[]) (2::1::4::1::[]) = False
test_subset2 = ?test_subset2_rhs
```

□

2.6.3. *Exercise: 3 stars, recommended (bag_theorem).* Write down an interesting theorem `bag_theorem` about bags involving the functions `count` and `add`, and prove it. Note that, since this problem is somewhat open-ended, it's possible that you may come up with a theorem which is true, but whose proof requires techniques you haven't learned yet. Feel free to ask for help if you get stuck!

```
bag_theorem : ?bag_theorem
```

□

3. Reasoning About Lists

As with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, the simplification performed by `Refl` is enough for this theorem…

```
nil_app : (l : NatList) → ([] ++ l) = l
nil_app l = Refl
```

… because the `[]` is substituted into the "scrutinee" (the value being "scrutinized" by the match) in the definition of `app`, allowing the match itself to be simplified.

Also, as with numbers, it is sometimes helpful to perform case analysis on the possible shapes (empty or non-empty) of an unknown list.

```
tl_length_pred : (l : NatList) → pred (length l) = length (tl l)
tl_length_pred Nil = Refl
tl_length_pred (Cons n l') = Refl
```

Here, the `Nil` case works because we've chosen to define `tl Nil = Nil`. Notice that the case for `Cons` introduces two names, `n` and `l'`, corresponding to the fact that the `Cons` constructor for lists takes two arguments (the head and tail of the list it is constructing).

Usually, though, interesting theorems about lists require induction for their proofs.

3.0.1. *Micro-Sermon.* Simply reading example proof scripts will not get you very far! It is important to work through the details of each one, using Idris and thinking about what each step achieves. Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

3.1. Induction on Lists. Proofs by induction over datatypes like `NatList` are a little less familiar than standard natural number induction, but the idea is equally simple. Each `data` declaration defines a set of data values that can be built up using the declared constructors: a boolean can be either `True` or `False`; a number

can be either `Z` or `S` applied to another number; a list can be either `Nil` or `Cons` applied to a number and a list.

Moreover, applications of the declared constructors to one another are the *only* possible shapes that elements of an inductively defined set can have, and this fact directly gives rise to a way of reasoning about inductively defined sets: a number is either `Z` or else it is `S` applied to some *smaller* number; a list is either `Nil` or else it is `Cons` applied to some number and some *smaller* list; etc. So, if we have in mind some proposition `P` that mentions a list `l` and we want to argue that `P` holds for *all* lists, we can reason as follows:

- First, show that `P` is true of `l` when `l` is `Nil`.
- Then show that `P` is true of `l` when `l` is `cons n l'` for some number `n` and some smaller list `l'`, assuming that `P` is true for `l'`.

Since larger lists can only be built up from smaller ones, eventually reaching `Nil`, these two arguments together establish the truth of `P` for all lists `l`. Here's a concrete example:

```
app_assoc : (l1, l2, l3 : NatList) → ((l1 ++ l2) ++ l3) = (l1 ++ (l2 ++ l3))
app_assoc Nil l2 l3 = Refl
app_assoc (Cons n l1') l2 l3 =
  let inductiveHypothesis = app_assoc l1' l2 l3 in
    rewrite inductiveHypothesis in Refl
```

Edit

Notice that, as when doing induction on natural numbers, the as … clause provided to the induction tactic gives a name to the induction hypothesis corresponding to the smaller list l1' in the cons case. Once again, this Coq proof is not especially illuminating as a static written document – it is easy to see what's going on if you are reading the proof in an interactive Coq session and you can see the current goal and context at each point, but this state is not visible in the written-down parts of the Coq proof. So a natural-language proof – one written for human readers – will need to include more explicit signposts; in particular, it will help the reader stay oriented if we remind them exactly what the induction hypothesis is in the second case.

For comparison, here is an informal proof of the same theorem.

Theorem: For all lists `l1`, `l2`, and `l3`,

`(l1 ++ l2) ++ l3 = l1 ++ (l2 ++l3)`.

Proof: By induction on `l1`.

- First, suppose `l1 = []`. We must show

  `([] ++ l2) ++ l3 = [] ++ (l2 ++ l3)`,

  which follows directly from the definition of `++`.
- Next, suppose `l1 = n :: l1'`, with

`(l1' ++ l2) ++ l3 = l1' ++ (l2 ++ l3)`

(the induction hypothesis). We must show

`((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ (l2 ++ l3).`

By the definition of ++, this follows from

`n :: ((l1' ++ l2) ++ l 3) = n :: (l1' ++ (l2 ++ l3)),`

which is immediate from the induction hypothesis. □

3.1.1. *Reversing a List.* For a slightly more involved example of inductive proof over lists, suppose we use `app` to define a list-reversing function `rev`:

```
rev : (l : NatList) → NatList
rev Nil = Nil
rev (h :: t) = (rev t) ++ (h::[])

test_rev1 : rev (1::2::3::[]) = (3::2::1::[])
test_rev1 = Refl

test_rev2 : rev Nil = Nil
test_rev2 = Refl
```

3.1.2. *Properties of rev.* Now let's prove some theorems about our newly defined `rev`. For something a bit more challenging than what we've seen, let's prove that reversing a list does not change its length. Our first attempt gets stuck in the successor case…

```
rev_length_firsttry : (l : NatList) → length (rev l) = length l
rev_length_firsttry Nil = Refl
rev_length_firsttry (n :: l') =
-- Now we seem to be stuck: the goal is an equality involving `++`, but we don't
-- have any useful equations in either the immediate context or in the global
-- environment! We can make a little progress by using the IH to rewrite the
-- goal...
  let inductiveHypothesis = rev_length_firsttry l' in
    rewrite inductiveHypothesis in
-- ... but now we can't go any further.
      Refl
```

So let's take the equation relating ++ and `length` that would have enabled us to make progress and prove it as a separate lemma.

```
app_length : (l1, l2 : NatList) →
             length (l1 ++ l2) = (length l1) + (length l2)
app_length Nil l2 = Refl
app_length (n :: l1') l2 =
  let inductiveHypothesis = app_length l1' l2 in
    rewrite inductiveHypothesis in
      Refl
```

Note that, to make the lemma as general as possible, we quantify over *all* `NatLists`, not just those that result from an application of `rev`. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is easier to prove the more general property.

Now we can complete the original proof.

