Functional Programming in Lean

by David Thrane Christiansen

This is a free book on using Lean 4 as a programming language. All code samples are tested with Lean 4 release `nightly-2023-05-22`.

Release history

May, 2023

The book is now complete! Compared to the April pre-release, many small details have been improved and minor mistakes have been fixed.

April, 2023

This release adds an interlude on writing proofs with tactics as well as a final chapter that combines discussion of performance and cost models with proofs of termination and program equivalence. This is the last release prior to the final release.

March, 2023

This release adds a chapter on programming with dependent types and indexed families.

January, 2023

This release adds a chapter on monad transformers that includes a description of the imperative features that are available in `do`-notation.

December, 2022

This release adds a chapter on applicative functors that additionally describes structures and type classes in more detail. This is accompanied with improvements to the description of monads. The December 2022 release was delayed until January 2023 due to winter holidays.

November, 2022

This release adds a chapter on programming with monads. Additionally, the example of using JSON in the coercions section has been updated to include the complete code.

October, 2022

This release completes the chapter on type classes. In addition, a short interlude introducing propositions, proofs, and tactics has been added just before the chapter on type classes, because a small amount of familiarity with the concepts helps to understand some of the standard library type classes.

September, 2022

This release adds the first half of a chapter on type classes, which are Lean's mechanism for overloading operators and an important means of organizing code and structuring libraries. Additionally, the second chapter has been updated to account for changes in Lean's stream API.

August, 2022

This third public release adds a second chapter, which describes compiling and running programs along with Lean's model for side effects.

July, 2022

The second public release completes the first chapter.

June, 2022

This was the first public release, consisting of an introduction and part of the first chapter.

About the Author

David Thrane Christiansen has been using functional languages for twenty years, and dependent types for ten. Together with Daniel P. Friedman, he wrote *The Little Typer*, an introduction to the key ideas of dependent type theory. He has a Ph.D. from the IT University of Copenhagen. During his studies, he was a major contributor to the first version of the Idris language. Since leaving academia, he has worked at Galois in Portland, Oregon and Deon Digital in Copenhagen, Denmark. At the time of writing, he is the Executive Director of the Haskell Foundation.

License

Lean is an interactive theorem prover developed at Microsoft Research, based on dependent type theory. Dependent type theory unites the worlds of programs and proofs; thus, Lean is also a programming language. Lean takes its dual nature seriously, and it is designed to be suitable for use as a general-purpose programming language—Lean is even implemented in itself. This book is about writing programs in Lean.

When viewed as a programming language, Lean is a strict pure functional language with dependent types. A large part of learning to program with Lean consists of learning how each of these attributes affects the way programs are written, and how to think like a functional programmer. *Strictness* means that function calls in Lean work similarly to the way they do in most languages: the arguments are fully computed before the function's body begins running. *Purity* means that Lean programs cannot have side effects such as modifying locations in memory, sending emails, or deleting files without the program's type saying so. Lean is a *functional* language in the sense that functions are first-class values like any other and that the execution model is inspired by the evaluation of mathematical expressions. *Dependent types*, which are the most unusual feature of Lean, make types into a first-class part of the language, allowing types to contain programs and programs to compute types.

This book is intended for programmers who want to learn Lean, but who have not necessarily used a functional programming language before. Familiarity with functional languages such as Haskell, OCaml, or F# is not required. On the other hand, this book does assume knowledge of concepts like loops, functions, and data structures that are common to most programming languages. While this book is intended to be a good first book on functional programming, it is not a good first book on programming in general.

Mathematicians who are using Lean as a proof assistant will likely need to write custom proof automation tools at some point. This book is also for them. As these tools become more sophisticated, they begin to resemble programs in functional languages, but most working mathematicians are trained in languages like Python and Mathematica. This book can help bridge the gap, empowering more mathematicians to write maintainable and understandable proof automation tools.

This book is intended to be read linearly, from the beginning to the end. Concepts are introduced one at a time, and later sections assume familiarity with earlier sections. Sometimes, later chapters will go into depth on a topic that was only briefly addressed earlier on. Some sections of the book contain exercises. These are worth doing, in order to cement your understanding of the section. It is also useful to explore Lean as you read the book, finding creative new ways to use what you have learned.

Getting Lean

Before writing and running programs written in Lean, you'll need to set up Lean on your own computer. The Lean tooling consists of the following:

- `elan` manages the Lean compiler toolchains, similarly to `rustup` or `ghcup`.
- `lake` builds Lean packages and their dependencies, similarly to `cargo`, `make`, or Gradle.
- `lean` type checks and compiles individual Lean files as well as providing information to programmer tools about files that are currently being written. Normally, `lean` is invoked by other tools rather than directly by users.
- Plugins for editors, such as Visual Studio Code or Emacs, that communicate with `lean` and present its information conveniently.

Please refer to the Lean manual for up-to-date instructions for installing Lean.

Typographical Conventions

Code examples that are provided to Lean as *input* are formatted like this:

```
def add1 (n : Nat) : Nat := n + 1

#eval add1 7
```

The last line above (beginning with `#eval`) is a command that instructs Lean to calculate an answer. Lean's replies are formatted like this:

```
8
```

Error messages returned by Lean are formatted like this:

```
application type mismatch
  add1 "seven"
argument
  "seven"
has type
  String : Type
but is expected to have type
  Nat : Type
```

Warnings are formatted like this:

```
declaration uses 'sorry'
```

Unicode

Idiomatic Lean code makes use of a variety of Unicode characters that are not part of ASCII. For instance, Greek letters like `α` and `β` and the arrow `→` both occur in the first chapter of this book. This allows Lean code to more closely resemble ordinary mathematical notation.

With the default Lean settings, both Visual Studio Code and Emacs allow these characters to be typed with a backslash (`\`) followed by a name. For example, to enter `α`, type `\alpha`. To find out how to type a character in Visual Studio Code, point the mouse at it and look at the tooltip. In Emacs, use `C-c C-k` with point on the character in question.

Acknowledgments

This free online book was made possible by the generous support of Microsoft Research, who paid for it to be written and given away. During the process of writing, they made the expertise of the Lean development team available to both answer my questions and make Lean easier to use. In particular, Leonardo de Moura initiated the project and helped me get started, Chris Lovett set up the CI and deployment automation and provided great feedback as a test reader, Gabriel Ebner provided technical reviews, Sarah Smith kept the administrative side working well, and Vanessa Rodriguez helped me diagnose a tricky interaction between the source-code highlighting library and certain versions of Safari on iOS.

Writing this book has taken up many hours outside of normal working hours. My wife Ellie Thrane Christiansen has taken on a larger than usual share of running the family, and this book could not exist if she had not done so. An extra day of work each week has not been easy for my family—thank you for your patience and support while I was writing.

The online community surrounding Lean provided enthusiastic support for the project, both technical and emotional. In particular, Sebastian Ullrich provided key help when I was learning Lean's metaprogramming system in order to write the supporting code that allowed the text of error messages to be both checked in CI and easily included in the book itself. Within hours of posting a new revision, excited readers would be finding mistakes, providing suggestions, and showering me with kindness. In particular, I'd like to thank Arien Malec, Asta Halkjær From, Bulhwi Cha, Daniel Fabian, Evgenia Karunus, eyelash, Floris van Doorn, František Silváši, Henrik Böving, Ian Young, Jeremy Salwen, Jireh Loreaux, Kevin Buzzard, Mac Malone, Malcolm Langfield, Mario Carneiro, Newell Jensen, Patrick Massot, Paul Chisholm, Tomas Puverle, Yaël Dillies, and Zhiyuan Bao for their many suggestions, both stylistic and technical.

According to tradition, a programming language should be introduced by compiling and running a program that displays `"Hello, world!"` on the console. This simple program ensures that the language tooling is installed correctly and that the programmer is able to run the compiled code.

Since the 1970s, however, programming has changed. Today, compilers are typically integrated into text editors, and the programming environment offers feedback as the program is written. Lean is no exception: it implements an extended version of the Language Server Protocol that allows it to communicate with a text editor and provide feedback as the user types.

Languages as varied as Python, Haskell, and JavaScript offer a read-eval-print-loop (REPL), also known as an interactive toplevel or a browser console, in which expressions or statements can be entered. The language then computes and displays the result of the user's input. Lean, on the other hand, integrates these features into the interaction with the editor, providing commands that cause the text editor to display feedback integrated into the program text itself. This chapter provides a short introduction to interacting with Lean in an editor, while Hello, World! describes how to use Lean traditionally from the command line in batch mode.

It is best if you read this book with Lean open in your editor, following along and typing in each example. Please play with the examples, and see what happens!

Evaluating Expressions

The most important thing to understand as a programmer learning Lean is how evaluation works. Evaluation is the process of finding the value of an expression, just as one does in arithmetic. For instance, the value of 15 - 6 is 9 and the value of 2 × (3 + 1) is 8. To find the value of the latter expression, 3 + 1 is first replaced by 4, yielding 2 × 4, which itself can be reduced to 8. Sometimes, mathematical expressions contain variables: the value of *x* + 1 cannot be computed until we know what the value of *x* is. In Lean, programs are first and foremost expressions, and the primary way to think about computation is as evaluating expressions to find their values.

Most programming languages are *imperative*, where a program consists of a series of statements that should be carried out in order to find the program's result. Programs have access to mutable memory, so the value referred to by a variable can change over time. In addition to mutable state, programs may have other side effects, such as deleting files, making outgoing network connections, throwing or catching exceptions, and reading data from a database. "Side effects" is essentially a catch-all term for describing things that may happen in a program that don't follow the model of evaluating mathematical expressions.

In Lean, however, programs work the same way as mathematical expressions. Once given a value, variables cannot be reassigned. Evaluating an expression cannot have side effects. If two expressions have the same value, then replacing one with the other will not cause the program to compute a different result. This does not mean that Lean cannot be used to write `Hello, world!` to the console, but performing I/O is not a core part of the experience of using Lean in the same way. Thus, this chapter focuses on how to evaluate expressions interactively with Lean, while the next chapter describes how to write, compile, and run the `Hello, world!` program.

To ask Lean to evaluate an expression, write `#eval` before it in your editor, which will then report the result back. Typically, the result is found by putting the cursor or mouse pointer over `#eval`. For instance,

```
#eval 1 + 2
```

yields the value `3`.

Lean obeys the ordinary rules of precedence and associativity for arithmetic operators. That is,

```
#eval 1 + 2 * 5
```

yields the value `11` rather than `15`.

While both ordinary mathematical notation and the majority of programming languages use parentheses (e.g. `f(x)`) to apply a function to its arguments, Lean simply writes the function next to its arguments (e.g. `f x`). Function application is one of the most common operations, so it pays to keep it concise. Rather than writing

```
#eval String.append("Hello, ", "Lean!")
```

to compute `"Hello, Lean!"`, one would instead write

```
#eval String.append "Hello, " "Lean!"
```

where the function's two arguments are simply written next to it with spaces.

Just as the order-of-operations rules for arithmetic demand parentheses in the expression `(1 + 2) * 5`, parentheses are also necessary when a function's argument is to be computed via another function call. For instance, parentheses are required in

```
#eval String.append "great " (String.append "oak " "tree")
```

because otherwise the second `String.append` would be interpreted as an argument to the first, rather than as a function being passed `"oak "` and `"tree"` as arguments. The value of the inner `String.append` call must be found first, after which it can be appended to `"great "`, yielding the final value `"great oak tree"`.

Imperative languages often have two kinds of conditional: a conditional *statement* that determines which instructions to carry out based on a Boolean value, and a conditional *expression* that determines which of two expressions to evaluate based on a Boolean value. For instance, in C and C++, the conditional statement is written using `if` and `else`, while the conditional expression is written with a ternary operator `?` and `:`. In Python, the conditional statement begins with `if`, while the conditional expression puts `if` in the middle. Because Lean is an expression-oriented functional language, there are no conditional statements, only conditional expressions. They are written using `if`, `then`, and `else`. For instance,

```
String.append "it is " (if 1 > 2 then "yes" else "no")
```

evaluates to

```
String.append "it is " (if false then "yes" else "no")
```

which evaluates to

```
String.append "it is " "no"
```

which finally evaluates to `"it is no"`.

For the sake of brevity, a series of evaluation steps like this will sometimes be written with arrows between them:

```
String.append "it is " (if 1 > 2 then "yes" else "no")
===>
String.append "it is " (if false then "yes" else "no")
===>
String.append "it is " "no"
===>
"it is no"
```

Messages You May Meet

Asking Lean to evaluate a function application that is missing an argument will lead to an error message. In particular, the example

```
#eval String.append "it is "
```

yields a quite long error message:

```
expression
  String.append "it is "
has type
  String → String
but instance
  Lean.MetaEval (String → String)
failed to be synthesized, this instance instructs Lean on how to display the
resulting value, recall that any type implementing the `Repr` class also
implements the `Lean.MetaEval` class
```

This message occurs because Lean functions that are applied to only some of their arguments return new functions that are waiting for the rest of the arguments. Lean cannot display functions to users, and thus returns an error when asked to do so.

Exercises

What are the values of the following expressions? Work them out by hand, then enter them into Lean to check your work.

- `42 + 19`
- `String.append "A" (String.append "B" "C")`
- `String.append (String.append "A" "B") "C"`
- `if 3 == 3 then 5 else 7`
- `if 3 == 4 then "equal" else "not equal"`

Types

Types classify programs based on the values that they can compute. Types serve a number of roles in a program:

1. They allow the compiler to make decisions about the in-memory representation of a value.

2. They help programmers to communicate their intent to others, serving as a lightweight specification for the inputs and outputs of a function that the compiler can ensure the program adheres to.

3. They prevent various potential mistakes, such as adding a number to a string, and thus reduce the number of tests that are necessary for a program.

4. They help the Lean compiler automate the production of auxiliary code that can save boilerplate.

Lean's type system is unusually expressive. Types can encode strong specifications like "this sorting function returns a permutation of its input" and flexible specifications like "this function has different return types, depending on the value of its argument". The type system can even be used as a full-blown logic for proving mathematical theorems. This cutting-edge expressive power doesn't obviate the need for simpler types, however, and understanding these simpler types is a prerequisite for using the more advanced features.

Every program in Lean must have a type. In particular, every expression must have a type before it can be evaluated. In the examples so far, Lean has been able to discover a type on its own, but it is sometimes necessary to provide one. This is done using the colon operator:

```
#eval (1 + 2 : Nat)
```

Here, `Nat` is the type of *natural numbers*, which are arbitrary-precision unsigned integers. In Lean, `Nat` is the default type for non-negative integer literals. This default type is not always the best choice. In C, unsigned integers underflow to the largest representable numbers when subtraction would otherwise yield a result less than zero. `Nat`, however, can represent arbitrarily-large unsigned numbers, so there is no largest number to underflow to. Thus, subtraction on `Nat` returns `0` when the answer would have otherwise been negative. For instance,

```
#eval 1 - 2
```

evaluates to `0` rather than `-1`. To use a type that can represent the negative integers, provide it directly:

```
#eval (1 - 2 : Int)
```

With this type, the result is `-1`, as expected.

To check the type of an expression without evaluating it, use `#check` instead of `#eval`. For instance:

```
#check (1 - 2 : Int)
```

reports `1 - 2 : Int` without actually performing the subtraction.

When a program can't be given a type, an error is returned from both `#check` and `#eval`. For instance:

```
#check String.append "hello" [" ", "world"]
```

outputs

```
application type mismatch
  String.append "hello" [" ", "world"]
argument
  [" ", "world"]
has type
  List String : Type
but is expected to have type
  String : Type
```

because the second argument to `String.append` is expected to be a string, but a list of strings was provided instead.

Functions and Definitions

In Lean, definitions are introduced using the `def` keyword. For instance, to define the name `hello` to refer to the string `"Hello"`, write:

```
def hello := "Hello"
```

In Lean, new names are defined using the colon-equal operator `:=` rather than `=`. This is because `=` is used to describe equalities between existing expressions, and using two different operators helps prevent confusion.

In the definition of `hello`, the expression `"Hello"` is simple enough that Lean is able to determine the definition's type automatically. However, most definitions are not so simple, so it will usually be necessary to add a type. This is done using a colon after the name being defined.

```
def lean : String := "Lean"
```

Now that the names have been defined, they can be used, so

```
#eval String.append hello (String.append " " lean)
```

outputs

```
"Hello Lean"
```

In Lean, defined names may only be used after their definitions.

In many languages, definitions of functions use a different syntax than definitions of other values. For instance, Python function definitions begin with the `def` keyword, while other definitions are defined with an equals sign. In Lean, functions are defined using the same `def` keyword as other values. Nonetheless, definitions such as `hello` introduce names that refer *directly* to their values, rather than to zero-argument functions that return equivalent results each time they are called.

Defining Functions

There are a variety of ways to define functions in Lean. The simplest is to place the function's arguments before the definition's type, separated by spaces. For instance, a function that adds one to its argument can be written:

```
def add1 (n : Nat) : Nat := n + 1
```

Testing this function with `#eval` gives `8`, as expected:

```
#eval add1 7
```

Just as functions are applied to multiple arguments by writing spaces between each argument, functions that accept multiple arguments are defined with spaces between the arguments' names and types. The function `maximum`, whose result is equal to the greatest of its two arguments, takes two `Nat` arguments `n` and `k` and returns a `Nat`.

```
def maximum (n : Nat) (k : Nat) : Nat :=
  if n < k then
    k
  else n
```

When a defined function like `maximum` has been provided with its arguments, the result is determined by first replacing the argument names with the provided values in the body, and then evaluating the resulting body. For example:

```
maximum (5 + 8) (2 * 7)
===>
maximum 13 14
===>
if 13 < 14 then 14 else 13
===>
14
```

Expressions that evaluate to natural numbers, integers, and strings have types that say this (`Nat`, `Int`, and `String`, respectively). This is also true of functions. A function that accepts a `Nat` and returns a `Bool` has type `Nat → Bool`, and a function that accepts two `Nat`s and returns a `Nat` has type `Nat → Nat → Nat`.