```
rev_length : (l : NatList) → length (rev l) = length l
rev_length Nil = Refl
rev_length (n :: l') =
  rewrite app_length (rev l') (n::[]) in
-- Prelude's version of `Induction.plus_comm`
    rewrite plusCommutative (length (rev l')) 1 in
      let inductiveHypothesis = rev_length l' in
        rewrite inductiveHypothesis in Refl
```

For comparison, here are informal proofs of these two theorems:

Theorem: For all lists `l1` and `l2`,

```
`length (l1 ++ l2) = length l1 + length l2`.
```

Proof: By induction on `l1`.

- First, suppose `l1 = []`. We must show

  ```
  length ([] ++ l2) = length [] + length l2,
  ```

  which follows directly from the definitions of `length` and `++`.

- Next, suppose `l1 = n :: l1'`, with

  ```
  length (l1' ++ l2) = length l1' + length l2.
  ```

  We must show

  ```
  length ((n :: l1') ++ l2) = length (n :: l1') + length l2).
  ```

  This follows directly from the definitions of `length` and `++` together with the induction hypothesis. □

Theorem: For all lists `l`, `length (rev l) = length l`.

Proof: By induction on `l`.

- First, suppose `l = []`. We must show

  ```
  length (rev []) = length [],
  ```

  which follows directly from the definitions of `length` and `rev`.

- Next, suppose l = n :: l' , with

  ```
  length (rev l') = length l'.
  ```

  We must show

  ```
  length (rev (n :: l')) = length (n :: l').
  ```

By the definition of `rev`, this follows from

```
length ((rev l') ++ [n]) = S (length l')
```

which, by the previous lemma, is the same as

```
length (rev l') + length [n] = S (length l').
```

This follows directly from the induction hypothesis and the definition of `length`. □

The style of these proofs is rather longwinded and pedantic. After the first few, we might find it easier to follow proofs that give fewer details (which can easily work out in our own minds or on scratch paper if necessary) and just highlight the non-obvious steps. In this more compressed style, the above proof might look like this:

Theorem: For all lists `l`, `length (rev l) = length l`.

Proof: First, observe that length `(l ++ [n]) = S (length l)` for any `l` (this follows by a straightforward induction on `l`). The main property again follows by induction on `l`, using the observation together with the induction hypothesis in the case where `l = n' :: l'`. □

Which style is preferable in a given situation depends on the sophistication of the expected audience and how similar the proof at hand is to ones that the audience will already be familiar with. The more pedantic style is a good default for our present purposes.

Edit: ‘apropos‘?

3.2. Search. We've seen that proofs can make use of other theorems we've already proved, e.g., using rewrite . But in order to refer to a theorem, we need to know its name! Indeed, it is often hard even to remember what theorems have been proven, much less what they are called.

Coq's Search command is quite helpful with this. Typing Search foo will cause Coq to display a list of all theorems involving foo . For example, try uncommenting the following line to see a list of theorems that we have proved about rev :

(* Search rev. *)

Keep Search in mind as you do the following exercises and throughout the rest of the book; it can save you a lot of time!

If you are using ProofGeneral, you can run Search with C-c C-a C-a. Pasting its response into your buffer can be accomplished with C-c C-;.

3.3. List Exercises, Part 1.

3.3.1. *Exercise: 3 stars (list_exercises).* More practice with lists:

```
app_nil_r : (l : NatList) → (l ++ []) = l
app_nil_r l = ?app_nil_r_rhs

rev_app_distr : (l1, l2 : NatList) → rev (l1 ++ l2) = (rev l2) ++ (rev l1)
rev_app_distr l1 l2 = ?rev_app_distr_rhs

rev_involutive : (l : NatList) → rev (rev l) = l
rev_involutive l = ?rev_involutive_rhs
```

There is a short solution to the next one. If you find yourself getting tangled up, step back and try to look for a simpler way.

```
app_assoc4 : (l1, l2, l3, l4 : NatList) →
             (l1 ++ (l2 ++ (l3 ++ l4))) = ((l1 ++ l2) ++ l3) ++ l4
app_assoc4 l1 l2 l3 l4 = ?app_assoc4_rhs
```

An exercise about your implementation of `nonzeros`:

```
nonzeros_app : (l1, l2 : NatList) →
                nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2)
nonzeros_app l1 l2 = ?nonzeros_app_rhs
```

□

3.3.2. *Exercise: 2 stars (beq_NatList).* Fill in the definition of `beq_NatList`, which compares lists of numbers for equality. Prove that `beq_NatList l l` yields `True` for every list `l`.

```
beq_NatList : (l1, l2 : NatList) → Bool
beq_NatList l1 l2 = ?beq_NatList_rhs

test_beq_NatList1 : beq_NatList Nil Nil = True
test_beq_NatList1 = ?test_beq_NatList1_rhs

test_beq_NatList2 : beq_NatList (1::2::3::[]) (1::2::3::[]) = True
test_beq_NatList2 = ?test_beq_NatList2_rhs

test_beq_NatList3 : beq_NatList (1::2::3::[]) (1::2::4::[]) = False
test_beq_NatList3 = ?test_beq_NatList3_rhs

beq_NatList_refl : (l : NatList) → True = beq_NatList l l
beq_NatList_refl l = ?beq_NatList_refl_rhs
```

□

3.4. List Exercises, Part 2.

3.4.1. *Exercise: 3 stars, advanced (bag_proofs).* Here are a couple of little theorems to prove about your definitions about bags above.

```
count_member_nonzero : (s : Bag) → leb 1 (count 1 (1 :: s)) = True
count_member_nonzero s = ?count_member_nonzero_rhs
```

The following lemma about `leb` might help you in the next proof.

```
ble_n_Sn : (n : Nat) → leb n (S n) = True
ble_n_Sn Z = Refl
ble_n_Sn (S k) =
  let inductiveHypothesis = ble_n_Sn k in
    rewrite inductiveHypothesis in Refl

remove_decreases_count : (s : Bag) →
                         leb (count 0 (remove_one 0 s)) (count 0 s) = True
remove_decreases_count s = ?remove_decreases_count_rhs
```

□

3.4.2. *Exercise: 3 stars, optional (bag_count_sum).* Write down an interesting theorem `bag_count_sum` about bags involving the functions `count` and `sum`, and prove it. (You may find that the difficulty of the proof depends on how you defined `count`!)

```
bag_count_sum : ?bag_count_sum
```

□

3.4.3. *Exercise: 4 stars, advanced (rev_injective).* Prove that the `rev` function is injective – that is,

```
rev_injective : (l1, l2 : NatList) → rev l1 = rev l2 → l1 = l2
rev_injective l1 l2 prf = ?rev_injective_rhs
```

(There is a hard way and an easy way to do this.)

□

4. Options

Suppose we want to write a function that returns the `n`th element of some list. If we give it type `Nat → NatList → Nat`, then we'll have to choose some number to return when the list is too short…

```
nth_bad : (l : NatList) → (n : Nat) → Nat
nth_bad Nil n = 42 -- arbitrary!
nth_bad (a :: l') n = case beq_nat n 0 of
                        True ⇒ a
                        False ⇒ nth_bad l' (pred n)
```

This solution is not so good: If `nth_bad` returns `42`, we can't tell whether that value actually appears on the input without further processing. A better alternative is to change the return type of `nth_bad` to include an error value as a possible outcome. We call this type `NatOption`.

```
data NatOption : Type where
  Some : Nat → NatOption
  None : NatOption
```

We can then change the above definition of `nth_bad` to return `None` when the list is too short and `Some a` when the list has enough members and `a` appears at position `n`. We call this new function `nth_error` to indicate that it may result in an error.

```
nth_error : (l : NatList) → (n : Nat) → NatOption
nth_error Nil n = None
nth_error (a :: l') n = case beq_nat n 0 of
                          True ⇒ Some a
                          False ⇒ nth_error l' (pred n)

test_nth_error1 : nth_error (4::5::6::7::[]) 0 = Some 4
test_nth_error1 = Refl

test_nth_error2 : nth_error (4::5::6::7::[]) 3 = Some 7
test_nth_error2 = Refl

test_nth_error3 : nth_error (4::5::6::7::[]) 9 = None
test_nth_error3 = Refl
```

This example is also an opportunity to introduce one more small feature of Idris programming language: conditional expressions…

```
nth_error' : (l : NatList) → (n : Nat) → NatOption
nth_error' Nil n = None
nth_error' (a :: l') n = if beq_nat n 0
                          then Some a
                          else nth_error' l' (pred n)
```

Edit this paragraph

Coq's conditionals are exactly like those found in any other language, with one small generalization. Since the boolean type is not built in, Coq actually supports conditional expressions over any inductively defined type with exactly two constructors. The guard is considered true if it evaluates to the first constructor in the Inductive definition and false if it evaluates to the second.

The function below pulls the `Nat` out of a `NatOption`, returning a supplied default in the `None` case.

```
option_elim : (d : Nat) → (o : NatOption) → Nat
option_elim d (Some k) = k
option_elim d None = d
```

4.0.1. *Exercise: 2 stars (hd_error).* Using the same idea, fix the `hd` function from earlier so we don't have to pass a default element for the `Nil` case.

```
hd_error : (l : NatList) → NatOption
hd_error l = ?hd_error_rhs

test_hd_error1 : hd_error [] = None
test_hd_error1 = ?test_hd_error1_rhs
```

```
test_hd_error2 : hd_error (1::[]) = Some 1
test_hd_error2 = ?test_hd_error2_rhs

test_hd_error3 : hd_error (5::6::[]) = Some 5
test_hd_error3 = ?test_hd_error3_rhs
```

□

4.0.2. *Exercise: 1 star, optional (option_elim_hd).* This exercise relates your new `hd_error` to the old `hd`.

```
option_elim_hd : (l : NatList) → (default : Nat) →
                 hd default l = option_elim default (hd_error l)
option_elim_hd l default = ?option_elim_hd_rhs
```

□

5. Partial Maps

As a final illustration of how data structures can be defined in Idris, here is a simple *partial map* data type, analogous to the map or dictionary data structures found in most programming languages.

First, we define a new inductive datatype `Id` to serve as the "keys" of our partial maps.

```
data Id : Type where
  MkId : Nat → Id
```

Internally, an `Id` is just a number. Introducing a separate type by wrapping each `Nat` with the tag `MkId` makes definitions more readable and gives us the flexibility to change representations later if we wish.

We'll also need an equality test for `Id`s:

```
beq_id : (x1, x2 : Id) → Bool
beq_id (MkId n1) (MkId n2) = beq_nat n1 n2
```

5.0.1. *Exercise: 1 star (beq_id_refl).*

```
beq_id_refl : (x : Id) → True = beq_id x x
beq_id_refl x = ?beq_id_refl_rhs
```

□

Now we define the type of partial maps:

```
namespace PartialMap

  data PartialMap : Type where
    Empty : PartialMap
    Record : Id → Nat → PartialMap → PartialMap
```

This declaration can be read: "There are two ways to construct a `PartialMap`: either using the constructor `Empty` to represent an empty partial map, or by applying the

constructor `Record` to a key, a value, and an existing `PartialMap` to construct a `PartialMap` with an additional key-to-value mapping."

The `update` function overrides the entry for a given key in a partial map (or adds a new entry if the given key is not already present).

```
update : (d : PartialMap) → (x : Id) → (value : Nat) → PartialMap
update d x value = Record x value d
```

Last, the `find` function searches a `PartialMap` for a given key. It returns `None` if the key was not found and `Some val` if the key was associated with `val`. If the same key is mapped to multiple values, `find` will return the first one it encounters.

```
find : (x : Id) → (d : PartialMap) → NatOption
find x Empty = None
find x (Record y v d') = if beq_id x y
                         then Some v
                         else find x d'
```

5.0.2. *Exercise: 1 star (update_eq).*

```
update_eq : (d : PartialMap) → (x : Id) → (v : Nat) →
            find x (update d x v) = Some v
update_eq d x v = ?update_eq_rhs
```

□

5.0.3. *Exercise: 1 star (update_neq).*

```
update_neq : (d : PartialMap) → (x, y : Id) → (o : Nat) →
             beq_id x y = False →
             find x (update d y o) = find x d
update_neq d x y o prf = ?update_neq_rhs
```

□

5.0.4. *Exercise: 2 stars (baz_num_elts).* Consider the following inductive definition:

```
data Baz : Type where
  Baz1 : Baz → Baz
  Baz2 : Baz → Bool → Baz
```

How *many* elements does the type `Baz` have? (Answer in English or the natural language of your choice.)

□

CHAPTER 5

Poly: Polymorphism and Higher-Order Functions

```
module Poly

import Basics

%hide Prelude.List.length
%hide Prelude.List.filter
%hide Prelude.List.partition
%hide Prelude.Functor.map
%hide Prelude.Nat.pred
%hide Basics.Playground2.plus

%access public export

%default total
```

1. Polymorphism

In this chapter we continue our development of basic concepts of functional programming. The critical new ideas are *polymorphism* (abstracting functions over the types of the data they manipulate) and *higher-order functions* (treating functions as data). We begin with polymorphism.

1.1. Polymorphic Lists. For the last couple of chapters, we've been working just with lists of numbers. Obviously, interesting programs also need to be able to manipulate lists with elements from other types – lists of strings, lists of booleans, lists of lists, etc. We *could* just define a new inductive datatype for each of these, for example…

```
data BoolList : Type where
  BoolNil : BoolList
  BoolCons : Bool → BoolList → BoolList
```

… but this would quickly become tedious, partly because we have to make up different constructor names for each datatype, but mostly because we would also need to define new versions of all our list manipulating functions (`length`, `rev`, etc.) for each new datatype definition.

To avoid all this repetition, Idris supports *polymorphic* inductive type definitions. For example, here is a *polymorphic list* datatype.