As a special case, Lean returns a function's signature when its name is used directly with `#check`. Entering `#check add1` yields `add1 (n : Nat) : Nat`. However, Lean can be "tricked" into showing the function's type by writing the function's name in parentheses, which causes the function to be treated as an ordinary expression, so `#check (add1)` yields `add1 : Nat → Nat` and `#check (maximum)` yields `maximum : Nat → Nat → Nat`. This arrow can also be written with an ASCII alternative arrow `->`, so the preceding function types can be written `Nat -> Nat` and `Nat -> Nat -> Nat`, respectively.

Behind the scenes, all functions actually expect precisely one argument. Functions like `maximum` that seem to take more than one argument are in fact functions that take one argument and then return a new function. This new function takes the next argument, and the process continues until no more arguments are expected. This can be seen by providing one argument to a multiple-argument function: `#check maximum 3` yields `maximum 3 : Nat → Nat` and `#check String.append "Hello "` yields `String.append "Hello " : String → String`. Using a function that returns a function to implement multiple-argument functions is called *currying* after the mathematician Haskell Curry. Function arrows associate to the right, which means that `Nat → Nat → Nat` should be parenthesized `Nat → (Nat → Nat)`.

Exercises

- Define the function `joinStringsWith` with type `String -> String -> String -> String` that creates a new string by placing its first argument between its second and third arguments. `joinStringsWith ", " "one" "and another"` should evaluate to `"one, and another"`.
- What is the type of `joinStringsWith ": "`? Check your answer with Lean.
- Define a function `volume` with type `Nat → Nat → Nat → Nat` that computes the volume of a rectangular prism with the given height, width, and depth.

Defining Types

Most typed programming languages have some means of defining aliases for types, such as C's `typedef`. In Lean, however, types are a first-class part of the language - they are expressions like any other. This means that definitions can refer to types just as well as they can refer to other values.

For instance, if `String` is too much to type, a shorter abbreviation `Str` can be defined:

```
def Str : Type := String
```

It is then possible to use `Str` as a definition's type instead of `String`:

```
def aStr : Str := "This is a string."
```

The reason this works is that types follow the same rules as the rest of Lean. Types are expressions, and in an expression, a defined name can be replaced with its definition. Because `Str` has been defined to mean `String`, the definition of `aStr` makes sense.

Messages You May Meet

Experimenting with using definitions for types is made more complicated by the way that Lean supports overloaded integer literals. If `Nat` is too short, a longer name `NaturalNumber` can be defined:

```
def NaturalNumber : Type := Nat
```

However, using `NaturalNumber` as a definition's type instead of `Nat` does not have the expected effect. In particular, the definition:

```
def thirtyEight : NaturalNumber := 38
```

results in the following error:

```
failed to synthesize instance
  OfNat NaturalNumber 38
```

This error occurs because Lean allows number literals to be *overloaded*. When it makes sense to do so, natural number literals can be used for new types, just as if those types were built in to the system. This is part of Lean's mission of making it convenient to represent mathematics, and different branches of mathematics use number notation for very different purposes. The specific feature that allows this overloading does not replace all defined names with their definitions before looking for overloading, which is what leads to the error message above.

One way to work around this limitation is by providing the type `Nat` on the right-hand side of the definition, causing `Nat`'s overloading rules to be used for `38`:

```
def thirtyEight : NaturalNumber := (38 : Nat)
```

The definition is still type-correct because `NaturalNumber` is the same type as `Nat`—by definition!

Another solution is to define an overloading for `NaturalNumber` that works equivalently to the one for `Nat`. This requires more advanced features of Lean, however.

Finally, defining the new name for `Nat` using `abbrev` instead of `def` allows overloading resolution to replace the defined name with its definition. Definitions written using `abbrev` are always unfolded. For instance,

```
abbrev N : Type := Nat
```

and

```
def thirtyNine : N := 39
```

are accepted without issue.

Behind the scenes, some definitions are internally marked as being unfoldable during overload resolution, while others are not. Definitions that are to be unfolded are called *reducible*. Control over reducibility is essential to allow Lean to scale: fully unfolding all definitions can result in very large types that are slow for a machine to process and difficult for users to understand. Definitions produced with `abbrev` are marked as reducible.

Structures

The first step in writing a program is usually to identify the problem domain's concepts, and then find suitable representations for them in code. Sometimes, a domain concept is a collection of other, simpler, concepts. In that case, it can be convenient to group these simpler components together into a single "package", which can then be given a meaningful name. In Lean, this is done using *structures*, which are analogous to `struct`s in C or Rust and `record`s in C#.

Defining a structure introduces a completely new type to Lean that can't be reduced to any other type. This is useful because multiple structures might represent different concepts that nonetheless contain the same data. For instance, a point might be represented using either Cartesian or polar coordinates, each being a pair of floating-point numbers. Defining separate structures prevents API clients from confusing one for another.

Lean's floating-point number type is called `Float`, and floating-point numbers are written in the usual notation.

```
#check 1.2
```

```
1.2 : Float
```

```
#check -454.2123215
```

```
-454.2123215 : Float
```

```
#check 0.0
```

```
0.0 : Float
```

When floating point numbers are written with the decimal point, Lean will infer the type `Float`. If they are written without it, then a type annotation may be necessary.

```
#check 0
```

```
0 : Nat
```

```
#check (0 : Float)
```

```
0 : Float
```

A Cartesian point is a structure with two `Float` fields, called `x` and `y`. This is declared using the `structure` keyword.

```
structure Point where
  x : Float
  y : Float
deriving Repr
```

After this declaration, `Point` is a new structure type. The final line, which says `deriving Repr`, asks Lean to generate code to display values of type `Point`. This code is used by `#eval` to render the result of evaluation for consumption by programmers, analogous to the `repr` function in Python. It is also possible to override the compiler's generated display code.

The typical way to create a value of a structure type is to provide values for all of its fields inside of curly braces. The origin of a Cartesian plane is where `x` and `y` are both zero:

```
def origin : Point := { x := 0.0, y := 0.0 }
```

If the `deriving Repr` line in `Point`'s definition were omitted, then attempting `#eval origin` would yield an error similar to that which occurs when omitting a function's argument:

```
expression
  origin
has type
  Point
but instance
  Lean.MetaEval Point
failed to be synthesized, this instance instructs Lean on how to display the
resulting value, recall that any type implementing the `Repr` class also
implements the `Lean.MetaEval` class
```

That message is saying that the evaluation machinery doesn't know how to communicate the result of evaluation back to the user.

Happily, with `deriving Repr`, the result of `#eval origin` looks very much like the definition of `origin`.

```
{ x := 0.000000, y := 0.000000 }
```

Because structures exist to "bundle up" a collection of data, naming it and treating it as a single unit, it is also important to be able to extract the individual fields of a structure. This is done using dot notation, as in C, Python, or Rust.

```
#eval origin.x
```

```
0.000000
```

```
#eval origin.y
```

```
0.000000
```

This can be used to define functions that take structures as arguments. For instance, addition of points is performed by adding the underlying coordinate values. It should be the case that `#eval addPoints { x := 1.5, y := 32 } { x := -8, y := 0.2 }` yields

```
{ x := -6.500000, y := 32.200000 }
```

The function itself takes two `Points` as arguments, called `p1` and `p2`. The resulting point is based on the `x` and `y` fields of both `p1` and `p2`:

```
def addPoints (p1 : Point) (p2 : Point) : Point :=
  { x := p1.x + p2.x, y := p1.y + p2.y }
```

Similarly, the distance between two points, which is the square root of the sum of the squares of the differences in their `x` and `y` components, can be written:

```
def distance (p1 : Point) (p2 : Point) : Float :=
  Float.sqrt (((p2.x - p1.x) ^ 2.0) + ((p2.y - p1.y) ^ 2.0))
```

For example, the distance between (1, 2) and (5, -1) is 5:

```
#eval distance { x := 1.0, y := 2.0 } { x := 5.0, y := -1.0 }
```

```
5.000000
```

Multiple structures may have fields with the same names. For instance, a three-dimensional point datatype may share the fields `x` and `y`, and be instantiated with the same field names:

```
structure Point3D where
  x : Float
  y : Float
  z : Float
deriving Repr

def origin3D : Point3D := { x := 0.0, y := 0.0, z := 0.0 }
```

This means that the structure's expected type must be known in order to use the curly-brace syntax. If the type is not known, Lean will not be able to instantiate the structure. For instance,

```
#check { x := 0.0, y := 0.0 }
```

leads to the error

```
invalid {...} notation, expected type is not known
```

As usual, the situation can be remedied by providing a type annotation.

```
#check ({ x := 0.0, y := 0.0 } : Point)
```

```
{ x := 0.0, y := 0.0 } : Point
```

To make programs more concise, Lean also allows the structure type annotation inside the curly braces.

```
#check { x := 0.0, y := 0.0 : Point}
```

```
{ x := 0.0, y := 0.0 } : Point
```

Updating Structures

Imagine a function `zeroX` that replaces the `x` field of a `Point` with `0.0`. In most programming language communities, this sentence would mean that the memory location pointed to by `x` was to be overwritten with a new value. However, Lean does not have mutable state. In functional programming communities, what is almost always meant by this kind of statement is that a fresh `Point` is allocated with the `x` field pointing to the new value, and all other fields pointing to the original values from the input. One way to write `zeroX` is to follow this description literally, filling out the new value for `x` and manually transferring `y`:

```
def zeroX (p : Point) : Point :=
  { x := 0, y := p.y }
```

This style of programming has drawbacks, however. First off, if a new field is added to a structure, then every site that updates any field at all must be updated, causing maintenance difficulties. Secondly, if the structure contains multiple fields with the same type, then there is a real risk of copy-paste coding leading to field contents being duplicated or switched. Finally, the program becomes long and bureaucratic.

Lean provides a convenient syntax for replacing some fields in a structure while leaving the others alone. This is done by using the `with` keyword in a structure initialization. The source of unchanged fields occurs before the `with`, and the new fields occur after. For instance, `zeroX` can be written with only the new `x` value:

```
def zeroX (p : Point) : Point :=
  { p with x := 0 }
```

Remember that this structure update syntax does not modify existing values—it creates new values that share some fields with old values. For instance, given the point `fourAndThree`:

```
def fourAndThree : Point :=
  { x := 4.3, y := 3.4 }
```

evaluating it, then evaluating an update of it using `zeroX`, then evaluating it again yields the original value:

```
#eval fourAndThree
```

```
{ x := 4.300000, y := 3.400000 }
```

```
#eval zeroX fourAndThree
```

```
{ x := 0.000000, y := 3.400000 }
```

```
#eval fourAndThree
```

```
{ x := 4.300000, y := 3.400000 }
```

One consequence of the fact that structure updates do not modify the original structure is that it becomes easier to reason about cases where the new value is computed from the old one. All references to the old structure continue to refer to the same field values in all of the new values provided.

Behind the Scenes

Every structure has a *constructor*. Here, the term "constructor" may be a source of confusion. Unlike constructors in languages such as Java or Python, constructors in Lean are not arbitrary code to be run when a datatype is initialized. Instead, constructors simply gather the data to be stored in the newly-allocated data structure. It is not possible to provide a custom constructor that pre-processes data or rejects invalid arguments. This is really a case of the word "constructor" having different, but related, meanings in the two contexts.

By default, the constructor for a structure named `S` is named `S.mk`. Here, `S` is a namespace qualifier, and `mk` is the name of the constructor itself. Instead of using curly-brace initialization syntax, the constructor can also be applied directly.

```
#check Point.mk 1.5 2.8
```

However, this is not generally considered to be good Lean style, and Lean even returns its feedback using the standard structure initializer syntax.

```
{ x := 1.5, y := 2.8 } : Point
```

Constructors have function types, which means they can be used anywhere that a function is expected. For instance, `Point.mk` is a function that accepts two `Float`s (respectively `x` and `y`) and returns a new `Point`.

```
#check (Point.mk)
```

```
Point.mk : Float → Float → Point
```

To override a structure's constructor name, write it with two colons at the beginning. For instance, to use `Point.point` instead of `Point.mk`, write:

```
structure Point where
  point ::
  x : Float
  y : Float
deriving Repr
```

In addition to the constructor, an accessor function is defined for each field of a structure. These have the same name as the field, in the structure's namespace. For `Point`, accessor functions `Point.x` and `Point.y` are generated.

```
#check (Point.x)
```

```
Point.x : Point → Float
```

```
#check (Point.y)
```

```
Point.y : Point → Float
```

In fact, just as the curly-braced structure construction syntax is converted to a call to the structure's constructor behind the scenes, the syntax `p1.x` in the prior definition of `addPoints` is converted into a call to the `Point.x` accessor. That is, `#eval origin.x` and `#eval Point.x origin` both yield

```
0.000000
```

Accessor dot notation is usable with more than just structure fields. It can also be used for functions that take any number of arguments. More generally, accessor notation has the form `TARGET.f ARG1 ARG2 ...`. If `TARGET` has type `T`, the function named `T.f` is called. `TARGET` becomes its leftmost argument of type `T`, which is often but not always the first one, and `ARG1 ARG2 ...` are provided in order as the remaining arguments. For instance, `String.append` can be invoked from a string with accessor notation, even though `String` is not a structure with an `append` field.

```
#eval "one string".append " and another"
```

```
"one string and another"
```

In that example, `TARGET` represents `"one string"` and `ARG1` represents `" and another"`.

The function `Point.modifyBoth` (that is, `modifyBoth` defined in the `Point` namespace) applies a function to both fields in a `Point`:

```
def Point.modifyBoth (f : Float → Float) (p : Point) : Point :=
  { x:= f p.x, y := f p.y }
```

Even though the `Point` argument comes after the function argument, it can be used with dot notation as well:

```
#eval fourAndThree.modifyBoth Float.floor
```

```
{ x := 4.000000, y := 3.000000 }
```

In this case, `TARGET` represents `fourAndThree`, while `ARG1` is `Float.floor`. This is because the target of the accessor notation is used as the first argument in which the type matches, not necessarily the first argument.

Exercises

- Define a structure named `RectangularPrism` that contains the height, width, and depth of a rectangular prism, each as a `Float`.
- Define a function named `volume : RectangularPrism → Float` that computes the volume of a rectangular prism.
- Define a structure named `Segment` that represents a line segment by its endpoints, and define a function `length : Segment → Float` that computes the length of a line segment. `Segment` should have at most two fields.
- Which names are introduced by the declaration of `RectangularPrism`?
- Which names are introduced by the following declarations of `Hamster` and `Book`? What are their types?

```
structure Hamster where
  name : String
  fluffy : Bool
```

```
structure Book where
  makeBook ::
  title : String
  author : String
  price : Float
```

Datatypes and Patterns

Structures enable multiple independent pieces of data to be combined into a coherent whole that is represented by a brand new type. Types such as structures that group together a collection of values are called *product types*. Many domain concepts, however, can't be naturally represented as structures. For instance, an application might need to track user permissions, where some users are document owners, some may edit documents, and others may only read them. A calculator has a number of binary operators, such as addition, subtraction, and multiplication. Structures do not provide an easy way to encode multiple choices.

Similarly, while a structure is an excellent way to keep track of a fixed set of fields, many applications require data that may contain an arbitrary number of elements. Most classic data structures, such as trees and lists, have a recursive structure, where the tail of a list is itself a list, or where the left and right branches of a binary tree are themselves binary trees. In the aforementioned calculator, the structure of expressions themselves is recursive. The summands in an addition expression may themselves be multiplication expressions, for instance.

Datatypes that allow choices are called *sum types* and datatypes that can include instances of themselves are called *recursive datatypes*. Recursive sum types are called *inductive datatypes*, because mathematical induction may be used to prove statements about them. When programming, inductive datatypes are consumed through pattern matching and recursive functions.

Many of the built-in types are actually inductive datatypes in the standard library. For instance, `Bool` is an inductive datatype:

```
inductive Bool where
  | false : Bool
  | true : Bool
```

This definition has two main parts. The first line provides the name of the new type (`Bool`), while the remaining lines each describe a constructor. As with constructors of structures, constructors of inductive datatypes are mere inert receivers of and containers for other data, rather than places to insert arbitrary initialization and validation code. Unlike structures, inductive datatypes may have multiple constructors. Here, there are two constructors, `true` and `false` , and neither takes any arguments. Just as a structure declaration places its names in a namespace named after the declared type, an inductive datatype places the names of its constructors in a namespace. In the Lean standard library, `true` and `false` are re-exported from this namespace so that they can be written alone, rather than as `Bool.true` and `Bool.false` , respectively.

From a data modeling perspective, inductive datatypes are used in many of the same contexts where a sealed abstract class might be used in other languages. In languages like

C# or Java, one might write a similar definition of `Bool`:

```
abstract class Bool {}
class True : Bool {}
class False : Bool {}
```

However, the specifics of these representations are fairly different. In particular, each non-abstract class creates both a new type and new ways of allocating data. In the object-oriented example, `True` and `False` are both types that are more specific than `Bool`, while the Lean definition introduces only the new type `Bool`.

The type `Nat` of non-negative integers is an inductive datatype:

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Here, `zero` represents 0, while `succ` represents the successor of some other number. The `Nat` mentioned in `succ`'s declaration is the very type `Nat` that is in the process of being defined. *Successor* means "one greater than", so the successor of five is six and the successor of 32,185 is 32,186. Using this definition, `4` is represented as `Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))`. This definition is almost like the definition of `Bool` with slightly different names. The only real difference is that `succ` is followed by `(n : Nat)`, which specifies that the constructor `succ` takes an argument of type `Nat` which happens to be named `n`. The names `zero` and `succ` are in a namespace named after their type, so they must be referred to as `Nat.zero` and `Nat.succ`, respectively.

Argument names, such as `n`, may occur in Lean's error messages and in feedback provided when writing mathematical proofs. Lean also has an optional syntax for providing arguments by name. Generally, however, the choice of argument name is less important than the choice of a structure field name, as it does not form as large a part of the API.

In C# or Java, `Nat` could be defined as follows:

```
abstract class Nat {}
class Zero : Nat {}
class Succ : Nat {
  public Nat n;
  public Succ(Nat pred) {
    n = pred;
  }
}
```

Just as in the `Bool` example above, this defines more types than the Lean equivalent. Additionally, this example highlights how Lean datatype constructors are much more like subclasses of an abstract class than they are like constructors in C# or Java, as the constructor shown here contains initialization code to be executed.

Sum types are also similar to using a string tag to encode discriminated unions in TypeScript. In TypeScript, `Nat` could be defined as follows:

```
interface Zero {
    tag: "zero";
}

interface Succ {
    tag: "succ";
    predecessor: Nat;
}

type Nat = Zero | Succ;
```

Just like C# and Java, this encoding ends up with more types than in Lean, because `Zero` and `Succ` are each a type on their own. It also illustrates that Lean constructors correspond to objects in JavaScript or TypeScript that include a tag that identifies the contents.

Pattern Matching

In many languages, these kinds of data are consumed by first using an instance-of operator to check which subclass has been received and then reading the values of the fields that are available in the given subclass. The instance-of check determines which code to run, ensuring that the data needed by this code is available, while the fields themselves provide the data. In Lean, both of these purposes are simultaneously served by *pattern matching*.

An example of a function that uses pattern matching is `isZero`, which is a function that returns `true` when its argument is `Nat.zero`, or false otherwise.

```
def isZero (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => false
```

The `match` expression is provided the function's argument `n` for destructuring. If `n` was constructed by `Nat.zero`, then the first branch of the pattern match is taken, and the result is `true`. If `n` was constructed by `Nat.succ`, then the second branch is taken, and the result is `false`.

Step-by-step, evaluation of `isZero Nat.zero` proceeds as follows:

```
isZero Nat.zero
===>
match Nat.zero with
| Nat.zero => true
| Nat.succ k => false
===>
true
```

Evaluation of `isZero 5` proceeds similarly:

```
isZero 5
===>
isZero (Nat.succ (Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))))
===>
match Nat.succ (Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))) with
| Nat.zero => true
| Nat.succ k => false
===>
false
```

The `k` in the second branch of the pattern in `isZero` is not decorative. It makes the `Nat` that is the argument to `succ` visible, with the provided name. That smaller number can then be used to compute the final result of the expression.

Just as the successor of some number n is one greater than n (that is, $n + 1$), the predecessor of a number is one less than it. If `pred` is a function that finds the predecessor of a `Nat`, then it should be the case that the following examples find the expected result:

```
#eval pred 5
```

```
4
```

```
#eval pred 839
```

```
838
```

Because `Nat` cannot represent negative numbers, `0` is a bit of a conundrum. Usually, when working with `Nat`, operators that would ordinarily produce a negative number are redefined to produce `0` itself:

```
#eval pred 0
```

```
0
```

To find the predecessor of a `Nat`, the first step is to check which constructor was used to create it. If it was `Nat.zero`, then the result is `Nat.zero`. If it was `Nat.succ`, then the name `k` is used to refer to the `Nat` underneath it. And this `Nat` is the desired predecessor, so the result of the `Nat.succ` branch is `k`.

```
def pred (n : Nat) : Nat :=
  match n with
  | Nat.zero => Nat.zero
  | Nat.succ k => k
```

Applying this function to `5` yields the following steps:

```
pred 5
===>
pred (Nat.succ 4)
===>
match Nat.succ 4 with
| Nat.zero => Nat.zero
| Nat.succ k => k
===>
4
```

Pattern matching can be used with structures as well as with sum types. For instance, a function that extracts the third dimension from a `Point3D` can be written as follows:

```
def depth (p : Point3D) : Float :=
  match p with
  | { x:= h, y := w, z := d } => d
```

In this case, it would have been much simpler to just use the `z` accessor, but structure patterns are occasionally the simplest way to write a function.

Recursive Functions

Definitions that refer to the name being defined are called *recursive definitions*. Inductive datatypes are allowed to be recursive; indeed, `Nat` is an example of such a datatype because `succ` demands another `Nat`. Recursive datatypes can represent arbitrarily large data, limited only by technical factors like available memory. Just as it would be impossible to write down one constructor for each natural number in the datatype definition, it is also impossible to write down a pattern match case for each possibility.

Recursive datatypes are nicely complemented by recursive functions. A simple recursive function over `Nat` checks whether its argument is even. In this case, `zero` is even. Non-recursive branches of the code like this one are called *base cases*. The successor of an odd number is even, and the successor of an even number is odd. This means that a number built with `succ` is even if and only if its argument is not even.

```
def even (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (even k)
```

This pattern of thought is typical for writing recursive functions on `Nat`. First, identify what to do for `zero`. Then, determine how to transform a result for an arbitrary `Nat` into a result for its successor, and apply this transformation to the result of the recursive call. This pattern is called *structural recursion*.

Unlike many languages, Lean ensures by default that every recursive function will eventually reach a base case. From a programming perspective, this rules out accidental infinite loops. But this feature is especially important when proving theorems, where infinite loops cause major difficulties. A consequence of this is that Lean will not accept a version of `even` that attempts to invoke itself recursively on the original number:

```
def evenLoops (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (evenLoops n)
```

The important part of the error message is that Lean could not determine that the recursive function always reaches a base case (because it doesn't).

```
fail to show termination for
  evenLoops
with errors
structural recursion cannot be used

well-founded recursion cannot be used, 'evenLoops' does not take any (non-fixed)
arguments
```

Even though addition takes two arguments, only one of them needs to be inspected. To add zero to a number n, just return n. To add the successor of k to n, take the successor of the result of adding k to n.

```
def plus (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => n
  | Nat.succ k' => Nat.succ (plus n k')
```

In the definition of `plus`, the name `k'` is chosen to indicate that it is connected to, but not identical with, the argument `k`. For instance, walking through the evaluation of `plus 3 2` yields the following steps:

```
plus 3 2
===>
plus 3 (Nat.succ (Nat.succ Nat.zero))
===>
match Nat.succ (Nat.succ Nat.zero) with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k')
===>
Nat.succ (plus 3 (Nat.succ Nat.zero))
===>
Nat.succ (match Nat.succ Nat.zero with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k'))
===>
Nat.succ (Nat.succ (plus 3 Nat.zero))
===>
Nat.succ (Nat.succ (match Nat.zero with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k')))
===>
Nat.succ (Nat.succ 3)
===>
5
```

One way to think about addition is that $n + k$ applies `Nat.succ` k times to n. Similarly, multiplication $n \times k$ adds n to itself k times and subtraction $n - k$ takes n's predecessor k times.

```
def times (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => Nat.zero
  | Nat.succ k' => plus n (times n k')

def minus (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => n
  | Nat.succ k' => pred (minus n k')
```

Not every function can be easily written using structural recursion. The understanding of addition as iterated `Nat.succ`, multiplication as iterated addition, and subtraction as iterated predecessor suggests an implementation of division as iterated subtraction. In this case, if the numerator is less than the divisor, the result is zero. Otherwise, the result is the successor of dividing the numerator minus the divisor by the divisor.

```
def div (n : Nat) (k : Nat) : Nat :=
  if n < k then
    0
  else Nat.succ (div (n - k) k)
```

As long as the second argument is not `0`, this program terminates, as it always makes progress towards the base case. However, it is not structurally recursive, because it doesn't follow the pattern of finding a result for zero and transforming a result for a smaller `Nat` into a result for its successor. In particular, the recursive invocation of the function is applied to the result of another function call, rather than to an input constructor's argument. Thus, Lean rejects it with the following message:

```
fail to show termination for
  div
with errors
argument #1 was not used for structural recursion
  failed to eliminate recursive application
    div (n - k) k

argument #2 was not used for structural recursion
  failed to eliminate recursive application
    div (n - k) k

structural recursion cannot be used

failed to prove termination, use `termination_by` to specify a well-founded
relation
```

This message means that `div` requires a manual proof of termination. This topic is explored in the final chapter.

Polymorphism

Just as in most languages, types in Lean can take arguments. For instance, the type `List Nat` describes lists of natural numbers, `List String` describes lists of strings, and `List (List Point)` describes lists of lists of points. This is very similar to `List<Nat>`, `List<String>`, or `List<List<Point>>` in a language like C# or Java. Just as Lean uses a space to pass an argument to a function, it uses a space to pass an argument to a type.

In functional programming, the term *polymorphism* typically refers to datatypes and definitions that take types as arguments. This is different from the object-oriented programming community, where the term typically refers to subclasses that may override some behavior of their superclass. In this book, "polymorphism" always refers to the first sense of the word. These type arguments can be used in the datatype or definition, which allows the same datatype or definition to be used with any type that results from replacing the arguments' names with some other types.

The `Point` structure requires that both the `x` and `y` fields are `Float`s. There is, however, nothing about points that require a specific representation for each coordinate. A polymorphic version of `Point`, called `PPoint`, can take a type as an argument, and then use that type for both fields:

```
structure PPoint (α : Type) where
  x : α
  y : α
deriving Repr
```

Just as a function definition's arguments are written immediately after the name being defined, a structure's arguments are written immediately after the structure's name. It is customary to use Greek letters to name type arguments in Lean when no more specific name suggests itself. `Type` is a type that describes other types, so `Nat`, `List String`, and `PPoint Int` all have type `Type`.

Just like `List`, `PPoint` can be used by providing a specific type as its argument:

```
def natOrigin : PPoint Nat :=
  { x := Nat.zero, y := Nat.zero }
```

In this example, both fields are expected to be `Nat`s. Just as a function is called by replacing its argument variables with its argument values, providing `PPoint` with the type `Nat` as an argument yields a structure in which the fields `x` and `y` have the type `Nat`, because the argument name `α` has been replaced by the argument type `Nat`. Types are ordinary expressions in Lean, so passing arguments to polymorphic types (like `PPoint`) doesn't require any special syntax.

Definitions may also take types as arguments, which makes them polymorphic. The function `replaceX` replaces the `x` field of a `PPoint` with a new value. In order to allow `replaceX` to work with *any* polymorphic point, it must be polymorphic itself. This is achieved by having its first argument be the type of the point's fields, with later arguments referring back to the first argument's name.

```
def replaceX (α : Type) (point : PPoint α) (newX : α) : PPoint α :=
  { point with x := newX }
```

In other words, when the types of the arguments `point` and `newX` mention `α`, they are referring to *whichever type was provided as the first argument*. This is similar to the way that function argument names refer to the values that were provided when they occur in the function's body.

This can be seen by asking Lean to check the type of `replaceX`, and then asking it to check the type of `replaceX Nat`.

```
#check (replaceX)
```

```
replaceX : (α : Type) → PPoint α → α → PPoint α
```

This function type includes the *name* of the first argument, and later arguments in the type refer back to this name. Just as the value of a function application is found by replacing the argument name with the provided argument value in the function's body, the type of a function application is found by replacing the argument's name with the provided value in the function's return type. Providing the first argument, `Nat`, causes all occurrences of `α` in the remainder of the type to be replaced with `Nat`:

```
#check replaceX Nat
```

```
replaceX Nat : PPoint Nat → Nat → PPoint Nat
```

Because the remaining arguments are not explicitly named, no further substitution occurs as more arguments are provided:

```
#check replaceX Nat natOrigin
```

```
replaceX Nat natOrigin : Nat → PPoint Nat
```

```
#check replaceX Nat natOrigin 5
```

```
replaceX Nat natOrigin 5 : PPoint Nat
```

The fact that the type of the whole function application expression was determined by passing a type as an argument has no bearing on the ability to evaluate it.

```
#eval replaceX Nat natOrigin 5
```

```
{ x := 5, y := 0 }
```

Polymorphic functions work by taking a named type argument and having later types refer to the argument's name. However, there's nothing special about type arguments that allows them to be named. Given a datatype that represents positive or negative signs:

```
inductive Sign where
  | pos
  | neg
```

it is possible to write a function whose argument is a sign. If the argument is positive, the function returns a `Nat`, while if it's negative, it returns an `Int`:

```
def posOrNegThree (s : Sign) : match s with | Sign.pos => Nat | Sign.neg => Int
:=
  match s with
  | Sign.pos => (3 : Nat)
  | Sign.neg => (-3 : Int)
```

Because types are first class and can be computed using the ordinary rules of the Lean language, they can be computed by pattern-matching against a datatype. When Lean is checking this function, it uses the fact that the `match`-expression in the function's body corresponds to the `match`-expression in the type to make `Nat` be the expected type for the `pos` case and to make `Int` be the expected type for the `neg` case.

Applying `posOrNegThree` to `Sign.pos` results in the argument name `s` in both the body of the function and its return type being replaced by `Sign.pos`. Evaluation can occur both in the expression and its type:

```
(posOrNegThree Sign.pos : match Sign.pos with | Sign.pos => Nat | Sign.neg =>
Int)
===>
((match Sign.pos with
  | Sign.pos => (3 : Nat)
  | Sign.neg => (-3 : Int)) :
 match Sign.pos with | Sign.pos => Nat | Sign.neg => Int)
===>
((3 : Nat) : Nat)
===>
3
```

Linked Lists

Lean's standard library includes a canonical linked list datatype, called `List`, and special syntax that makes it more convenient to use. Lists are written in square brackets. For instance, a list that contains the prime numbers less than 10 can be written:

```
def primesUnder10 : List Nat := [2, 3, 5, 7]
```

Behind the scenes, `List` is an inductive datatype, defined like this:

```
inductive List (α : Type) where
  | nil : List α
  | cons : α → List α → List α
```

The actual definition in the standard library is slightly different, because it uses features that have not yet been presented, but it is substantially similar. This definition says that `List` takes a single type as its argument, just as `PPoint` did. This type is the type of the entries stored in the list. According to the constructors, a `List α` can be built with either `nil` or `cons`. The constructor `nil` represents empty lists and the constructor `cons` is used for non-empty lists. The first argument to `cons` is the head of the list, and the second argument is its tail. A list that contains n entries contains n `cons` constructors, the last of which has `nil` as its tail.

The `primesUnder10` example can be written more explicitly by using `List`'s constructors directly:

```
def explicitPrimesUnder10 : List Nat :=
  List.cons 2 (List.cons 3 (List.cons 5 (List.cons 7 List.nil)))
```

These two definitions are completely equivalent, but `primesUnder10` is much easier to read than `explicitPrimesUnder10`.

Functions that consume `List`s can be defined in much the same way as functions that consume `Nat`s. Indeed, one way to think of a linked list is as a `Nat` that has an extra data field dangling off each `succ` constructor. From this point of view, computing the length of a list is the process of replacing each `cons` with a `succ` and the final `nil` with a `zero`. Just as `replaceX` took the type of the fields of the point as an argument, `length` takes the type of the list's entries. For example, if the list contains strings, then the first argument is `String`: `length String ["Sourdough", "bread"]`. It should compute like this:

```
length String ["Sourdough", "bread"]
===>
length String (List.cons "Sourdough" (List.cons "bread" List.nil))
===>
Nat.succ (length String (List.cons "bread" List.nil))
===>
Nat.succ (Nat.succ (length String List.nil))
===>
Nat.succ (Nat.succ Nat.zero)
===>
2
```

The definition of `length` is both polymorphic (because it takes the list entry type as an argument) and recursive (because it refers to itself). Generally, functions follow the shape of the data: recursive datatypes lead to recursive functions, and polymorphic datatypes lead to polymorphic functions.

```
def length (α : Type) (xs : List α) : Nat :=
  match xs with
  | List.nil => Nat.zero
  | List.cons y ys => Nat.succ (length α ys)
```

Names such as `xs` and `ys` are conventionally used to stand for lists of unknown values. The `s` in the name indicates that they are plural, so they are pronounced "exes" and "whys" rather than "x s" and "y s".