```
data List : (x : Type) → Type where
  Nil : List x
  Cons : x → List x → List x
```

(This type is already defined in Idris' standard library, but the `Cons` constructor is named `(::)`).

This is exactly like the definition of `NatList` from the previous chapter, except that the `Nat` argument to the `Cons` constructor has been replaced by an arbitrary type `x`, a binding for `x` has been added to the header, and the occurrences of `NatList` in the types of the constructors have been replaced by `List x`. (We can re-use the constructor names `Nil` and `Cons` because the earlier definition of `NatList` was inside of a `module` definition that is now out of scope.)

What sort of thing is `List` itself? One good way to think about it is that `List` is a *function* from `Type`s to inductive definitions; or, to put it another way, `List` is a function from `Type`s to `Type`s. For any particular type `x`, the type `List x` is an inductively defined set of lists whose elements are of type `x`.

With this definition, when we use the constructors `Nil` and `Cons` to build lists, we need to tell Idris the type of the elements in the lists we are building – that is, `Nil` and `Cons` are now *polymorphic constructors*. Observe the types of these constructors:

```
λΠ> :t Nil
-- Prelude.List.Nil : List elem
λΠ> :t (::)
-- Prelude.List.(::) : elem → List elem → List elem
```

How to edit these 3 paragraphs? Implicits are defined later in this chapter, and Idris doesn't require type parameters to constructors

(Side note on notation: In .v files, the "forall" quantifier is spelled out in letters. In the generated HTML files and in the way various IDEs show .v files (with certain settings of their display controls), is usually typeset as the usual mathematical "upside down A," but you'll still see the spelled-out "forall" in a few places. This is just a quirk of typesetting: there is no difference in meaning.)

The " X" in these types can be read as an additional argument to the constructors that determines the expected types of the arguments that follow. When `Nil` and `Cons` are used, these arguments are supplied in the same way as the others. For example, the list containing `2` and `1` is written like this:

Check (cons nat 2 (cons nat 1 (nil nat))).

(We've written `Nil` and `Cons` explicitly here because we haven't yet defined the `[]` and :: notations for the new version of lists. We'll do that in a bit.)

We can now go back and make polymorphic versions of all the list-processing functions that we wrote before. Here is `repeat`, for example:

```
repeat : (x_ty : Type) → (x : x_ty) → (count : Nat) → List x_ty
repeat x_ty x Z = Nil
repeat x_ty x (S count') = x :: repeat x_ty x count'
```

As with `Nil` and `Cons`, we can use `repeat` by applying it first to a type and then to its list argument:

```
test_repeat1 : repeat Nat 4 2 = 4 :: (4 :: Nil)
test_repeat1 = Refl
```

To use `repeat` to build other kinds of lists, we simply instantiate it with an appropriate type parameter:

```
test_repeat2 : repeat Bool False 1 = False :: Nil
test_repeat2 = Refl
```

Explain implicits and {x=foo} syntax first? Move after the "Supplying Type Arguments Explicitly" section?

1.1.1. *Exercise: 2 starsM (mumble_grumble).*

```
namespace MumbleGrumble
```

Consider the following two inductively defined types.

```
  data Mumble : Type where
    A : Mumble
    B : Mumble → Nat → Mumble
    C : Mumble

  data Grumble : (x : Type) → Type where
    D : Mumble → Grumble x
    E : x → Grumble x
```

Which of the following are well-typed elements of `Grumble x` for some type `x`?

- `D (B A 5)`
- `D (B A 5) {x=Mumble}`
- `D (B A 5) {x=Bool}`
- `E True {x=Bool}`
- `E (B C 0) {x=Mumble}`
- `E (B C 0) {x=Bool}`
- `C`

```
-- FILL IN HERE
```

□

Merge 3 following sections into one about Idris implicits? Mention the lowercase/uppercase distinction.

1.1.2. *Type Annotation Inference.*

This has already happened earlier at 'repeat', delete most

Let's write the definition of `repeat` again, but this time we won't specify the types of any of the arguments. Will Idris still accept it?

Fixpoint repeat' X x count : list X := match count with | 0 nil X | S count' cons X x (repeat' X x count') end.

Indeed it will. Let's see what type Idris has assigned to `repeat'`:

Check repeat'. (* ===> forall X : Type, X -> nat -> list X) *Check repeat.* (===> forall X : Type, X -> nat -> list X *)

It has exactly the same type type as `repeat`. Idris was able to use *type inference* to deduce what the types of `X`, `x`, and `count` must be, based on how they are used. For example, since `X` is used as an argument to `Cons`, it must be a `Type`, since `Cons` expects a `Type` as its first argument; matching `count` with `Z` and `S` means it must be a `Nat`; and so on.

This powerful facility means we don't always have to write explicit type annotations everywhere, although explicit type annotations are still quite useful as documentation and sanity checks, so we will continue to use them most of the time. You should try to find a balance in your own code between too many type annotations (which can clutter and distract) and too few (which forces readers to perform type inference in their heads in order to understand your code).

We should mention the _ parameters but it won't work like

1.1.3. *Type Argument Synthesis.*

To use a polymorphic function, we need to pass it one or more types in addition to its other arguments. For example, the recursive call in the body of the `repeat` function above must pass along the type `x_ty`. But since the second argument to `repeat` is an element of `x_ty`, it seems entirely obvious that the first argument can only be `x_ty` — why should we have to write it explicitly?

Fortunately, Idris permits us to avoid this kind of redundancy. In place of any type argument we can write the "implicit argument" _, which can be read as "Please try to figure out for yourself what belongs here." More precisely, when Idris encounters a _, it will attempt to *unify* all locally available information – the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears – to determine what concrete type should replace the _.

This may sound similar to type annotation inference – indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like

```
repeat' X x count : list X :=
```

we can also replace the types with _

```
repeat' (X : _) (x : _) (count : _) : list X :=
```

to tell Idris to attempt to infer the missing information.

Using implicit arguments, the `count` function can be written like this:

Fixpoint repeat'‘X x count : list X := match count with | 0 nil _ | S count' cons _ x (repeat'‘_ x count') end.

In this instance, we don't save much by writing _ instead of x. But in many cases the difference in both keystrokes and readability is nontrivial. For example, suppose we want to write down a list containing the numbers 1, 2, and 3. Instead of writing this…

Definition list123 := cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).

…we can use argument synthesis to write this:

Definition list123' := cons _ 1 (cons _ 2 (cons _ 3 (nil _))).

1.1.4. *Implicit Arguments.* We can go further and even avoid writing _'s in most cases by telling Idris *always* to infer the type argument(s) of a given function. The Arguments directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern _.)

Arguments nil {X}. Arguments cons {X} _ _. Arguments repeat {X} x count.

Now, we don't have to supply type arguments at all:

Definition list123'' := cons 1 (cons 2 (cons 3 nil)).

Alternatively, we can declare an argument to be implicit when defining the function itself, by surrounding it in curly braces instead of parens. For example:

```
repeat' : {x_ty : Type} → (x : x_ty) → (count : Nat) → List x_ty
repeat' x Z = Nil
repeat' x (S count') = x :: repeat' x count'
```

(Note that we didn't even have to provide a type argument to the recursive call to `repeat'`; indeed, it would be invalid to provide one!)

We will use the latter style whenever possible, but we will continue to use explicit declarations in data types. The reason for this is that marking the parameter of an inductive type as implicit causes it to become implicit for the type itself, not just for its constructors. For instance, consider the following alternative definition of the `List` type:

```
data List' : {x : Type} → Type where
  Nil' : List'
  Cons' : x → List' → List'
```

Because `x` is declared as implicit for the *entire* inductive definition including `List'` itself, we now have to write just `List'` whether we are talking about lists of numbers or booleans or anything else, rather than `List' Nat` or `List' Bool` or whatever; this is a step too far.

Added the implicit inference explanation here

There's another step towards conciseness that we can take in Idris – drop the implicit argument completely in function definitions! Idris will automatically insert them for us when it encounters unknown variables. *Note that by convention this will only happen for variables starting on a lowercase letter.*

```
repeat'' : (x : x_ty) → (count : Nat) → List x_ty
repeat'' x Z = Nil
repeat'' x (S count') = x :: repeat'' x count'
```

Let's finish by re-implementing a few other standard list functions on our new polymorphic lists…

```
app : (l1, l2 : List x) → List x
app Nil l2 = l2
app (h::t) l2 = h :: app t l2

rev : (l: List x) → List x
rev [] = []
rev (h::t) = app (rev t) (h::Nil)

length : (l : List x) → Nat
length [] = Z
length (_::l') = S (length l')

test_rev1 : rev (1::2::[]) = 2::1::[]
test_rev1 = Refl

test_rev2 : rev (True::[]) = True::[]
test_rev2 = Refl

test_length1 : length (1::2::3::[]) = 3
test_length1 = Refl
```

1.1.5. *Supplying Type Arguments Explicitly.* One small problem with declaring arguments implicit is that, occasionally, Idris does not have enough local information to determine a type argument; in such cases, we need to tell Idris that we want to give the argument explicitly just this time. For example, suppose we write this:

```
λΠ> :let mynil = Nil
-- (input):Can't infer argument elem to []
```

Here, Idris gives us an error because it doesn't know what type argument to supply to `Nil`. We can help it by providing an explicit type declaration via `the` function (so that Idris has more information available when it gets to the "application" of `Nil`):

```
λΠ> :let mynil = the (List Nat) Nil
```

Alternatively, we can force the implicit arguments to be explicit by supplying them as arguments in curly braces.

```
λΠ> :let mynil' = Nil {elem=Nat}
```

Describe here how to bring variables from the type into definition scope via implicits?

Explain that Idris has built-in notation for lists instead?

Using argument synthesis and implicit arguments, we can define convenient notation for lists, as before. Since we have made the constructor type arguments implicit, Coq will know to automatically infer these when we use the notations.

Notation "x :: y" := (cons x y) (at level 60, right associativity). Notation "[ ]" := nil. Notation "[x ; .. ; y]" := (cons x .. (cons y []) ..). Notation "x ++ y" := (app x y) (at level 60, right associativity).

Now lists can be written just the way we'd hope:

```
list123''' : List Nat
list123''' = [1, 2, 3]
```

1.1.6. *Exercise: 2 stars, optional (poly_exercises).* Here are a few simple exercises, just like ones in the `Lists` chapter, for practice with polymorphism. Complete the proofs below.

```
app_nil_r : (l : List x) → l ++ [] = l
app_nil_r l = ?app_nil_r_rhs

app_assoc : (l, m, n : List a) → l ++ m ++ n = (l ++ m) ++ n
app_assoc l m n = ?app_assoc_rhs

app_length : (l1, l2 : List x) → length (l1 ++ l2) = length l1 + length l2
app_length l1 l2 = ?app_length_rhs
```

□

1.1.7. *Exercise: 2 stars, optional (more_poly_exercises).* Here are some slightly more interesting ones…

```
rev_app_distr : (l1, l2 : List x) → rev (l1 ++ l2) = rev l2 ++ rev l1
rev_app_distr l1 l2 = ?rev_app_distr_rhs

rev_involutive : (l : List x) → rev (rev l) = l
rev_involutive l = ?rev_involutive_rhs
```

□

1.2. Polymorphic Pairs. Following the same pattern, the type definition we gave in the last chapter for pairs of numbers can be generalized to *polymorphic pairs*, often called *products*:

```
data Prod : (x, y : Type) → Type where
  PPair : x → y → Prod x y
```

As with lists, we make the type arguments implicit and define the familiar concrete notation.

```
syntax "(" [x] "," [y] ")" = PPair x y
```

We can also use the `syntax` mechanism to define the standard notation for product *types*:

```
syntax [x_ty] "*" [y_ty] = Prod x_ty y_ty
```

(The annotation : type_scope tells Coq that this abbreviation should only be used when parsing types. This avoids a clash with the multiplication symbol.)

It is easy at first to get `(x,y)` and `x_ty*y_ty` confused. Remember that `(x,y)` is a value built from two other values, while `x_ty*y_ty` is a type built from two other types. If `x` has type `x` and `y` has type `y`, then `(x,y)` has type `x_ty*y_ty`.

The first and second projection functions now look pretty much as they would in any functional programming language.

```
fst : (p : x*y) → x
fst (x,y) = x

snd : (p : x*y) → y
snd (x,y) = y
```

The following function takes two lists and combines them into a list of pairs. In functional languages, it is usually called `zip` (though the Coq's standard library calls it `combine`).

```
zip : (lx : List x) → (ly : List y) → List (x*y)
zip [] _ = []
zip _ [] = []
zip (x::tx) (y::ty) = (x,y) :: zip tx ty
```

1.2.1. *Exercise: 1 star, optionalM (combine_checks).* Try answering the following questions on paper and checking your answers in Idris:

- What is the type of `zip` (i.e., what does `:t zip` print?)
- What does

```
combine [1,2] [False,False,True,True]
```

print?

□

1.2.2. *Exercise: 2 stars, recommended (split).* The function `split` is the right inverse of `zip`: it takes a list of pairs and returns a pair of lists. In many functional languages, it is called `unzip`.

Fill in the definition of `split` below. Make sure it passes the given unit test.

```
split : (l : List (x*y)) → (List x) * (List y)
split l = ?split_rhs

test_split: split [(1,False),(2,False)] = ([1,2],[False,False])
test_split = ?test_split_rhs
```

□

1.2.3. *Polymorphic Options.* One last polymorphic type for now: *polymorphic options*, which generalize `NatOption` from the previous chapter:

```
data Option : (x : Type) → Type where
 Some : x → Option x
 None : Option x
```

In Idris' standard library this type is called `Maybe`, with constructors `Just x` and `Nothing`.

We can now rewrite the `nth_error` function so that it works with any type of lists.

```
nth_error : (l : List x) → (n : Nat) → Option x
nth_error [] n = None
nth_error (a::l') n = if beq_nat n 0
                        then Some a
                        else nth_error l' (pred n)

test_nth_error1 : nth_error [4,5,6,7] 0 = Some 4
test_nth_error1 = Refl

test_nth_error2 : nth_error [[1],[2]] 1 = Some [2]
test_nth_error2 = Refl

test_nth_error3 : nth_error [True] 2 = None
test_nth_error3 = Refl
```

1.2.4. *Exercise: 1 star, optional (hd_error_poly).* Complete the definition of a polymorphic version of the `hd_error` function from the last chapter. Be sure that it passes the unit tests below.

```
hd_error : (l : List x) → Option x
hd_error l = ?hd_error_rhs

test_hd_error1 : hd_error [1,2] = Some 1
test_hd_error1 = ?test_hd_error1_rhs

test_hd_error2 : hd_error [[1],[2]] = Some [1]
test_hd_error2 = ?test_hd_error2_rhs
```

□

2. Functions as Data

Like many other modern programming languages – including all functional languages (ML, Haskell, Scheme, Scala, Clojure etc.) – Idris treats functions as first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

2.1. Higher-Order Functions. Functions that manipulate other functions are often called *higher-order* functions. Here's a simple one:

```
doit3times : (f: x → x) → (n : x) → x
doit3times f n = f (f (f n))
```

The argument `f` here is itself a function (from `x` to `x`); the body of `doit3times` applies `f` three times to some value `n`.

```
λ∏> :t doit3times
-- doit3times : (x → x) → x → x

test_doit3times : doit3times Numbers.minusTwo 9 = 3
test_doit3times = Refl

test_doit3times' : doit3times Booleans.negb True = False
test_doit3times' = Refl
```

2.2. Filter. Here is a more useful higher-order function, taking a list of `xs` and a *predicate* on `x` (a function from `x` to `Bool`) and "filtering" the list, returning a new list containing just those elements for which the predicate returns `True`.

```
filter : (test : x → Bool) → (l: List x) → List x
filter test [] = []
filter test (h::t) = if test h
                     then h :: (filter test t)
                     else filter test t
```

For example, if we apply `filter` to the predicate `evenb` and a list of numbers `l`, it returns a list containing just the even members of `l`.

```
test_filter1 : filter Numbers.evenb [1,2,3,4] = [2,4]
test_filter1 = Refl

length_is_1 : (l : List x) → Bool
length_is_1 l = beq_nat (length l) 1
```

Why doesn't this work without `{x=Nat}`? Apparently it even works with `{x=_}`!

```
test_filter2 : filter (length_is_1 {x=Nat})
                      [ [1,2], [3], [4], [5,6,7], [], [8] ]
                    = [        [3], [4],              [8] ]
test_filter2 = Refl
```

We can use `filter` to give a concise version of the `countoddmembers` function from the `Lists` chapter.

```
countoddmembers' : (l: List Nat) → Nat
countoddmembers' l = length (filter Numbers.oddb l)

test_countoddmembers'1 : countoddmembers' [1,0,3,1,4,5] = 4
test_countoddmembers'1 = Refl
```

```
test_countoddmembers'2 : countoddmembers' [0,2,4] = 0
test_countoddmembers'2 = Refl

test_countoddmembers'3 : countoddmembers' Nil = 0
test_countoddmembers'3 = Refl
```

2.3. Anonymous Functions. It is arguably a little sad, in the example just above, to be forced to define the function `length_is_1` and give it a name just to be able to pass it as an argument to `filter`, since we will probably never use it again. Moreover, this is not an isolated example: when using higher-order functions, we often want to pass as arguments "one-off" functions that we will never use again; having to give each of these functions a name would be tedious.

Fortunately, there is a better way. We can construct a function "on the fly" without declaring it at the top level or giving it a name.

Can't use '*' here due to the interference from our tuple sugar

```
test_anon_fun' : doit3times (\n ⇒ mult n n) 2 = 256
test_anon_fun' = Refl
```

The expression `\n ⇒ mult n n` can be read as "the function that, given a number `n`, yields `n * n`."

Here is the `filter` example, rewritten to use an anonymous function.

```
test_filter2' : filter (\l ⇒ beq_nat (length l) 1)
                       [ [1,2], [3], [4], [5,6,7], [], [8] ]
                     = [        [3], [4],               [8] ]
test_filter2' = Refl
```

2.3.1. *Exercise: 2 stars (filter_even_gt7).* Use `filter` (instead of function definition) to write an Idris function `filter_even_gt7` that takes a list of natural numbers as input and returns a list of just those that are even and greater than 7.

```
filter_even_gt7 : (l : List Nat) → List Nat
filter_even_gt7 l = ?filter_even_gt7_rhs

test_filter_even_gt7_1 : filter_even_gt7 [1,2,6,9,10,3,12,8] = [10,12,8]
test_filter_even_gt7_1 = ?test_filter_even_gt7_1_rhs

test_filter_even_gt7_2 : filter_even_gt7 [5,2,6,19,129] = []
test_filter_even_gt7_2 = ?test_filter_even_gt7_2_rhs
```

□

2.3.2. *Exercise: 3 stars (partition).* Use `filter` to write an Idris function `partition`:

```
partition : (test : x → Bool) → (l : List x) → (List x) * (List x)
partition f xs = ?partition_rhs
```

Given a set `x`, a test function of type `x → Bool` and a `List x`, `partition` should return a pair of lists. The first member of the pair is the sublist of the original

list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

```
test_partition1 : partition Numbers.oddb [1,2,3,4,5] = ([1,3,5], [2,4])
test_partition1 = ?test_partition1_rhs

test_partition2 : partition (\x ⇒ false) [5,9,0] = (([], [5,9,0]))
test_partition2 = ?test_partition2_rhs
```

□

2.4. Map. Another handy higher-order function is called `map`.