To make it easier to read functions on lists, the bracket notation `[]` can be used to pattern-match against `nil`, and an infix `::` can be used in place of `cons`:

```
def length (α : Type) (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length α ys)
```

Implicit Arguments

Both `replaceX` and `length` are somewhat bureaucratic to use, because the type argument is typically uniquely determined by the later values. Indeed, in most languages, the compiler is perfectly capable of determining type arguments on its own, and only occasionally needs help from users. This is also the case in Lean. Arguments can be declared *implicit* by wrapping them in curly braces instead of parentheses when defining a function. For instance, a version of `replaceX` with an implicit type argument looks like this:

```
def replaceX {α : Type} (point : PPoint α) (newX : α) : PPoint α :=
  { point with x := newX }
```

It can be used with `natOrigin` without providing `Nat` explicitly, because Lean can *infer* the value of `α` from the later arguments:

```
#eval replaceX natOrigin 5
```

```
{ x := 5, y := 0 }
```

Similarly, `length` can be redefined to take the entry type implicitly:

```
def length {α : Type} (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This `length` function can be applied directly to `primesUnder10`:

```
#eval length primesUnder10
```

```
4
```

In the standard library, Lean calls this function `List.length`, which means that the dot syntax that is used for structure field access can also be used to find the length of a list:

```
#eval primesUnder10.length
```

```
4
```

Just as C# and Java require type arguments to be provided explicitly from time to time, Lean is not always capable of finding implicit arguments. In these cases, they can be provided using their names. For instance, a version of `List.length` that only works for lists of integers can be specified by setting `α` to `Int`:

```
#check List.length (α := Int)
```

```
List.length : List Int → Nat
```

More Built-In Datatypes

In addition to lists, Lean's standard library contains a number of other structures and inductive datatypes that can be used in a variety of contexts.

`Option`

Not every list has a first entry—some lists are empty. Many operations on collections may fail to find what they are looking for. For instance, a function that finds the first entry in a list may not find any such entry. It must therefore have a way to signal that there was no first entry.

Many languages have a `null` value that represents the absence of a value. Instead of equipping existing types with a special `null` value, Lean provides a datatype called `Option` that equips some other type with an indicator for missing values. For instance, a nullable `Int` is represented by `Option Int`, and a nullable list of strings is represented by the type

`Option (List String)`. Introducing a new type to represent nullability means that the type system ensures that checks for `null` cannot be forgotten, because an `Option Int` can't be used in a context where an `Int` is expected.

`Option` has two constructors, called `some` and `none`, that respectively represent the non-null and null versions of the underlying type. The non-null constructor, `some`, contains the underlying value, while `none` takes no arguments:

```
inductive Option (α : Type) : Type where
  | none : Option α
  | some (val : α) : Option α
```

The `Option` type is very similar to nullable types in languages like C# and Kotlin, but it is not identical. In these languages, if a type (say, `Boolean`) always refers to actual values of the type (`true` and `false`), the type `Boolean?` or `Nullable<Boolean>` additionally admits the `null` value. Tracking this in the type system is very useful: the type checker and other tooling can help programmers remember to check for null, and APIs that explicitly describe nullability through type signatures are more informative than ones that don't. However, these nullable types differ from Lean's `Option` in one very important way, which is that they don't allow multiple layers of optionality. `Option (Option Int)` can be constructed with `none`, `some none`, or `some (some 360)`. C#, on the other hand, forbids multiple layers of nullability by only allowing `?` to be added to non-nullable types, while Kotlin treats `T??` as being equivalent to `T?`. This subtle difference is rarely relevant in practice, but it can matter from time to time.

To find the first entry in a list, if it exists, use `List.head?`. The question mark is part of the name, and is not related to the use of question marks to indicate nullable types in C# or Kotlin. In the definition of `List.head?`, an underscore is used to represent the tail of the list. In patterns, underscores match anything at all, but do not introduce variables to refer to the matched data. Using underscores instead of names is a way to clearly communicate to readers that part of the input is ignored.

```
def List.head? {α : Type} (xs : List α) : Option α :=
  match xs with
  | [] => none
  | y :: _ => some y
```

A Lean naming convention is to define operations that might fail in groups using the suffixes `?` for a version that returns an `Option`, `!` for a version that crashes when provided with invalid input, and `D` for a version that returns a default value when the operation would otherwise fail. For instance, `head` requires the caller to provide mathematical evidence that the list is not empty, `head?` returns an `Option`, `head!` crashes the program when passed an empty list, and `headD` takes a default value to return in case the list is empty. The question mark and exclamation mark are part of the name, not special syntax, as Lean's naming rules are more liberal than many languages.

Because `head?` is defined in the `List` namespace, it can be used with accessor notation:

```
#eval primesUnder10.head?
```

```
some 2
```

However, attempting to test it on the empty list leads to two errors:

```
#eval [].head?
```

```
don't know how to synthesize implicit argument
  @List.nil ?m.20368
context:
⊢ Type ?u.20365

don't know how to synthesize implicit argument
  @_root_.List.head? ?m.20368 []
context:
⊢ Type ?u.20365
```

This is because Lean was unable to fully determine the expression's type. In particular, it could neither find the implicit type argument to `List.head?`, nor could it find the implicit type argument to `List.nil`. In Lean's output, `?m.XYZ` represents a part of a program that could not be inferred. These unknown parts are called *metavariables*, and they occur in some error messages. In order to evaluate an expression, Lean needs to be able to find its type, and the type was unavailable because the empty list does not have any entries from which the type can be found. Explicitly providing a type allows Lean to proceed:

```
#eval [].head? (α := Int)
```

```
none
```

The type can also be provided with a type annotation:

```
#eval ([] : List Int).head?
```

```
none
```

The error messages provide a useful clue. Both messages use the *same* metavariable to describe the missing implicit argument, which means that Lean has determined that the two missing pieces will share a solution, even though it was unable to determine the actual value of the solution.

`Prod`

The `Prod` structure, short for "Product", is a generic way of joining two values together. For instance, a `Prod Nat String` contains a `Nat` and a `String`. In other words, `PPoint Nat` could be replaced by `Prod Nat Nat`. `Prod` is very much like C#'s tuples, the `Pair` and `Triple` types in Kotlin, and `tuple` in C++. Many applications are best served by defining their own structures, even for simple cases like `Point`, because using domain terminology can make it easier to read the code. Additionally, defining structure types helps catch more errors by assigning different types to different domain concepts, preventing them from being mixed up.

On the other hand, there are some cases where it is not worth the overhead of defining a new type. Additionally, some libraries are sufficiently generic that there is no more specific concept than "pair". Finally, the standard library contains a variety of convenience functions that make it easier to work with the built-in pair type.

The standard pair structure is called `Prod`.

```
structure Prod (α : Type) (β : Type) : Type where
  fst : α
  snd : β
```

Lists are used so frequently that there is special syntax to make them more readable. For the same reason, both the product type and its constructor have special syntax. The type `Prod α β` is typically written `α × β`, mirroring the usual notation for a Cartesian product of sets. Similarly, the usual mathematical notation for pairs is available for `Prod`. In other words, instead of writing:

```
def fives : String × Int := { fst := "five", snd := 5 }
```

it suffices to write:

```
def fives : String × Int := ("five", 5)
```

Both notations are right-associative. This means that the following definitions are equivalent:

```
def sevens : String × Int × Nat := ("VII", 7, 4 + 3)

def sevens : String × (Int × Nat) := ("VII", (7, 4 + 3))
```

In other words, all products of more than two types, and their corresponding constructors, are actually nested products and nested pairs behind the scenes.

`Sum`

The `Sum` datatype is a generic way of allowing a choice between values of two different types. For instance, a `Sum String Int` is either a `String` or an `Int`. Like `Prod`, `Sum` should be used either when writing very generic code, for a very small section of code where there is no sensible domain-specific type, or when the standard library contains useful functions. In most situations, it is more readable and maintainable to use a custom inductive type.

Values of type `Sum α β` are either the constructor `inl` applied to a value of type `α` or the constructor `inr` applied to a value of type `β`:

```
inductive Sum (α : Type) (β : Type) : Type where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

These names are abbreviations for "left injection" and "right injection", respectively. Just as the Cartesian product notation is used for `Prod`, a "circled plus" notation is used for `Sum`, so `α ⊕ β` is another way to write `Sum α β`. There is no special syntax for `Sum.inl` and `Sum.inr`.

For instance, if pet names can either be dog names or cat names, then a type for them can be introduced as a sum of strings:

```
def PetName : Type := String ⊕ String
```

In a real program, it would usually be better to define a custom inductive datatype for this purpose with informative constructor names. Here, `Sum.inl` is to be used for dog names, and `Sum.inr` is to be used for cat names. These constructors can be used to write a list of animal names:

```
def animals : List PetName :=
  [Sum.inl "Spot", Sum.inr "Tiger", Sum.inl "Fifi", Sum.inl "Rex", Sum.inr
"Floof"]
```

Pattern matching can be used to distinguish between the two constructors. For instance, a function that counts the number of dogs in a list of animal names (that is, the number of `Sum.inl` constructors) looks like this:

```
def howManyDogs (pets : List PetName) : Nat :=
  match pets with
  | [] => 0
  | Sum.inl _ :: morePets => howManyDogs morePets + 1
  | Sum.inr _ :: morePets => howManyDogs morePets
```

Function calls are evaluated before infix operators, so `howManyDogs morePets + 1` is the same as `(howManyDogs morePets) + 1`. As expected, `#eval howManyDogs animals` yields `3`.

`Unit`

`Unit` is a type with just one argumentless constructor, called `unit`. In other words, it describes only a single value, which consists of said constructor applied to no arguments whatsoever. `Unit` is defined as follows:

```
inductive Unit : Type where
  | unit : Unit
```

On its own, `Unit` is not particularly useful. However, in polymorphic code, it can be used as a placeholder for data that is missing. For instance, the following inductive datatype represents arithmetic expressions:

```
inductive ArithExpr (ann : Type) : Type where
  | int : ann → Int → ArithExpr ann
  | plus : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
  | minus : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
  | times : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
```

The type argument `ann` stands for annotations, and each constructor is annotated. Expressions coming from a parser might be annotated with source locations, so a return type of `ArithExpr SourcePos` ensures that the parser put a `SourcePos` at each subexpression. Expressions that don't come from the parser, however, will not have source locations, so their type can be `ArithExpr Unit`.

Additionally, because all Lean functions have arguments, zero-argument functions in other languages can be represented as functions that take a `Unit` argument. In a return position, the `Unit` type is similar to `void` in languages derived from C. In the C family, a function that returns `void` will return control to its caller, but it will not return any interesting value. By being an intentionally uninteresting value, `Unit` allows this to be expressed without requiring a special-purpose `void` feature in the type system. Unit's constructor can be written as empty parentheses: `() : Unit`.

`Empty`

The `Empty` datatype has no constructors whatsoever. Thus, it indicates unreachable code, because no series of calls can ever terminate with a value at type `Empty`.

`Empty` is not used nearly as often as `Unit`. However, it is useful in some specialized contexts. Many polymorphic datatypes do not use all of their type arguments in all of their constructors. For instance, `Sum.inl` and `Sum.inr` each use only one of `Sum`'s type arguments. Using `Empty` as one of the type arguments to `Sum` can rule out one of the constructors at a particular point in a program. This can allow generic code to be used in contexts that have additional restrictions.

Naming: Sums, Products, and Units

Generally speaking, types that offer multiple constructors are called *sum types*, while types whose single constructor takes multiple arguments are called *product types*. These terms are related to sums and products used in ordinary arithmetic. The relationship is easiest to see when the types involved contain a finite number of values. If `α` and `β` are types that contain n and k distinct values, respectively, then `α ⊕ β` contains $n + k$ distinct values and `α × β` contains $n \times k$ distinct values. For instance, `Bool` has two values: `true` and `false`, and `Unit` has one value: `Unit.unit`. The product `Bool × Unit` has the two values `(true, Unit.unit)` and `(false, Unit.unit)`, and the sum `Bool ⊕ Unit` has the three values `Sum.inl true`, `Sum.inl false`, and `Sum.inr unit`. Similarly, $2 \times 1 = 2$, and $2 + 1 = 3$.

Messages You May Meet

Not all definable structures or inductive types can have the type `Type`. In particular, if a constructor takes an arbitrary type as an argument, then the inductive type must have a different type. These errors usually state something about "universe levels". For example, for this inductive type:

```
inductive MyType : Type where
  | ctor : (α : Type) → α → MyType
```

Lean gives the following error:

```
invalid universe level in constructor 'MyType.ctor', parameter 'α' has type
  Type
at universe level
  2
it must be smaller than or equal to the inductive datatype universe level
  1
```

A later chapter describes why this is the case, and how to modify definitions to make them work. For now, try making the type an argument to the inductive type as a whole, rather than to the constructor.

Similarly, if a constructor's argument is a function that takes the datatype being defined as an argument, then the definition is rejected. For example:

```
inductive MyType : Type where
  | ctor : (MyType → Int) → MyType
```

yields the message:

```
(kernel) arg #1 of 'MyType.ctor' has a non positive occurrence of the datatypes
being declared
```

For technical reasons, allowing these datatypes could make it possible to undermine Lean's internal logic, making it unsuitable for use as a theorem prover.

Forgetting an argument to an inductive type can also yield a confusing message. For example, when the argument `α` is not passed to `MyType` in `ctor`'s type:

```
inductive MyType (α : Type) : Type where
  | ctor : α → MyType
```

Lean replies with the following error:

```
type expected, got
  (MyType : Type → Type)
```

The error message is saying that `MyType`'s type, which is `Type → Type`, does not itself describe types. `MyType` requires an argument to become an actual honest-to-goodness type.

The same message can appear when type arguments are omitted in other contexts, such as in a type signature for a definition:

```
inductive MyType (α : Type) : Type where
  | ctor : α → MyType α

def ofFive : MyType := ctor 5
```

Exercises

- Write a function to find the last entry in a list. It should return an `Option`.
- Write a function that finds the first entry in a list that satisfies a given predicate. Start the definition with `def List.findFirst? {α : Type} (xs : List α) (predicate : α → Bool) : Option α :=`
- Write a function `Prod.swap` that swaps the two fields in a pair. Start the definition with `def Prod.swap {α β : Type} (pair : α × β) : β × α :=`
- Rewrite the `PetName` example to use a custom datatype and compare it to the version that uses `Sum`.
- Write a function `zip` that combines two lists into a list of pairs. The resulting list should be as long as the shortest input list. Start the definition with `def zip {α β : Type} (xs : List α) (ys : List β) : List (α × β) :=`.
- Write a polymorphic function `take` that returns the first n entries in a list, where n is a `Nat`. If the list contains fewer than `n` entries, then the resulting list should be the input list. `#eval take 3 ["bolete", "oyster"]` should yield `["bolete", "oyster"]`, and `#eval take 1 ["bolete", "oyster"]` should yield `["bolete"]`.

- Using the analogy between types and arithmetic, write a function that distributes products over sums. In other words, it should have type `α × (β ⊕ γ) → (α × β) ⊕ (α × γ)`.
- Using the analogy between types and arithmetic, write a function that turns multiplication by two into a sum. In other words, it should have type `Bool × α → α ⊕ α`.

Additional Conveniences

Lean contains a number of convenience features that make programs much more concise.

Automatic Implicit Arguments

When writing polymorphic functions in Lean, it is typically not necessary to list all the implicit arguments. Instead, they can simply be mentioned. If Lean can determine their type, then they are automatically inserted as implicit arguments. In other words, the previous definition of `length`:

```
def length {α : Type} (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

can be written without `{α : Type}`:

```
def length (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This can greatly simplify highly polymorphic definitions that take many implicit arguments.

Pattern-Matching Definitions

When defining functions with `def`, it is quite common to name an argument and then immediately use it with pattern matching. For instance, in `length`, the argument `xs` is used only in `match`. In these situations, the cases of the `match` expression can be written directly, without naming the argument at all.

The first step is to move the arguments' types to the right of the colon, so the return type is a function type. For instance, the type of `length` is `List α → Nat`. Then, replace the `:=` with each case of the pattern match:

```
def length : List α → Nat
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This syntax can also be used to define functions that take more than one argument. In this case, their patterns are separated by commas. For instance, `drop` takes a number n and a list, and returns the list after removing the first n entries.

```
def drop : Nat → List α → List α
  | Nat.zero, xs => xs
  | _, [] => []
  | Nat.succ n, x :: xs => drop n xs
```

Named arguments and patterns can also be used in the same definition. For instance, a function that takes a default value and an optional value, and returns the default when the optional value is `none`, can be written:

```
def fromOption (default : α) : Option α → α
  | none => default
  | some x => x
```

This function is called `Option.getD` in the standard library, and can be called with dot notation:

```
#eval (some "salmonberry").getD ""
```

```
"salmonberry"
```

```
#eval none.getD ""
```

```
""
```

Local Definitions

It is often useful to name intermediate steps in a computation. In many cases, intermediate values represent useful concepts all on their own, and naming them explicitly can make the program easier to read. In other cases, the intermediate value is used more than once. As in most other languages, writing down the same code twice in Lean causes it to be computed twice, while saving the result in a variable leads to the result of the computation being saved and re-used.

For instance, `unzip` is a function that transforms a list of pairs into a pair of lists. When the list of pairs is empty, then the result of `unzip` is a pair of empty lists. When the list of pairs has a pair at its head, then the two fields of the pair are added to the result of unzipping the rest of the list. This definition of `unzip` follows that description exactly:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    (x :: (unzip xys).fst, y :: (unzip xys).snd)
```

Unfortunately, there is a problem: this code is slower than it needs to be. Each entry in the list of pairs leads to two recursive calls, which makes this function take exponential time. However, both recursive calls will have the same result, so there is no reason to make the recursive call twice.

In Lean, the result of the recursive call can be named, and thus saved, using `let`. Local definitions with `let` resemble top-level definitions with `def`: it takes a name to be locally defined, arguments if desired, a type signature, and then a body following `:=`. After the local definition, the expression in which the local definition is available (called the *body* of the `let`-expression) must be on a new line, starting at a column in the file that is less than or equal to that of the `let` keyword. For instance, `let` can be used in `unzip` like this:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped : List α × List β := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

To use `let` on a single line, separate the local definition from the body with a semicolon.