```
map : (f : x → y) → (l : List x) → List y
map f [] = []
map f (h::t) = (f h) :: map f t
```

It takes a function `f` and a list `l = [n1, n2, n3, ...]` and returns the list `[f n1, f n2, f n3, ...]`, where `f` has been applied to each element of `l` in turn. For example:

```
test_map1 : map (\x ⇒ plus 3 x) [2,0,2] = [5,3,5]
test_map1 = Refl
```

The element types of the input and output lists need not be the same, since `map` takes *two* type arguments, `x` and `y`; it can thus be applied to a list of numbers and a function from numbers to booleans to yield a list of booleans:

```
test_map2 : map Numbers.oddb [2,1,2,5] = [False,True,False,True]
test_map2 = Refl
```

It can even be applied to a list of numbers and a function from numbers to *lists* of booleans to yield a *list of lists* of booleans:

```
test_map3 : map (\n ⇒ [evenb n, oddb n]) [2,1,2,5]
                  = [[True,False],[False,True],[True,False],[False,True]]
test_map3 = Refl
```

2.4.1. *Exercise: 3 stars (map_rev).* Show that `map` and `rev` commute. You may need to define an auxiliary lemma.

```
map_rev : (f : x → y) → (l : List x) → map f (rev l) = rev (map f l)
map_rev f l = ?map_rev_rhs
```

□

2.4.2. *Exercise: 2 stars, recommended (flat_map).* The function `map` maps a `List x` to a `List y` using a function of type `x → y`. We can define a similar function, `flat_map`, which maps a `List x` to a `List y` using a function `f` of type `X → List y`. Your definition should work by 'flattening' the results of `f`, like so:

```
flat_map (\n ⇒ [n,n+1,n+2]) [1,5,10] = [1,2,3, 5,6,7, 10,11,12]
```

```
flat_map : (f : x → List y) → (l : List x) → List y
flat_map f l = ?flat_map_rhs

test_flat_map1 : flat_map (\n ⇒ [n,n,n]) [1,5,4] = [1,1,1, 5,5,5, 4,4,4]
test_flat_map1 = ?test_flat_map1_rhs
```

□

Lists are not the only inductive type that we can write a `map` function for. Here is the definition of `map` for the `Option` type:

```
option_map : (f : x → y) → (xo : Option x) → Option y
option_map f None = None
option_map f (Some x) = Some (f x)
```

2.4.3. *Exercise: 2 stars, optional (implicit_args).* The definitions and uses of `filter` and `map` use implicit arguments in many places. Add explicit type parameters where necessary and use Idris to check that you've done so correctly. (This exercise is not to be turned in; it is probably easiest to do it on a copy of this file that you can throw away afterwards.)

□

2.5. Fold. An even more powerful higher-order function is called `fold`. This function is the inspiration for the "`reduce`" operation that lies at the heart of Google's map/reduce distributed programming framework.

```
fold : (f : x → y → y) → (l : List x) → (b : y) → y
fold f [] b = b
fold f (h::t) b = f h (fold f t b)
```

Intuitively, the behavior of the `fold` operation is to insert a given binary operator `f` between every pair of elements in a given list. For example, `fold plus [1;2;3;4]` intuitively means `1+2+3+4`. To make this precise, we also need a "starting element" that serves as the initial second input to `f`. So, for example,

```
fold plus [1,2,3,4] 0
```

yields

```
1 + (2 + (3 + (4 + 0)))
```

Some more examples:

```
λΠ> :t fold andb
-- fold andb : List Bool → Bool → Bool

fold_example1 : fold Nat.mult [1,2,3,4] 1 = 24
fold_example1 = Refl

fold_example2 : fold Booleans.andb [True,True,False,True] True = False
fold_example2 = Refl

fold_example3 : fold (++) [[1],[],[2,3],[4]] [] = [1,2,3,4]
fold_example3 = Refl
```

2.5.1. *Exercise: 1 star, advancedM (fold_types_different).* Observe that the type of `fold` is parameterized by *two* type variables, `x` and `y`, and the parameter `f` is a binary operator that takes an `x` and a `y` and returns a `y`. Can you think of a situation where it would be useful for `x` and `y` to be different?

– FILL IN HERE

□

2.6. Functions That Construct Functions. Most of the higher-order functions we have talked about so far take functions as arguments. Let's look at some examples that involve *returning* functions as the results of other functions. To begin, here is a function that takes a value `x` (drawn from some type `x`) and returns a function from `Nat` to `x` that yields `x` whenever it is called, ignoring its `Nat` argument.

```
constfun : (x : x_ty) → Nat → x_ty
constfun x = \k ⇒ x

ftrue : Nat → Bool
ftrue = constfun True

constfun_example1 : ftrue 0 = True
constfun_example1 = Refl

constfun_example2 : (constfun 5) 99 = 5
constfun_example2 = Refl
```

In fact, the multiple-argument functions we have already seen are also examples of passing functions as data. To see why, recall the type of `plus`.

```
λΠ> :t plus
Prelude.Nat.plus : Nat → Nat → Nat
```

Each → in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of `plus` is really a shorthand for `Nat → (Nat → Nat)` – i.e., it can be read as saying that "`plus` is a one-argument function that takes a `Nat` and returns a one-argument function that takes another `Nat` and returns a `Nat`." In the examples above, we have always applied `plus` to both of its arguments at once, but if we like we can supply just the first. This is called *partial application*.

```
plus3 : Nat → Nat
plus3 = plus 3

λΠ> :t plus3

test_plus3 : plus3 4 = 7
test_plus3 = Refl
```

This is apparently a bug in Idris, https://github.com/idris-lang/Idris-dev/issues/3908

```
-- test_plus3' : doit3times plus3 0 = 9
-- test_plus3' = Refl
```

```
test_plus3'' : doit3times (plus 3) 0 = 9
test_plus3'' = Refl
```

3. Additional Exercises

```
namespace Exercises
```

3.0.1. *Exercise: 2 stars (fold_length).* Many common functions on lists can be implemented in terms of `fold`. For example, here is an alternative definition of `length`:

```
  fold_length : (l : List x) → Nat
  fold_length l = fold (\_, n ⇒ S n) l 0

  test_fold_length1 : fold_length [4,7,0] = 3
  test_fold_length1 = Refl
```

Prove the correctness of `fold_length`.

```
  fold_length_correct : (l : List x) → fold_length l = length l
  fold_length_correct l = ?fold_length_correct_rhs
```

□

3.0.2. *Exercise: 3 starsM (fold_map).* We can also define `map` in terms of fold. Finish `fold_map` below.

```
  fold_map : (f : x → y) → (l : List x) → List y
  fold_map f l = ?fold_map_rhs
```

Write down a theorem `fold_map_correct` in Idris stating that `fold_map` is correct, and prove it.

```
  fold_map_correct : ?fold_map_correct
```

□

3.0.3. *Exercise: 2 stars, advanced (currying).* In Idris, a function `f: a → b → c` really has the type `a → (b → c)`. That is, if you give `f` a value of type `a`, it will give you function `f' : b → c`. If you then give `f'` a value of type `b`, it will return a value of type `c`. This allows for partial application, as in `plus3`. Processing a list of arguments with functions that return functions is called *currying*, in honor of the logician Haskell Curry.

Conversely, we can reinterpret the type `a → b → c` as `(a * b) → c`. This is called *uncurrying*. With an uncurried binary function, both arguments must be given at once as a pair; there is no partial application.

We can define currying as follows:

```
  prod_curry : (f : (x * y) → z) → (x_val : x) → (y_val : y) → z
  prod_curry f x_val y_val = f (x_val, y_val)
```

As an exercise, define its inverse, `prod_uncurry`. Then prove the theorems below to show that the two are inverses.

```
prod_uncurry : (f : x → y → z) → (p : x * y) → z
prod_uncurry f p = ?prod_uncurry_rhs
```

As a (trivial) example of the usefulness of currying, we can use it to shorten one of the examples that we saw above:

Not sure what are they shortening here

```
test_map2' : map (\x ⇒ plus 3 x) [2,0,2] = [5,3,5]
test_map2' = Refl
```

Didn't we just write out these types explicitly?

Thought exercise: before running the following commands, can you calculate the types of `prod_curry` and `prod_uncurry`?

```
λ∏> :t prod_curry
λ∏> :t prod_uncurry
```

```
uncurry_curry : (f : x → y → z) → (x_val : x) → (y_val : y) →
                prod_curry (prod_uncurry f) x_val y_val = f x_val y_val
uncurry_curry f x_val y_val = ?uncurry_curry_rhs

curry_uncurry : (f : (x * y) → z) → (p : x * y) →
                prod_uncurry (prod_curry f) p = f p
curry_uncurry f p = ?curry_uncurry_rhs
```

□

3.0.4. *Exercise: 2 stars, advancedM (nth_error_informal).* Recall the definition of the `nth_error` function:

```
nth_error : (l : List x) → (n : Nat) → Option x
nth_error [] n = None
nth_error (a::l') n = if beq_nat n 0
                        then Some a
                        else nth_error l' (pred n)
```

Write an informal proof of the following theorem:

```
n → l → length l = n → nth_error l n = None
```

– FILL IN HERE

□

3.0.5. *Exercise: 4 stars, advanced (church_numerals).* This exercise explores an alternative way of defining natural numbers, using the so-called *Church numerals*, named after mathematician Alonzo Church. We can represent a natural number `n` as a function that takes a function `f` as a parameter and returns `f` iterated `n` times.

```
namespace Church
```

```
Nat' : {x : Type} → Type
Nat' {x} = (x → x) → x → x
```

Let's see how to write some numbers with this notation. Iterating a function once should be the same as just applying it. Thus:

```
one : Nat'
one f x = f x
```

Similarly, `two` should apply `f` twice to its argument:

```
two : Nat'
two f x = f (f x)
```

Defining `zero` is somewhat trickier: how can we "apply a function zero times"? The answer is actually simple: just return the argument untouched.

```
zero : Nat'
zero f x = x
```

More generally, a number `n` can be written as `\f, x ⇒ f (f ... (f x) ...)`, with `n` occurrences of `f`. Notice in particular how the `doit3times` function we've defined previously is actually just the Church representation of `3`.

```
three : Nat'
three = doit3times
```

Complete the definitions of the following functions. Make sure that the corresponding unit tests pass by proving them with `Refl`.

Successor of a natural number:

```
succ' : (n : Nat' {x}) → Nat' {x}
succ' n = ?succ'_rhs

succ'_1 : succ' zero = one
succ'_1 = ?succ_1_rhs

succ'_2 : succ' one = two
succ'_2 = ?succ_2_rhs

succ'_3 : succ' two = three
succ'_3 = ?succ'_3_rhs
```

Addition of two natural numbers:

```
plus' : (n, m : Nat' {x}) → Nat' {x}
plus' n m = ?plus'_rhs

plus'_1 : plus' zero one = one
plus'_1 = ?plus'_1_rhs

plus'_2 : plus' two three = plus' three two
plus'_2 = ?plus'_2_rhs
```

```
plus'_3 : plus' (plus' two two) three = plus' one (plus' three three)
plus'_3 = ?plus'_3_rhs
```

Multiplication:

```
mult' : (n, m : Nat' {x}) → Nat' {x}
mult' n m = ?mult'_rhs

mult'_1 : mult' one one = one
mult'_1 = ?mult'_1_rhs

mult'_2 : mult' zero (plus' three three) = zero
mult'_2 = ?mult'_2_rhs

mult'_3 : mult' two three = plus' three three
mult'_3 = ?mult'_3_rhs
```

Exponentiation:

Edit the hint. Can't make it work with 'exp' : (n, m : Nat' x) -> Nat' x'.

(Hint: Polymorphism plays a crucial role here. However, choosing the right type to iterate over can be tricky. If you hit a "Universe inconsistency" error, try iterating over a different type: `Nat'` itself is usually problematic.)

```
exp' : (n : Nat' {x}) → (m : Nat' {x=x→x}) → Nat' {x}
exp' n m = ?exp'_rhs
```

This won't typecheck under this signature of 'exp' because of 2 instances of 'two'

```
-- exp'_1 : exp' two two = plus' two two
-- exp'_1 = ?exp'_1_rhs

exp'_2 : exp' three two = plus' (mult' two (mult' two two)) one
exp'_2 = ?exp'_2_rhs

exp'_3 : exp' three zero = one
exp'_3 = ?exp'_3_rhs
```

□

Glossary

algebraic data type: . 9 define

computation rule: . 17 define

expression: . 16 define

first-class: . 9 define

fully certified: . 12 define

function type: . 15 define

idris-add-clause: (idris-add-clause PROOF)

Add clauses to the declaration at point. 11, 12

idris-case-split: (idris-case-split)

Case split the pattern variable at point. 11

idris-load-file: (idris-load-file &optional SET-LINE)

Pass the current buffer's file to the inferior Idris process.

A prefix argument restricts loading to the current line. 11

induction: . 21 define

inductive rule: . 15 define

module system: . 15 define

pattern matching: . 9 define

structural recursion: . 24 define

syntax: . 19 define

tactic: . 10 define