Local definitions with `let` may also use pattern matching when one pattern is enough to match all cases of a datatype. In the case of `unzip`, the result of the recursive call is a pair. Because pairs have only a single constructor, the name `unzipped` can be replaced with a pair pattern:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let (xs, ys) : List α × List β := unzip xys
    (x :: xs, y :: ys)
```

Judicious use of patterns with `let` can make code easier to read, compared to writing the accessor calls by hand.

The biggest difference between `let` and `def` is that recursive `let` definitions must be explicitly indicated by writing `let rec`. For instance, one way to reverse a list involves a recursive helper function, as in this definition:

```
def reverse (xs : List α) : List α :=
  let rec helper : List α → List α → List α
    | [], soFar => soFar
    | y :: ys, soFar => helper ys (y :: soFar)
  helper xs []
```

The helper function walks down the input list, moving one entry at a time over to `soFar`. When it reaches the end of the input list, `soFar` contains a reversed version of the input.

Type Inference

In many situations, Lean can automatically determine an expression's type. In these cases, explicit types may be omitted from both top-level definitions (with `def`) and local definitions (with `let`). For instance, the recursive call to `unzip` does not need an annotation:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

As a rule of thumb, omitting the types of literal values (like strings and numbers) usually works, although Lean may pick a type for literal numbers that is more specific than the intended type. Lean can usually determine a type for a function application, because it already knows the argument types and the return type. Omitting return types for function definitions will often work, but function arguments typically require annotations. Definitions that are not functions, like `unzipped` in the example, do not need type annotations if their bodies do not need type annotations, and the body of this definition is a function application.

Omitting the return type for `unzip` is possible when using an explicit `match` expression:

```
def unzip (pairs : List (α × β)) :=
  match pairs with
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

Generally speaking, it is a good idea to err on the side of too many, rather than too few, type annotations. First off, explicit types communicate assumptions about the code to readers. Even if Lean can determine the type on its own, it can still be easier to read code without having to repeatedly query Lean for type information. Secondly, explicit types help localize errors. The more explicit a program is about its types, the more informative the error messages can be. This is especially important in a language like Lean that has a very expressive type system. Thirdly, explicit types make it easier to write the program in the first place. The type is a specification, and the compiler's feedback can be a helpful tool in writing a program that meets the specification. Finally, Lean's type inference is a best-effort system. Because Lean's type system is so expressive, there is no "best" or most general type to find for all expressions. This means that even if you get a type, there's no guarantee that it's the *right* type for a given application. For instance, `14` can be a `Nat` or an `Int`:

```
#check 14
```

```
14 : Nat
```

```
#check (14 : Int)
```

```
14 : Int
```

Missing type annotations can give confusing error messages. Omitting all types from the definition of `unzip`:

```
def unzip pairs :=
  match pairs with
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

leads to a message about the `match` expression:

```
invalid match-expression, pattern contains metavariables
  []
```

This is because `match` needs to know the type of the value being inspected, but that type was not available. A "metavariable" is an unknown part of a program, written `?m.XYZ` in error messages—they are described in the section on Polymorphism. In this program, the type annotation on the argument is required.

Even some very simple programs require type annotations. For instance, the identity function just returns whatever argument it is passed. With argument and type annotations, it looks like this:

```
def id (x : α) : α := x
```

Lean is capable of determining the return type on its own:

```
def id (x : α) := x
```

Omitting the argument type, however, causes an error:

```
def id x := x
```

```
failed to infer binder type
```

In general, messages that say something like "failed to infer" or that mention metavariables are often a sign that more type annotations are necessary. Especially while still learning Lean, it is useful to provide most types explicitly.

Simultaneous Matching

Pattern-matching expressions, just like pattern-matching definitions, can match on multiple values at once. Both the expressions to be inspected and the patterns that they match against are written with commas between them, similarly to the syntax used for definitions. Here is a version of `drop` that uses simultaneous matching:

```
def drop (n : Nat) (xs : List α) : List α :=
  match n, xs with
  | Nat.zero, ys => ys
  | _, [] => []
  | Nat.succ n , y :: ys => drop n ys
```

Natural Number Patterns

In the section on datatypes and patterns, `even` was defined like this:

```
def even (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (even k)
```

Just as there is special syntax to make list patterns more readable than using `List.cons` and `List.nil` directly, natural numbers can be matched using literal numbers and `+`. For instance, `even` can also be defined like this:

```
def even : Nat → Bool
  | 0 => true
  | n + 1 => not (even n)
```

In this notation, the arguments to the `+` pattern serve different roles. Behind the scenes, the left argument (`n` above) becomes an argument to some number of `Nat.succ` patterns, and the right argument (`1` above) determines how many `Nat.succ`s to wrap around the pattern. The explicit patterns in `halve`, which divides a `Nat` by two and drops the remainder:

```
def halve : Nat → Nat
  | Nat.zero => 0
  | Nat.succ Nat.zero => 0
  | Nat.succ (Nat.succ n) => halve n + 1
```

can be replaced by numeric literals and `+`:

```
def halve : Nat → Nat
  | 0 => 0
  | 1 => 0
  | n + 2 => halve n + 1
```

Behind the scenes, both definitions are completely equivalent. Remember: `halve n + 1` is equivalent to `(halve n) + 1`, not `halve (n + 1)`.

When using this syntax, the second argument to `+` should always be a literal `Nat`. Even though addition is commutative, flipping the arguments in a pattern can result in errors like the following:

```
def halve : Nat → Nat
  | 0 => 0
  | 1 => 0
  | 2 + n => halve n + 1
```

```
invalid patterns, `n` is an explicit pattern variable, but it only occurs in
positions that are inaccessible to pattern matching
  .(Nat.add 2 n)
```

This restriction enables Lean to transform all uses of the `+` notation in a pattern into uses of the underlying `Nat.succ`, keeping the language simpler behind the scenes.

Anonymous Functions

Functions in Lean need not be defined at the top level. As expressions, functions are produced with the `fun` syntax. Function expressions begin with the keyword `fun`, followed by one or more arguments, which are separated from the return expression using `=>`. For instance, a function that adds one to a number can be written:

```
#check fun x => x + 1
```

```
fun x => x + 1 : Nat → Nat
```

Type annotations are written the same way as on `def`, using parentheses and colons:

```
#check fun (x : Int) => x + 1
```

```
fun x => x + 1 : Int → Int
```

Similarly, implicit arguments may be written with curly braces:

```
#check fun {α : Type} (x : α) => x
```

```
fun {α} x => x : {α : Type} → α → α
```

This style of anonymous function expression is often referred to as a *lambda expression*, because the typical notation used in mathematical descriptions of programming languages uses the Greek letter λ (lambda) where Lean has the keyword `fun`. Even though Lean does permit `λ` to be used instead of `fun`, it is most common to write `fun`.

Anonymous functions also support the multiple-pattern style used in `def`. For instance, a function that returns the predecessor of a natural number if it exists can be written:

```
#check fun
  | 0 => none
  | n + 1 => some n
```

```
fun x =>
  match x with
  | 0 => none
  | Nat.succ n => some n : Nat → Option Nat
```

Note that Lean's own description of the function has a named argument and a `match` expression. Many of Lean's convenient syntactic shorthands are expanded to simpler syntax behind the scenes, and the abstraction sometimes leaks.

Definitions using `def` that take arguments may be rewritten as function expressions. For instance, a function that doubles its argument can be written as follows:

```
def double : Nat → Nat := fun
  | 0 => 0
  | k + 1 => double k + 2
```

When an anonymous function is very simple, like `fun x => x + 1`, the syntax for creating the function can be fairly verbose. In that particular example, six non-whitespace characters are used to introduce the function, and its body consists of only three non-whitespace characters. For these simple cases, Lean provides a shorthand. In an expression surrounded by parentheses, a centered dot character `·` can stand for an argument, and the expression inside the parentheses becomes the function's body. That particular function can also be written `(· + 1)`.

The centered dot always creates a function out of the *closest* surrounding set of parentheses. For instance, `(· + 5, 3)` is a function that returns a pair of numbers, while `((· + 5), 3)` is a pair of a function and a number. If multiple dots are used, then they become arguments from left to right:

```
(· , ·) 1 2
===>
(1, ·) 2
===>
(1, 2)
```

Anonymous functions can be applied in precisely the same way as functions defined using `def` or `let`. The command `#eval (fun x => x + x) 5` results in:

```
10
```

while `#eval (· * 2) 5` results in:

```
10
```

Namespaces

Each name in Lean occurs in a *namespace*, which is a collection of names. Names are placed in namespaces using `.`, so `List.map` is the name `map` in the `List` namespace. Names in different namespaces do not conflict with each other, even if they are otherwise identical. This means that `List.map` and `Array.map` are different names. Namespaces may be nested, so `Project.Frontend.User.loginTime` is the name `loginTime` in the nested namespace `Project.Frontend.User`.

Names can be directly defined within a namespace. For instance, the name `double` can be defined in the `Nat` namespace:

```
def Nat.double (x : Nat) : Nat := x + x
```

Because `Nat` is also the name of a type, dot notation is available to call `Nat.double` on expressions with type `Nat`:

```
#eval (4 : Nat).double
```

```
8
```

In addition to defining names directly in a namespace, a sequence of declarations can be placed in a namespace using the `namespace` and `end` commands. For instance, this defines `triple` and `quadruple` in the namespace `NewNamespace`:

```
namespace NewNamespace
def triple (x : Nat) : Nat := 3 * x
def quadruple (x : Nat) : Nat := 2 * x + 2 * x
end NewNamespace
```

To refer to them, prefix their names with `NewNamespace.`:

```
#check NewNamespace.triple
```

```
NewNamespace.triple (x : Nat) : Nat
```

```
#check NewNamespace.quadruple
```

```
NewNamespace.quadruple (x : Nat) : Nat
```

Namespaces may be *opened*, which allows the names in them to be used without explicit qualification. Writing `open MyNamespace in` before an expression causes the contents of `MyNamespace` to be available in the expression. For example, `timesTwelve` uses both `quadruple` and `triple` after opening `NewNamespace`:

```
def timesTwelve (x : Nat) :=
  open NewNamespace in
  quadruple (triple x)
```

Namespaces can also be opened prior to a command. This allows all parts of the command to refer to the contents of the namespace, rather than just a single expression. To do this, place the `open ... in` prior to the command.

```
open NewNamespace in
#check quadruple
```

```
NewNamespace.quadruple (x : Nat) : Nat
```

Function signatures show the name's full namespace. Namespaces may additionally be opened for *all* following commands for the rest of the file. To do this, simply omit the `in` from a top-level usage of `open`.

if let

When consuming values that have a sum type, it is often the case that only a single constructor is of interest. For instance, given this type that represents a subset of Markdown inline elements:

```
inductive Inline : Type where
  | lineBreak
  | string : String → Inline
  | emph : Inline → Inline
  | strong : Inline → Inline
```

a function that recognizes string elements and extracts their contents can be written:

```
def Inline.string? (inline : Inline) : Option String :=
  match inline with
  | Inline.string s => some s
  | _ => none
```

An alternative way of writing this function's body uses `if` together with `let`:

```
def Inline.string? (inline : Inline) : Option String :=
  if let Inline.string s := inline then
    some s
  else none
```

This is very much like the pattern-matching `let` syntax. The difference is that it can be used with sum types, because a fallback is provided in the `else` case. In some contexts, using `if let` instead of `match` can make code easier to read.

Positional Structure Arguments

The section on structures presents two ways of constructing structures:

1. The constructor can be called directly, as in `Point.mk 1 2`.
2. Brace notation can be used, as in `{ x := 1, y := 2 }`.

In some contexts, it can be convenient to pass arguments positionally, rather than by name, but without naming the constructor directly. For instance, defining a variety of similar structure types can help keep domain concepts separate, but the natural way to read the code may treat each of them as being essentially a tuple. In these contexts, the arguments can be enclosed in angle brackets `⟨` and `⟩`. A `Point` can be written `⟨1, 2⟩`. Be careful! Even though they look like the less-than sign `<` and greater-than sign `>`, these brackets are different. They can be input using `\<` and `\>`, respectively.

Just as with the brace notation for named constructor arguments, this positional syntax can only be used in a context where Lean can determine the structure's type, either from a type annotation or from other type information in the program. For instance, `#eval ⟨1, 2⟩` yields the following error:

```
invalid constructor ⟨...⟩, expected type must be an inductive type
  ?m.35347
```

The metavariable in the error is because there is no type information available. Adding an annotation, such as in `#eval (⟨1, 2⟩ : Point)`, solves the problem:

```
{ x := 1.000000, y := 2.000000 }
```

String Interpolation

In Lean, prefixing a string with `s!` triggers *interpolation*, where expressions contained in curly braces inside the string are replaced with their values. This is similar to `f`-strings in Python and `$`-prefixed strings in C#. For instance,

```
#eval s!"three fives is {NewNamespace.triple 5}"
```

yields the output

```
"three fives is 15"
```

Not all expressions can be interpolated into a string. For instance, attempting to interpolate a function results in an error.

```
#check s!"three fives is {NewNamespace.triple}"
```

yields the output

```
failed to synthesize instance
  ToString (Nat → Nat)
```

This is because there is no standard way to convert functions into strings. The Lean compiler maintains a table that describes how to convert values of various types into strings, and the message `failed to synthesize instance` means that the Lean compiler didn't find an entry in this table for the given type. This uses the same language feature as the `deriving Repr` syntax that was described in the section on structures.

Summary

Evaluating Expressions

In Lean, computation occurs when expressions are evaluated. This follows the usual rules of mathematical expressions: sub-expressions are replaced by their values following the usual order of operations, until the entire expression has become a value. When evaluating an `if` or a `match`, the expressions in the branches are not evaluated until the value of the condition or the match subject has been found.

Once they have been given a value, variables never change. Similarly to mathematics but unlike most programming languages, Lean variables are simply placeholders for values, rather than addresses to which new values can be written. Variables' values may come from global definitions with `def`, local definitions with `let`, as named arguments to functions, or from pattern matching.

Functions

Functions in Lean are first-class values, meaning that they can be passed as arguments to other functions, saved in variables, and used like any other value. Every Lean function takes exactly one argument. To encode a function that takes more than one argument, Lean uses a technique called currying, where providing the first argument returns a function that expects the remaining arguments. To encode a function that takes no arguments, Lean uses the `Unit` type, which is the least informative possible argument.

There are three primary ways of creating functions:

1. Anonymous functions are written using `fun`. For instance, a function that swaps the fields of a `Point` can be written `fun (point : Point) => { x := point.y, y := point.x : Point}`
2. Very simple anonymous functions are written by placing one or more centered dots `·` inside of parentheses. Each centered dot becomes an argument to the function, and the parentheses delimit its body. For instance, a function that subtracts one from its argument can be written as `(· - 1)` instead of as `fun x => x - 1`.
3. Functions can be defined using `def` or `let` by adding an argument list or by using pattern-matching notation.

Types

Lean checks that every expression has a type. Types, such as `Int`, `Point`, `{α : Type} → Nat → α → List α`, and `Option (String ⊕ (Nat × String))`, describe the values that may eventually be found for an expression. Like other languages, types in Lean can express lightweight specifications for programs that are checked by the Lean compiler, obviating the need for certain classes of unit test. Unlike most languages, Lean's types can also express arbitrary mathematics, unifying the worlds of programming and theorem proving. While using Lean for proving theorems is mostly out of scope for this book, *Theorem Proving in Lean 4* contains more information on this topic.

Some expressions can be given multiple types. For instance, `3` can be an `Int` or a `Nat`. In Lean, this should be understood as two separate expressions, one with type `Nat` and one with type `Int`, that happen to be written in the same way, rather than as two different types for the same thing.

Lean is sometimes able to determine types automatically, but types must often be provided by the user. This is because Lean's type system is so expressive. Even when Lean can find a type, it may not find the desired type—`3` could be intended to be used as an `Int`, but Lean will give it the type `Nat` if there are no further constraints. In general, it is a good idea to write most types explicitly, only letting Lean fill out the very obvious types. This improves Lean's error messages and helps make programmer intent more clear.

Some functions or datatypes take types as arguments. They are called *polymorphic*. Polymorphism allows programs such as one that calculates the length of a list without caring what type the entries in the list have. Because types are first class in Lean, polymorphism does not require any special syntax, so types are passed just like other arguments. Giving an argument a name in a function type allows later types to mention that argument, and the type of applying that function to an argument is found by replacing the argument's name with the argument's value.

Structures and Inductive Types

Brand new datatypes can be introduced to Lean using the `structure` or `inductive` features. These new types are not considered to be equivalent to any other type, even if their definitions are otherwise identical. Datatypes have *constructors* that explain the ways in which their values can be constructed, and each constructor takes some number of arguments. Constructors in Lean are not the same as constructors in object-oriented languages: Lean's constructors are inert holders of data, rather than active code that initializes an allocated object.

Typically, `structure` is used to introduce a product type (that is, a type with just one constructor that takes any number of arguments), while `inductive` is used to introduce a sum type (that is, a type with many distinct constructors). Datatypes defined with `structure` are provided with one accessor function for each of the constructor's arguments. Both structures and inductive datatypes may be consumed with pattern matching, which exposes the values stored inside of constructors using a subset of the syntax used to call said constructors. Pattern matching means that knowing how to create a value implies knowing how to consume it.

Recursion

A definition is recursive when the name being defined is used in the definition itself. Because Lean is an interactive theorem prover in addition to being a programming language, there are certain restrictions placed on recursive definitions. In Lean's logical side, circular definitions could lead to logical inconsistency.

In order to ensure that recursive definitions do not undermine the logical side of Lean, Lean must be able to prove that all recursive functions terminate, no matter what arguments they are called with. In practice, this means either that recursive calls are all performed on a structurally-smaller piece of the input, which ensures that there is always progress towards a base case, or that users must provide some other evidence that the function always terminates. Similarly, recursive inductive types are not allowed to have a constructor that takes a function *from* the type as an argument, because this would make it possible to encode non-terminating functions.

Hello, World!

While Lean has been designed to have a rich interactive environment in which programmers can get quite a lot of feedback from the language without leaving the confines of their favorite text editor, it is also a language in which real programs can be written. This means that it also has a batch-mode compiler, a build system, a package manager, and all the other tools that are necessary for writing programs.

While the previous chapter presented the basics of functional programming in Lean, this chapter explains how to start a programming project, compile it, and run the result. Programs that run and interact with their environment (e.g. by reading input from standard input or creating files) are difficult to reconcile with the understanding of computation as the evaluation of mathematical expressions. In addition to a description of the Lean build tools, this chapter also provides a way to think about functional programs that interact with the world.

Running a Program

The simplest way to run a Lean program is to use the `--run` option to the Lean executable. Create a file called `Hello.lean` and enter the following contents:

```
def main : IO Unit := IO.println "Hello, world!"
```

Then, from the command line, run:

```
lean --run Hello.lean
```

The program displays `Hello, world!` and exits.

Anatomy of a Greeting

When Lean is invoked with the `--run` option, it invokes the program's `main` definition. In programs that do not take command-line arguments, `main` should have type `IO Unit`. This means that `main` is not a function, because there are no arrows (`→`) in its type. Instead of being a function that has side effects, `main` consists of a description of effects to be carried out.

As discussed in the preceding chapter, `Unit` is the simplest inductive type. It has a single constructor called `unit` that takes no arguments. Languages in the C tradition have a notion of a `void` function that does not return any value at all. In Lean, all functions take an argument and return a value, and the lack of interesting arguments or return values can be signaled by using the `Unit` type instead. If `Bool` represents a single bit of information, `Unit` represents zero bits of information.

`IO α` is the type of a program that, when executed, will either throw an exception or return a value of type `α`. During execution, this program may have side effects. These programs are referred to as `IO` *actions*. Lean distinguishes between *evaluation* of expressions, which strictly adheres to the mathematical model of substitution of values for variables and reduction of sub-expressions without side effects, and *execution* of `IO` actions, which rely on an external system to interact with the world. `IO.println` is a function from strings to `IO` actions that, when executed, write the given string to standard output. Because this action doesn't read any interesting information from the environment in the process of emitting the string, `IO.println` has type `String → IO Unit`. If it did return something interesting, then that would be indicated by the `IO` action having a type other than `Unit`.

Functional Programming vs Effects

Lean's model of computation is based on the evaluation of mathematical expressions, in which variables are given exactly one value that does not change over time. The result of evaluating an expression does not change, and evaluating the same expression again will always yield the same result.

On the other hand, useful programs must interact with the world. A program that performs neither input nor output can't ask a user for data, create files on disk, or open network connections. Lean is written in itself, and the Lean compiler certainly reads files, creates files, and interacts with text editors. How can a language in which the same expression always yields the same result support programs that read files from disk, when the contents of these files might change over time?

This apparent contradiction can be resolved by thinking a bit differently about side effects. Imagine a café that sells coffee and sandwiches. This café has two employees: a cook who fulfills orders, and a worker at the counter who interacts with customers and places order slips. The cook is a surly person, who really prefers not to have any contact with the world outside, but who is very good at consistently delivering the food and drinks that the café is known for. In order to do this, however, the cook needs peace and quiet, and can't be disturbed with conversation. The counter worker is friendly, but completely incompetent in the kitchen. Customers interact with the counter worker, who delegates all actual cooking to the cook. If the cook has a question for a customer, such as clarifying an allergy, they send a little note to the counter worker, who interacts with the customer and passes a note back to the cook with the result.

In this analogy, the cook is the Lean language. When provided with an order, the cook faithfully and consistently delivers what is requested. The counter worker is the surrounding run-time system that interacts with the world and can accept payments, dispense food, and have conversations with customers. Working together, the two employees serve all the functions of the restaurant, but their responsibilities are divided, with each performing the tasks that they're best at. Just as keeping customers away allows the cook to focus on making truly excellent coffee and sandwiches, Lean's lack of side effects allows programs to be used as part of formal mathematical proofs. It also helps programmers understand the parts of the program in isolation from each other, because there are no hidden state changes that create subtle coupling between components. The cook's notes represent `IO` actions that are produced by evaluating Lean expressions, and the counter worker's replies are the values that are passed back from effects.

This model of side effects is quite similar to how the overall aggregate of the Lean language, its compiler, and its run-time system (RTS) work. Primitives in the run-time system, written in C, implement all the basic effects. When running a program, the RTS invokes the `main` action, which returns new `IO` actions to the RTS for execution. The RTS executes these actions, delegating to the user's Lean code to carry out computations. From the internal perspective of Lean, programs are free of side effects, and `IO` actions are just descriptions of tasks to be carried out. From the external perspective of the program's user, there is a layer of side effects that create an interface to the program's core logic.

Real-World Functional Programming

The other useful way to think about side effects in Lean is by considering `IO` actions to be functions that take the entire world as an argument and return a value paired with a new world. In this case, reading a line of text from standard input *is* a pure function, because a different world is provided as an argument each time. Writing a line of text to standard output is a pure function, because the world that the function returns is different from the one that it began with. Programs do need to be careful to never re-use the world, nor to fail to return a new world—this would amount to time travel or the end of the world, after all. Careful abstraction boundaries can make this style of programming safe. If every primitive `IO` action accepts one world and returns a new one, and they can only be combined with tools that preserve this invariant, then the problem cannot occur.

This model cannot be implemented. After all, the entire universe cannot be turned into a Lean value and placed into memory. However, it is possible to implement a variation of this model with an abstract token that stands for the world. When the program is started, it is provided with a world token. This token is then passed on to the IO primitives, and their returned tokens are similarly passed to the next step. At the end of the program, the token is returned to the operating system.

This model of side effects is a good description of how `IO` actions as descriptions of tasks to be carried out by the RTS are represented internally in Lean. The actual functions that transform the real world are behind an abstraction barrier. But real programs typically consist of a sequence of effects, rather than just one. To enable programs to use multiple effects, there is a sub-language of Lean called `do` notation that allows these primitive `IO` actions to be safely composed into a larger, useful program.

Combining `IO` Actions

Most useful programs accept input in addition to producing output. Furthermore, they may take decisions based on input, using the input data as part of a computation. The following program, called `HelloName.lean`, asks the user for their name and then greets them:

```
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace

  stdout.putStrLn s!"Hello, {name}!"
```

In this program, the `main` action consists of a `do` block. This block contains a sequence of *statements*, which can be both local variables (introduced using `let`) and actions that are to be executed. Just as SQL can be thought of as a special-purpose language for interacting with databases, the `do` syntax can be thought of as a special-purpose sub-language within Lean that is dedicated to modeling imperative programs. `IO` actions that are built with a `do` block are executed by executing the statements in order.

This program can be run in the same manner as the prior program:

```
lean --run HelloName.lean
```

If the user responds with `David`, a session of interaction with the program reads:

```
How would you like to be addressed?
David
Hello, David!
```

The type signature line is just like the one for `Hello.lean`:

```
def main : IO Unit := do
```

The only difference is that it ends with the keyword `do`, which initiates a sequence of commands. Each indented line following the keyword `do` is part of the same sequence of commands.

The first two lines, which read:

```
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
```

retrieve the `stdin` and `stdout` handles by executing the library actions `IO.getStdin` and `IO.getStdout`, respectively. In a `do` block, `let` has a slightly different meaning than in an ordinary expression. Ordinarily, the local definition in a `let` can be used in just one expression, which immediately follows the local definition. In a `do` block, local bindings introduced by `let` are available in all statements in the remainder of the `do` block, rather than just the next one. Additionally, `let` typically connects the name being defined to its definition using `:=`, while some `let` bindings in `do` use a left arrow (`←` or `<-`) instead. Using an arrow means that the value of the expression is an `IO` action that should be executed, with the result of the action saved in the local variable. In other words, if the expression to the right of the arrow has type `IO α`, then the variable has type `α` in the remainder of the `do` block. `IO.getStdin` and `IO.getStdout` are `IO` actions in order to allow `stdin` and `stdout` to be locally overridden in a program, which can be convenient. If they were global variables as in C, then there would be no meaningful way to override them, but `IO` actions can return different values each time they are executed.

The next part of the `do` block is responsible for asking the user for their name:

```
  stdout.putStrLn "How would you like to be addressed?"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace
```

The first line writes the question to `stdout`, the second line requests input from `stdin`, and the third line removes the trailing newline (plus any other trailing whitespace) from the input line. The definition of `name` uses `:=`, rather than `←`, because `String.dropRightWhile` is an ordinary function on strings, rather than an `IO` action.

Finally, the last line in the program is:

```
  stdout.putStrLn s!"Hello, {name}!"
```

It uses string interpolation to insert the provided name into a greeting string, writing the result to `stdout`.

Step By Step

A `do` block can be executed one line at a time. Start with the program from the prior section:

```
let stdin ← IO.getStdin
let stdout ← IO.getStdout
stdout.putStrLn "How would you like to be addressed?"
let input ← stdin.getLine
let name := input.dropRightWhile Char.isWhitespace
stdout.putStrLn s!"Hello, {name}!"
```

Standard IO

The first line is `let stdin ← IO.getStdin`, while the remainder is:

```
let stdout ← IO.getStdout
stdout.putStrLn "How would you like to be addressed?"
let input ← stdin.getLine
let name := input.dropRightWhile Char.isWhitespace
stdout.putStrLn s!"Hello, {name}!"
```

To execute a `let` statement that uses a `←`, start by evaluating the expression to the right of the arrow (in this case, `IO.getStdIn`). Because this expression is just a variable, its value is looked up. The resulting value is a built-in primitive `IO` action. The next step is to execute this `IO` action, resulting in a value that represents the standard input stream, which has type `IO.FS.Stream`. Standard input is then associated with the name to the left of the arrow (here `stdin`) for the remainder of the `do` block.

Executing the second line, `let stdout ← IO.getStdout`, proceeds similarly. First, the expression `IO.getStdout` is evaluated, yielding an `IO` action that will return the standard output. Next, this action is executed, actually returning the standard output. Finally, this value is associated with the name `stdout` for the remainder of the `do` block.

Asking a Question

Now that `stdin` and `stdout` have been found, the remainder of the block consists of a question and an answer:

```
  stdout.putStrLn "How would you like to be addressed?"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace
  stdout.putStrLn s!"Hello, {name}!"
```

The first statement in the block, `stdout.putStrLn "How would you like to be addressed?"`, consists of an expression. To execute an expression, it is first evaluated. In this case, `IO.FS.Stream.putStrLn` has type `IO.FS.Stream → String → IO Unit`. This means that it is a function that accepts a stream and a string, returning an `IO` action. The expression uses accessor notation for a function call. This function is applied to two arguments: the standard output stream and a string. The value of the expression is an `IO` action that will write the string and a newline character to the output stream. Having found this value, the next step is to execute it, which causes the string and newline to actually be written to `stdout`. Statements that consist only of expressions do not introduce any new variables.

The next statement in the block is `let input ← stdin.getLine`. `IO.FS.Stream.getLine` has type `IO.FS.Stream → IO String`, which means that it is a function from a stream to an `IO` action that will return a string. Once again, this is an example of accessor notation. This `IO` action is executed, and the program waits until the user has typed a complete line of input. Assume the user writes "`David`". The resulting line (`"David\n"`) is associated with `input`, where the escape sequence `\n` denotes the newline character.

```
  let name := input.dropRightWhile Char.isWhitespace
  stdout.putStrLn s!"Hello, {name}!"
```

The next line, `let name := input.dropRightWhile Char.isWhitespace`, is a `let` statement. Unlike the other `let` statements in this program, it uses `:=` instead of `←`. This means that the expression will be evaluated, but the resulting value need not be an `IO` action and will not be executed. In this case, `String.dropRightWhile` takes a string and a predicate over characters and returns a new string from which all the characters at the end of the string that satisfy the predicate have been removed. For example,

```
#eval "Hello!!!".dropRightWhile (· == '!')
```

yields

```
"Hello"
```

and

```
#eval "Hello...   ".dropRightWhile (fun c => not (c.isAlphanum))
```

yields

```
"Hello"
```

in which all non-alphanumeric characters have been removed from the right side of the string. In the current line of the program, whitespace characters (including the newline) are removed from the right side of the input string, resulting in `"David"`, which is associated with `name` for the remainder of the block.

Greeting the User

All that remains to be executed in the `do` block is a single statement:

```
  stdout.putStrLn s!"Hello, {name}!"
```

The string argument to `putStrLn` is constructed via string interpolation, yielding the string `"Hello, David!"`. Because this statement is an expression, it is evaluated to yield an `IO` action that will print this string with a newline to standard output. Once the expression has been evaluated, the resulting `IO` action is executed, resulting in the greeting.

`IO` Actions as Values

In the above description, it can be difficult to see why the distinction between evaluating expressions and executing `IO` actions is necessary. After all, each action is executed immediately after it is produced. Why not simply carry out the effects during evaluation, as is done in other languages?

The answer is twofold. First off, separating evaluation from execution means that programs must be explicit about which functions can have side effects. Because the parts of the program that do not have effects are much more amenable to mathematical reasoning, whether in the heads of programmers or using Lean's facilities for formal proof, this separation can make it easier to avoid bugs. Secondly, not all `IO` actions need be executed at the time that they come into existence. The ability to mention an action without carrying it out allows ordinary functions to be used as control structures.

For instance, the function `twice` takes an `IO` action as its argument, returning a new action that will execute the first one twice.

```
def twice (action : IO Unit) : IO Unit := do
  action
  action
```

For instance, executing

```
twice (IO.println "shy")
```

results in

```
shy
shy
```

being printed. This can be generalized to a version that runs the underlying action any number of times:

```
def nTimes (action : IO Unit) : Nat → IO Unit
  | 0 => pure ()
  | n + 1 => do
    action
    nTimes action n
```

In the base case for `Nat.zero`, the result is `pure ()`. The function `pure` creates an `IO` action that has no side effects, but returns `pure`'s argument, which in this case is the constructor for `Unit`. As an action that does nothing and returns nothing interesting, `pure ()` is at the same time utterly boring and very useful. In the recursive step, a `do` block is used to create an action that first executes `action` and then executes the result of the recursive call. Executing `nTimes (IO.println "Hello") 3` causes the following output:

```
Hello
Hello
Hello
```

In addition to using functions as control structures, the fact that `IO` actions are first-class values means that they can be saved in data structures for later execution. For instance, the function `countdown` takes a `Nat` and returns a list of unexecuted `IO` actions, one for each `Nat`:

```
def countdown : Nat → List (IO Unit)
  | 0 => [IO.println "Blast off!"]
  | n + 1 => IO.println s!"{n + 1}" :: countdown n
```

This function has no side effects, and does not print anything. For example, it can be applied to an argument, and the length of the resulting list of actions can be checked:

```
def from5 : List (IO Unit) := countdown 5
```

This list contains six elements (one for each number, plus a `"Blast off!"` action for zero):

```
#eval from5.length
```

```
6
```

The function `runActions` takes a list of actions and constructs a single action that runs them all in order:

```
def runActions : List (IO Unit) → IO Unit
  | [] => pure ()
  | act :: actions => do
    act
    runActions actions
```

Its structure is essentially the same as that of `nTimes`, except instead of having one action that is executed for each `Nat.succ`, the action under each `List.cons` is to be executed. Similarly, `runActions` does not itself run the actions. It creates a new action that will run them, and that action must be placed in a position where it will be executed as a part of `main`:

```
def main : IO Unit := runActions from5
```

Running this program results in the following output:

```
5
4
3
2
1
Blast off!
```

What happens when this program is run? The first step is to evaluate `main`. That occurs as follows:

```
main
===>
runActions from5
===>
runActions (countdown 5)
===>
runActions
  [IO.println "5",
   IO.println "4",
   IO.println "3",
   IO.println "2",
   IO.println "1",
   IO.println "Blast off!"]
===>
do IO.println "5"
   IO.println "4"
   IO.println "3"
   IO.println "2"
   IO.println "1"
   IO.println "Blast off!"
   pure ()
```

The resulting `IO` action is a `do` block. Each step of the `do` block is then executed, one at a time, yielding the expected output. The final step, `pure ()`, does not have any effects, and it is only present because the definition of `runActions` needs a base case.

Exercise

Step through the execution of the following program on a piece of paper:

```
def main : IO Unit := do
  let englishGreeting := IO.println "Hello!"
  IO.println "Bonjour!"
  englishGreeting
```

While stepping through the program's execution, identify when an expression is being evaluated and when an `IO` action is being executed. When executing an `IO` action results in a side effect, write it down. After doing this, run the program with Lean and double-check that your predictions about the side effects were correct.

Starting a Project

As a program written in Lean becomes more serious, an ahead-of-time compiler-based workflow that results in an executable becomes more attractive. Like other languages, Lean has tools for building multiple-file packages and managing dependencies. The standard Lean build tool is called Lake (short for "Lean Make"), and it is configured in Lean. Just as Lean contains a special-purpose language for writing programs with effects (the `do` language), Lake contains a special-purpose language for configuring builds. These languages are referred to as *embedded domain-specific languages* (or sometimes *domain-specific embedded languages*, abbreviated EDSL or DSEL). They are *domain-specific* in the sense that they are used for a particular purpose, with concepts from some sub-domain, and they are typically not suitable for general-purpose programming. They are *embedded* because they occur inside another language's syntax. While Lean contains rich facilities for creating EDSLs, they are beyond the scope of this book.

First steps

To get started with a project that uses Lake, use the command `lake new greeting` in a directory that does not already contain a file or directory called `greeting`. This creates a directory called `greeting` that contains the following files:

- `Main.lean` is the file in which the Lean compiler will look for the `main` action.
- `Greeting.lean` is the scaffolding of a support library for the program.
- `lakefile.lean` contains the configuration that `lake` needs to build the application.
- `lean-toolchain` contains an identifier for the specific version of Lean that is used for the project.

Additionally, `lake new` initializes the project as a Git repository and configures its `.gitignore` file to ignore intermediate build products. Typically, the majority of the application logic will be in a collection of libraries for the program, while `Main.lean` will contain a small wrapper around these pieces that does things like parsing command lines and executing the central application logic. To create a project in an already-existing directory, run `lake init` instead of `lake new`.

By default, the library file `Greeting.lean` contains a single definition:

```
def hello := "world"
```

while the executable source `Main.lean` contains:

```
import «Greeting»

def main : IO Unit :=
  IO.println s!"Hello, {hello}!"
```

The `import` line makes the contents of `Greeting.lean` available in `Main.lean`. Placing guillemets around a name, as in `«Greeting»`, allow it to contain spaces or other characters that are normally not allowed in Lean names, and it allows reserved keywords such as `if` or `def` to be used as ordinary names by writing `«if»` or `«def»`. This prevents issues when the package name provided to `lake new` contains such characters.

To build the package, run the command `lake build`. After a number of build commands scroll by, the resulting binary has been placed in `build/bin`. Running `./build/bin/greeting` results in `Hello, world!`.

Lakefiles

A `lakefile.lean` describes a *package*, which is a coherent collection of Lean code for distribution, analogous to an `npm` or `nuget` package or a Rust crate. A package may contain any number of libraries or executables. While the documentation for Lake describes the available options in a lakefile, it makes use of a number of Lean features that have not yet been described here. The generated `lakefile.lean` contains the following:

```
import Lake
open Lake DSL

package «greeting» {
  -- add package configuration options here
}

lean_lib «Greeting» {
  -- add library configuration options here
}

@[default_target]
lean_exe «greeting» {
  root := `Main
}
```

This initial Lakefile consists of three items:

- a *package* declaration, named `greeting`,
- a *library* declaration, named `Greeting`, and
- an *executable*, also named `greeting`.

Each of these names is enclosed in guillemets to allow users more freedom in picking package names.

Each Lakefile will contain exactly one package, but any number of libraries or executables. Additionally, Lakefiles may contain *external libraries*, which are libraries not written in Lean to be statically linked with the resulting executable, *custom targets*, which are build targets that don't fit naturally into the library/executable taxonomy, *dependencies*, which are declarations of other Lean packages (either locally or from remote Git repositories), and *scripts*, which are essentially `IO` actions (similar to `main`), but that additionally have access to metadata about the package configuration. The items in the Lakefile allow things like source file locations, module hierarchies, and compiler flags to be configured. Generally speaking, however, the defaults are reasonable.

Libraries, executables, and custom targets are all called *targets*. By default, `lake build` builds those targets that are annotated with `@[default_target]`. This annotation is an *attribute*, which is metadata that can be associated with a Lean declaration. Attributes are similar to Java annotations or C# and Rust attributes. They are used pervasively throughout Lean. To build a target that is not annotated with `@[default_target]`, specify the target's name as an argument after `lake build`.

Libraries and Imports

A Lean library consists of a hierarchically organized collection of source files from which names can be imported, called *modules*. By default, a library has a single root file that matches its name. In this case, the root file for the library `Greeting` is `Greeting.lean`. The first line of `Main.lean`, which is `import Greeting`, makes the contents of `Greeting.lean` available in `Main.lean`.

Additional module files may be added to the library by creating a directory called `Greeting` and placing them inside. These names can be imported by replacing the directory separator with a dot. For instance, creating the file `Greeting/Smile.lean` with the contents:

```
def expression : String := "a big smile"
```

means that `Main.lean` can use the definition as follows:

```
import Greeting
import Greeting.Smile

def main : IO Unit :=
  IO.println s!"Hello, {hello}, with {expression}!"
```

The module name hierarchy is decoupled from the namespace hierarchy. In Lean, modules are units of code distribution, while namespaces are units of code organization. That is, names defined in the module `Greeting.Smile` are not automatically in a corresponding namespace `Greeting.Smile`. Modules may place names into any namespace they like, and the code that imports them may `open` the namespace or not. `import` is used to make the contents of a source file available, while `open` makes names from a namespace available in the current context without prefixes. In the Lakefile, the line `import Lake` makes the contents of the `Lake` module available, while the line `open Lake DSL` makes the contents of the `Lake` and `Lake.DSL` namespaces available without any prefixes. `Lake.DSL` is opened because opening `Lake` makes `Lake.DSL` available as just `DSL`, just like all other names in the `Lake` namespace. The `Lake` module places names into both the `Lake` and `Lake.DSL` namespaces.

Namespaces may also be opened *selectively*, making only some of their names available without explicit prefixes. This is done by writing the desired names in parentheses. For example, `Nat.toFloat` converts a natural number to a `Float`. It can be made available as `toFloat` using `open Nat (toFloat)`.

Worked Example: `cat`

The standard Unix utility `cat` takes a number of command-line options, followed by zero or more input files. If no files are provided, or if one of them is a dash (`-`), then it takes the standard input as the corresponding input instead of reading a file. The contents of the inputs are written, one after the other, to the standard output. If a specified input file does not exist, this is noted on standard error, but `cat` continues concatenating the remaining inputs. A non-zero exit code is returned if any of the input files do not exist.

This section describes a simplified version of `cat`, called `feline`. Unlike commonly-used versions of `cat`, `feline` has no command-line options for features such as numbering lines, indicating non-printing characters, or displaying help text. Furthermore, it cannot read more than once from a standard input that's associated with a terminal device.

To get the most benefit from this section, follow along yourself. It's OK to copy-paste the code examples, but it's even better to type them in by hand. This makes it easier to learn the mechanical process of typing in code, recovering from mistakes, and interpreting feedback from the compiler.

Getting started

The first step in implementing `feline` is to create a package and decide how to organize the code. In this case, because the program is so simple, all the code will be placed in `Main.lean`. The first step is to run `lake new feline`. Edit the Lakefile to remove the library, and delete the generated library code and the reference to it from `Main.lean`. Once this has been done, `lakefile.lean` should contain:

```
import Lake
open Lake DSL

package «feline» {
  -- add package configuration options here
}

@[default_target]
lean_exe «feline» {
  root := `Main
}
```

and `Main.lean` should contain something like:

```
def main : IO Unit :=
  IO.println s!"Hello, cats!"
```

Alternatively, running `lake new feline exe` instructs `lake` to use a template that does not include a library section, making it unnecessary to edit the file.

Ensure that the code can be built by running `lake build`.

Concatenating Streams

Now that the basic skeleton of the program has been built, it's time to actually enter the code. A proper implementation of `cat` can be used with infinite IO streams, such as `/dev/random`, which means that it can't read its input into memory before outputting it. Furthermore, it should not work one character at a time, as this leads to frustratingly slow performance. Instead, it's better to read contiguous blocks of data all at once, directing the data to the standard output one block at a time.

The first step is to decide how big of a block to read. For the sake of simplicity, this implementation uses a conservative 20 kilobyte block. `USize` is analogous to `size_t` in C—it's an unsigned integer type that is big enough to represent all valid array sizes.

```
def bufsize : USize := 20 * 1024
```

Streams

The main work of `feline` is done by `dump`, which reads input one block at a time, dumping the result to standard output, until the end of the input has been reached:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

The `dump` function is declared `partial`, because it calls itself recursively on input that is not immediately smaller than an argument. When a function is declared to be partial, Lean does not require a proof that it terminates. On the other hand, partial functions are also much less amenable to proofs of correctness, because allowing infinite loops in Lean's logic would make it unsound. However, there is no way to prove that `dump` terminates, because infinite input (such as from `/dev/random`) would mean that it does not, in fact, terminate. In cases like this, there is no alternative to declaring the function `partial`.

The type `IO.FS.Stream` represents a POSIX stream. Behind the scenes, it is represented as a structure that has one field for each POSIX stream operation. Each operation is represented as an IO action that provides the corresponding operation:

```
structure Stream where
  flush   : IO Unit
  read    : USize → IO ByteArray
  write   : ByteArray → IO Unit
  getLine : IO String
  putStr  : String → IO Unit
```

The Lean compiler contains `IO` actions (such as `IO.getStdout`, which is called in `dump`) to get streams that represent standard input, standard output, and standard error. These are `IO` actions rather than ordinary definitions because Lean allows these standard POSIX streams to be replaced in a process, which makes it easier to do things like capturing the output from a program into a string by writing a custom `IO.FS.Stream`.

The control flow in `dump` is essentially a `while` loop. When `dump` is called, if the stream has reached the end of the file, `pure ()` terminates the function by returning the constructor for `Unit`. If the stream has not yet reached the end of the file, one block is read, and its contents are written to `stdout`, after which `dump` calls itself directly. The recursive calls continue until `stream.read` returns an empty byte array, which indicates that the end of the file has been reached.

When an `if` expression occurs as a statement in a `do`, as in `dump`, each branch of the `if` is implicitly provided with a `do`. In other words, the sequence of steps following the `else` are treated as a sequence of `IO` actions to be executed, just as if they had a `do` at the beginning. Names introduced with `let` in the branches of the `if` are visible only in their own branches, and are not in scope outside of the `if`.

There is no danger of running out of stack space while calling `dump` because the recursive call happens as the very last step in the function, and its result is returned directly rather than being manipulated or computed with. This kind of recursion is called *tail recursion*, and it is described in more detail later in this book. Because the compiled code does not need to retain any state, the Lean compiler can compile the recursive call to a jump.

If `feline` only redirected standard input to standard output, then `dump` would be sufficient. However, it also needs to be able to open files that are provided as command-line arguments and emit their contents. When its argument is the name of a file that exists, `fileStream` returns a stream that reads the file's contents. When the argument is not a file, `fileStream` emits an error and returns `none`.

```
def fileStream (filename : System.FilePath) : IO (Option IO.FS.Stream) := do
  let fileExists ← filename.pathExists
  if not fileExists then
    let stderr ← IO.getStderr
    stderr.putStrLn s!"File not found: {filename}"
    pure none
  else
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
    pure (some (IO.FS.Stream.ofHandle handle))
```

Opening a file as a stream takes two steps. First, a file handle is created by opening the file in read mode. A Lean file handle tracks an underlying file descriptor. When there are no references to the file handle value, a finalizer closes the file descriptor. Second, the file handle is given the same interface as a POSIX stream using `IO.FS.Stream.ofHandle`, which fills each field of the `Stream` structure with the corresponding `IO` action that works on file handles.

Handling Input

The main loop of `feline` is another tail-recursive function, called `process`. In order to return a non-zero exit code if any of the inputs could not be read, `process` takes an argument `exitCode` that represents the current exit code for the whole program. Additionally, it takes a list of input files to be processed.

```
def process (exitCode : UInt32) (args : List String) : IO UInt32 := do
  match args with
  | [] => pure exitCode
  | "-" :: args =>
    let stdin ← IO.getStdin
    dump stdin
    process exitCode args
  | filename :: args =>
    let stream ← fileStream ⟨filename⟩
    match stream with
    | none =>
      process 1 args
    | some stream =>
      dump stream
      process exitCode args
```

Just as with `if`, each branch of a `match` that is used as a statement in a `do` is implicitly provided with its own `do`.

There are three possibilities. One is that no more files remain to be processed, in which case `process` returns the error code unchanged. Another is that the specified filename is `"-"`, in which case `process` dumps the contents of the standard input and then processes the remaining filenames. The final possibility is that an actual filename was specified. In this case, `fileStream` is used to attempt to open the file as a POSIX stream. Its argument is encased in `⟨ ... ⟩` because a `FilePath` is a single-field structure that contains a string. If the file could not be opened, it is skipped, and the recursive call to `process` sets the exit code to `1`. If it could, then it is dumped, and the recursive call to `process` leaves the exit code unchanged.

`process` does not need to be marked `partial` because it is structurally recursive. Each recursive call is provided with the tail of the input list, and all Lean lists are finite. Thus, `process` does not introduce any non-termination.

Main

The final step is to write the `main` action. Unlike prior examples, `main` in `feline` is a function. In Lean, `main` can have one of three types:

- `main : IO Unit` corresponds to programs that cannot read their command-line arguments and always indicate success with an exit code of `0`,
- `main : IO UInt32` corresponds to `int main(void)` in C, for programs without arguments that return exit codes, and
- `main : List String → IO UInt32` corresponds to `int main(int argc, char **argv)` in C, for programs that take arguments and signal success or failure.

If no arguments were provided, `feline` should read from standard input as if it were called with a single `"-"` argument. Otherwise, the arguments should be processed one after the other.

```
def main (args : List String) : IO UInt32 :=
  match args with
  | [] => process 0 ["-"]
  | _ =>  process 0 args
```

Meow!

To check whether `feline` works, the first step is to build it with `lake build`. First off, when called without arguments, it should emit what it receives from standard input. Check that

```
echo "It works!" | ./build/bin/feline
```

emits `It works!`.

Secondly, when called with files as arguments, it should print them. If the file `test1.txt` contains

```
It's time to find a warm spot
```

and `test2.txt` contains

```
and curl up!
```

then the command

```
./build/bin/feline test1.txt test2.txt
```

should emit

```
It's time to find a warm spot
and curl up!
```

Finally, the `-` argument should be handled appropriately.

```
echo "and purr" | ./build/bin/feline test1.txt - test2.txt
```

should yield

```
It's time to find a warm spot
and purr
and curl up!
```

Exercise

Extend `feline` with support for usage information. The extended version should accept a command-line argument `--help` that causes documentation about the available command-line options to be written to standard output.

Additional Conveniences

Nested Actions

Many of the functions in `feline` exhibit a repetitive pattern in which an `IO` action's result is given a name, and then used immediately and only once. For instance, in `dump`:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

the pattern occurs for `stdout`:

```
    let stdout ← IO.getStdout
    stdout.write buf
```

Similarly, `fileStream` contains the following snippet:

```
  let fileExists ← filename.pathExists
  if not fileExists then
```

When Lean is compiling a `do` block, expressions that consist of a left arrow immediately under parentheses are lifted to the nearest enclosing `do`, and their results are bound to a unique name. This unique name replaces the origin of the expression. This means that `dump` can also be written as follows:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    (← IO.getStdout).write buf
    dump stream
```

This version of `dump` avoids introducing names that are used only once, which can greatly simplify a program. `IO` actions that Lean lifts from a nested expression context are called *nested actions*.

`fileStream` can be simplified using the same technique:

```
def fileStream (filename : System.FilePath) : IO (Option IO.FS.Stream) := do
  if not (← filename.pathExists) then
    (← IO.getStderr).putStrLn s!"File not found: {filename}"
    pure none
  else
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
    pure (some (IO.FS.Stream.ofHandle handle))
```

In this case, the local name of `handle` could also have been eliminated using nested actions, but the resulting expression would have been long and complicated. Even though it's often good style to use nested actions, it can still sometimes be helpful to name intermediate results.

It is important to remember, however, that nested actions are only a shorter notation for `IO` actions that occur in a surrounding `do` block. The side effects that are involved in executing them still occur in the same order, and execution of side effects is not interspersed with the evaluation of expressions. For an example of where this might be confusing, consider the following helper definitions that return data after announcing to the world that they have been executed:

```
def getNumA : IO Nat := do
  (← IO.getStdout).putStrLn "A"
  pure 5

def getNumB : IO Nat := do
  (← IO.getStdout).putStrLn "B"
  pure 7
```

These definitions are intended to stand in for more complicated `IO` code that might validate user input, read a database, or open a file.

A program that prints `0` when number A is five, or number `B` otherwise, can be written as follows:

```
def test : IO Unit := do
  let a : Nat := if (← getNumA) == 5 then 0 else (← getNumB)
  (← IO.getStdout).putStrLn s!"The answer is {a}"
```

However, this program probably has more side effects (such as prompting for user input or reading a database) than was intended. The definition of `getNumA` makes it clear that it will always return `5`, and thus the program should not read number B. However, running the program results in the following output:

```
A
B
The answer is 0
```

`getNumB` was executed because `test` is equivalent to this definition:

```
def test : IO Unit := do
  let x ← getNumA
  let y ← getNumB
  let a : Nat := if x == 5 then 0 else y
  (← IO.getStdout).putStrLn s!"The answer is {a}"
```

This is due to the rule that nested actions are lifted to the *closest enclosing* `do` block. The branches of the `if` were not implicitly wrapped in `do` blocks because the `if` is not itself a statement in the `do` block—the statement is the `let` that defines `a`. Indeed, they could not be wrapped this way, because the type of the conditional expression is `Nat`, not `IO Nat`.

Flexible Layouts for `do`

In Lean, `do` expressions are whitespace-sensitive. Each `IO` action or local binding in the `do` is expected to start on its own line, and they should all have the same indentation. Almost all uses of `do` should be written this way. In some rare contexts, however, manual control over whitespace and indentation may be necessary, or it may be convenient to have multiple small actions on a single line. In these cases, newlines can be replaced with a semicolon and indentation can be replaced with curly braces.

For instance, all of the following programs are equivalent:

```
-- This version uses only whitespace-sensitive layout
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let name := (← stdin.getLine).trim
  stdout.putStrLn s!"Hello, {name}!"

-- This version is as explicit as possible
def main : IO Unit := do {
  let stdin ← IO.getStdin;
  let stdout ← IO.getStdout;

  stdout.putStrLn "How would you like to be addressed?";
  let name := (← stdin.getLine).trim;
  stdout.putStrLn s!"Hello, {name}!"
}

-- This version uses a semicolon to put two actions on the same line
def main : IO Unit := do
  let stdin ← IO.getStdin; let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let name := (← stdin.getLine).trim
  stdout.putStrLn s!"Hello, {name}!"
```

Idiomatic Lean code uses curly braces with `do` very rarely.

Running `IO` Actions With `#eval`

Lean's `#eval` command can be used to execute `IO` actions, rather than just evaluating them. Normally, adding a `#eval` command to a Lean file causes Lean to evaluate the provided expression, convert the resulting value to a string, and provide that string as a tooltip and in the info window. Rather than failing because `IO` actions can't be converted to strings, `#eval` executes them, carrying out their side effects. If the result of execution is the `Unit` value `()`, then no result string is shown, but if it is a type that can be converted to a string, then Lean displays the resulting value.

This means that, given the prior definitions of `countdown` and `runActions`,

```
#eval runActions (countdown 3)
```

displays

```
3
2
1
Blast off!
```

This is the output produced by running the `IO` action, rather than some opaque representation of the action itself. In other words, for `IO` actions, `#eval` both *evaluates* the provided expression and *executes* the resulting action value.

Quickly testing `IO` actions with `#eval` can be much more convenient that compiling and running whole programs. However, there are some limitations. For instance, reading from standard input simply returns empty input. Additionally, the `IO` action is re-executed whenever Lean needs to update the diagnostic information that it provides to users, and this can happen at unpredictable times. An action that reads and writes files, for instance, may do so at inconvenient times.

Summary

Evaluation vs Execution

Side effects are aspects of program execution that go beyond the evaluation of mathematical expressions, such as reading files, throwing exceptions, or triggering industrial machinery. While most languages allow side effects to occur during evaluation, Lean does not. Instead, Lean has a type called `IO` that represents *descriptions* of programs that use side effects. These descriptions are then executed by the language's run-time system, which invokes the Lean expression evaluator to carry out specific computations. Values of type `IO α` are called *`IO` actions*. The simplest is `pure`, which returns its argument and has no actual side effects.

`IO` actions can also be understood as functions that take the whole world as an argument and return a new world in which the side effect has occurred. Behind the scenes, the `IO` library ensures that the world is never duplicated, created, or destroyed. While this model of side effects cannot actually be implemented, as the whole universe is too big to fit in memory, the real world can be represented by a token that is passed around through the program.

An `IO` action `main` is executed when the program starts. `main` can have one of three types:

- `main : IO Unit` is used for simple programs that cannot read their command-line arguments and always return exit code `0`,
- `main : IO UInt32` is used for programs without arguments that may signal success or failure, and
- `main : List String → IO UInt32` is used for programs that take command-line arguments and signal success or failure.

`do` Notation

The Lean standard library provides a number of basic `IO` actions that represent effects such as reading from and writing to files and interacting with standard input and standard output. These base `IO` actions are composed into larger `IO` actions using `do` notation, which is a built-in domain-specific language for writing descriptions of programs with side effects. A `do` expression contains a sequence of *statements*, which may be:

- expressions that represent `IO` actions,

- ordinary local definitions with `let` and `:=`, where the defined name refers to the value of the provided expression, or
- local definitions with `let` and `←`, where the defined name refers to the result of executing the value of the provided expression.

`IO` actions that are written with `do` are executed one statement at a time.

Furthermore, `if` and `match` expressions that occur immediately under a `do` are implicitly considered to have their own `do` in each branch. Inside of a `do` expression, *nested actions* are expressions with a left arrow immediately under parentheses. The Lean compiler implicitly lifts them to the nearest enclosing `do`, which may be implicitly part of a branch of a `match` or `if` expression, and gives them a unique name. This unique name then replaces the origin site of the nested action.

Compiling and Running Programs

A Lean program that consists of a single file with a `main` definition can be run using `lean --run FILE`. While this can be a nice way to get started with a simple program, most programs will eventually graduate to a multiple-file project that should be compiled before running.

Lean projects are organized into *packages*, which are collections of libraries and executables together with information about dependencies and a build configuration. Packages are described using Lake, a Lean build tool. Use `lake new` to create a Lake package in a new directory, or `lake init` to create one in the current directory. Lake package configuration is another domain-specific language. Use `lake build` to build a project.

Partiality

One consequence of following the mathematical model of expression evaluation is that every expression must have a value. This rules out both incomplete pattern matches that fail to cover all constructors of a datatype and programs that can fall into an infinite loop. Lean ensures that all `match` expressions cover all cases, and that all recursive functions are either structurally recursive or have an explicit proof of termination.

However, some real programs require the possibility of looping infinitely, because they handle potentially-infinite data, such as POSIX streams. Lean provides an escape hatch: functions whose definition is marked `partial` are not required to terminate. This comes at a cost. Because types are a first-class part of the Lean language, functions can return types. Partial functions, however, are not evaluated during type checking, because an infinite loop in a function could cause the type checker to enter an infinite loop. Furthermore, mathematical proofs are unable to inspect the definitions of partial functions, which means that programs that use them are much less amenable to formal proof.

Interlude: Propositions, Proofs, and Indexing

Like many languages, Lean uses square brackets for indexing into arrays and lists. For instance, if `woodlandCritters` is defined as follows:

```
def woodlandCritters : List String :=
  ["hedgehog", "deer", "snail"]
```

then the individual components can be extracted:

```
def hedgehog := woodlandCritters[0]
def deer := woodlandCritters[1]
def snail := woodlandCritters[2]
```

However, attempting to extract the fourth element results in a compile-time error, rather than a run-time error:

```
def oops := woodlandCritters[3]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is perfomed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
⊢ 3 < List.length woodlandCritters
```

This error message is saying Lean tried to automatically mathematically prove that `3 < List.length woodlandCritters`, which would mean that the lookup was safe, but that it could not do so. Out-of-bounds errors are a common class of bugs, and Lean uses its dual nature as a programming language and a theorem prover to rule out as many as possible.

Understanding how this works requires an understanding of three key ideas: propositions, proofs, and tactics.

Propositions and Proofs

A *proposition* is a statement that can be true or false. All of the following are propositions:

- 1 + 1 = 2
- Addition is commutative
- There are infinitely many prime numbers

- 1 + 1 = 15
- Paris is the capital of France
- Buenos Aires is the capital of South Korea
- All birds can fly

On the other hand, nonsense statements are not propositions. None of the following are propositions:

- 1 + green = ice cream
- All capital cities are prime numbers
- At least one gorg is a fleep

Propositions come in two varieties: those that are purely mathematical, relying only on our definitions of concepts, and those that are facts about the world. Theorem provers like Lean are concerned with the former category, and have nothing to say about the flight capabilities of penguins or the legal status of cities.

A *proof* is a convincing argument that a proposition is true. For mathematical propositions, these arguments make use of the definitions of the concepts that are involved as well as the rules of logical argumentation. Most proofs are written for people to understand, and leave out many tedious details. Computer-aided theorem provers like Lean are designed to allow mathematicians to write proofs while omitting many details, and it is the software's responsibility to fill in the missing explicit steps. This decreases the likelihood of oversights or mistakes.

In Lean, a program's type describes the ways it can be interacted with. For instance, a program of type `Nat → List String` is a function that takes a `Nat` argument and produces a list of strings. In other words, each type specifies what counts as a program with that type.

In Lean, propositions are in fact types. They specify what counts as evidence that the statement is true. The proposition is proved by providing this evidence. On the other hand, if the proposition is false, then it will be impossible to construct this evidence.

For example, the proposition "1 + 1 = 2" can be written directly in Lean. The evidence for this proposition is the constructor `rfl`, which is short for *reflexivity*:

```
def onePlusOneIsTwo : 1 + 1 = 2 := rfl
```

On the other hand, `rfl` does not prove the false proposition "1 + 1 = 15":

```
def onePlusOneIsFifteen : 1 + 1 = 15 := rfl
```

```
type mismatch
  rfl
has type
  1 + 1 = 1 + 1 : Prop
but is expected to have type
  1 + 1 = 15 : Prop
```

This error message indicates that `rfl` can prove that two expressions are equal when both sides of the equality statement are already the same number. Because `1 + 1` evaluates directly to `2`, they are considered to be the same, which allows `onePlusOneIsTwo` to be accepted. Just as `Type` describes types such as `Nat`, `String`, and `List (Nat × String × (Int → Float))` that represent data structures and functions, `Prop` describes propositions.

When a proposition has been proven, it is called a *theorem*. In Lean, it is conventional to declare theorems with the `theorem` keyword instead of `def`. This helps readers see which declarations are intended to be read as mathematical proofs, and which are definitions. Generally speaking, with a proof, what matters is that there is evidence that a proposition is true, but it's not particularly important *which* evidence was provided. With definitions, on the other hand, it matters very much which particular value is selected—after all, a definition of addition that always returns `0` is clearly wrong.

The prior example could be rewritten as follows:

```
def OnePlusOneIsTwo : Prop := 1 + 1 = 2

theorem onePlusOneIsTwo : OnePlusOneIsTwo := rfl
```

Tactics

Proofs are normally written using *tactics*, rather than by providing evidence directly. Tactics are small programs that construct evidence for a proposition. These programs run in a *proof state* that tracks the statement that is to be proved (called the *goal*) along with the assumptions that are available to prove it. Running a tactic on a goal results in a new proof state that contains new goals. The proof is complete when all goals have been proven.

To write a proof with tactics, begin the definition with `by`. Writing `by` puts Lean into tactic mode until the end of the next indented block. While in tactic mode, Lean provides ongoing feedback about the current proof state. Written with tactics, `onePlusOneIsTwo` is still quite short:

```
theorem onePlusOneIsTwo : 1 + 1 = 2 := by
  simp
```

The `simp` tactic, short for "simplify", is the workhorse of Lean proofs. It rewrites the goal to as simple a form as possible, taking care of parts of the proof that are small enough. In particular, it proves simple equality statements. Behind the scenes, a detailed formal proof is constructed, but using `simp` hides this complexity.

Tactics are useful for a number of reasons:

1. Many proofs are complicated and tedious when written out down to the smallest detail, and tactics can automate these uninteresting parts.
2. Proofs written with tactics are easier to maintain over time, because flexible automation can paper over small changes to definitions.
3. Because a single tactic can prove many different theorems, Lean can use tactics behind the scenes to free users from writing proofs by hand. For instance, an array lookup requires a proof that the index is in bounds, and a tactic can typically construct that proof without the user needing to worry about it.

Behind the scenes, indexing notation uses a tactic to prove that the user's lookup operation is safe. This tactic is `simp`, configured to take certain arithmetic identities into account.

Connectives

The basic building blocks of logic, such as "and", "or", "true", "false", and "not", are called *logical connectives*. Each connective defines what counts as evidence of its truth. For example, to prove a statement "*A* and *B*", one must prove both *A* and *B*. This means that evidence for "*A* and *B*" is a pair that contains both evidence for *A* and evidence for *B*. Similarly, evidence for "*A* or *B*" consists of either evidence for *A* or evidence for *B*.

In particular, most of these connectives are defined like datatypes, and they have constructors. If `A` and `B` are propositions, then "`A` and `B`" (written `A ∧ B`) is a proposition. Evidence for `A ∧ B` consists of the constructor `And.intro`, which has the type `A → B → A ∧ B`. Replacing `A` and `B` with concrete propositions, it is possible to prove `1 + 1 = 2 ∧ "Str".append "ing" = "String"` with `And.intro rfl rfl`. Of course, `simp` is also powerful enough to find this proof:

```
theorem addAndAppend : 1 + 1 = 2 ∧ "Str".append "ing" = "String" := by simp
```

Similarly, "`A` or `B`" (written `A ∨ B`) has two constructors, because a proof of "`A` or `B`" requires only that one of the two underlying propositions be true. There are two constructors: `Or.inl`, with type `A → A ∨ B`, and `Or.inr`, with type `B → A ∨ B`.

Implication (if *A* then *B*) is represented using functions. In particular, a function that transforms evidence for *A* into evidence for *B* is itself evidence that *A* implies *B*. This is different from the usual description of implication, in which `A → B` is shorthand for `¬A ∨ B`, but the two formulations are equivalent.

Because evidence for an "and" is a constructor, it can be used with pattern matching. For instance, a proof that *A* and *B* implies *A* or *B* is a function that pulls the evidence of *A* (or of *B*) out of the evidence for *A* and *B*, and then uses this evidence to produce evidence of *A* or *B*:

```
theorem andImpliesOr : A ∧ B → A ∨ B :=
  fun andEvidence =>
    match andEvidence with
    | And.intro a b => Or.inl a
```

Connective	Lean Syntax	Evidence
True	`True`	`True.intro : True`
False	`False`	No evidence
A and *B*	`A ∧ B`	`And.intro : A → B → A ∧ B`
A or *B*	`A ∨ B`	Either `Or.inl : A → A ∨ B` or `Or.inr : B → A ∨ B`
A implies *B*	`A → B`	A function that transforms evidence of *A* into evidence of *B*
not *A*	`¬A`	A function that would transform evidence of *A* into evidence of `False`

The `simp` tactic can prove theorems that use these connectives. For example:

```
theorem onePlusOneAndLessThan : 1 + 1 = 2 ∨ 3 < 5 := by simp
theorem notTwoEqualFive : ¬(1 + 1 = 5) := by simp
theorem trueIsTrue : True := True.intro
theorem trueOrFalse : True ∨ False := by simp
theorem falseImpliesTrue : False → True := by simp
```

Evidence as Arguments

While `simp` does a great job proving propositions that involve equalities and inequalities of specific numbers, it is not very good at proving statements that involve variables. For instance, `simp` can prove that `4 < 15`, but it can't easily tell that because `x < 4`, it's also true that `x < 15`. Because index notation uses `simp` behind the scenes to prove that array access is safe, it can require a bit of hand-holding.

One of the easiest ways to make indexing notation work well is to have the function that performs a lookup into a data structure take the required evidence of safety as an argument. For instance, a function that returns the third entry in a list is not generally safe because lists might contain zero, one, or two entries:

```
def third (xs : List α) : α := xs[2]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is perfomed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
α : Type ?u.3900
xs : List α
⊢ 2 < List.length xs
```

However, the obligation to show that the list has at least three entries can be imposed on the caller by adding an argument that consists of evidence that the indexing operation is safe:

```
def third (xs : List α) (ok : xs.length > 2) : α := xs[2]
```

In this example, `xs.length > 2` is not a program that checks *whether* `xs` has more than 2 entries. It is a proposition that could be true or false, and the argument `ok` must be evidence that it is true.

When the function is called on a concrete list, its length is known. In these cases, `by simp` can construct the evidence automatically:

```
#eval third woodlandCritters (by simp)
```

```
"snail"
```

Indexing Without Evidence

In cases where it's not practical to prove that an indexing operation is in bounds, there are other alternatives. Adding a question mark results in an `Option`, where the result is `some` if the index is in bounds, and `none` otherwise. For example:

```
def thirdOption (xs : List α) : Option α := xs[2]?

#eval thirdOption woodlandCritters
```

```
some "snail"
```

```
#eval thirdOption ["only", "two"]
```

```
none
```

There is also a version that crashes the program when the index is out of bounds, rather than returning an `Option`:

```
#eval woodlandCritters[1]!
```

```
"deer"
```

Be careful! Because code that is run with `#eval` runs in the context of the Lean compiler, selecting the wrong index can crash your IDE.

Messages You May Meet

In addition to the error that occurs when Lean is unable to find compile-time evidence that an indexing operation is safe, polymorphic functions that use unsafe indexing may produce the following message:

```
def unsafeThird (xs : List α) : α := xs[2]!
```

```
failed to synthesize instance
  Inhabited α
```

This is due to a technical restriction that is part of keeping Lean usable as both a logic for proving theorems and a programming language. In particular, only programs whose types contain at least one value are allowed to crash. This is because a proposition in Lean is a kind of type that classifies evidence of its truth. False propositions have no such evidence. If a program with an empty type could crash, then that crashing program could be used as a kind of fake evidence for a false proposition.

Internally, Lean contains a table of types that are known to have at least one value. This error is saying that some arbitrary type `α` is not necessarily in that table. The next chapter describes how to add to this table, and how to successfully write functions like `unsafeThird`.

Adding whitespace between a list and the brackets used for lookup can cause another message:

```
#eval woodlandCritters [1]
```

```
function expected at
  woodlandCritters
term has type
  List String
```

Adding a space causes Lean to treat the expression as a function application, and the index as a list that contains a single number. This error message results from having Lean attempt to treat `woodlandCritters` as a function.

Exercises

- Prove the following theorems using `rfl`: `2 + 3 = 5`, `15 - 8 = 7`, `"Hello, ".append "world" = "Hello, world"`. What happens if `rfl` is used to prove `5 < 18`? Why?
- Prove the following theorems using `by simp`: `2 + 3 = 5`, `15 - 8 = 7`, `"Hello, ".append "world" = "Hello, world"`, `5 < 18`.
- Write a function that looks up the fifth entry in a list. Pass the evidence that this lookup is safe as an argument to the function.