Metaprogramming in Lean 4

Arthur Paulino, Damiano Testa, Edward Ayers, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, Siddharth Bhat

Formatted for PDF by Julian Berman using Pascal Wagler's Template

Contents

Introduction

What's the goal of this book?

This book aims to build up enough knowledge about metaprogramming in Lean 4 so you can be comfortable enough to:

- Start building your own meta helpers
- Read and discuss metaprogramming API's like the ones in Lean 4 core and Mathlib4

We by no means intend to provide an exhaustive exploration/explanation of the entire Lean 4 metaprogramming API. We also don't cover the topic of monadic programming in Lean 4. However, we hope that the examples provided will be simple enough for the reader to follow and comprehend without a super deep understanding of monadic programming.

Book structure

The book is organized in a way to build up enough content for the chapters that cover DSLs and tactics. Backtracking the pre-requisites for each chapter, the dependency structure is as follows:

- "Tactics" builds on top of "Macros" and "Elaboration"
- "DSLs" builds on top of "Elaboration"
- "Macros" builds on top of "`Syntax`"
- "Elaboration" builds on top of "`Syntax`" and "`MetaM`"
- "`MetaM`" builds on top of "Expressions"

After the chapter on tactics, you find a cheat-sheet containing a wrap-up of key concepts and functions. And after that, There are some chapters with extra content, showing other applications of metaprogramming in Lean 4.

The rest of this chapter is a gentle introduction for what metaprogramming is, offering some small examples to serve as appetizers for what the book shall cover.

What does it mean to be in meta?

When we write code in most programming languages such as Python, C, Java or Scala, we usually have to stick to a pre-defined syntax otherwise the compiler or the interpreter won't be able to figure out what we're trying to say. In Lean, that would be defining an inductive type, implementing a function, proving a theorem etc. The compiler, then, has to parse the code, build an abstract syntax tree and elaborate its syntax nodes into terms that can be processed by the language kernel. We say that such activities performed by the compiler are done in the **meta-level**, which will be studied throughout the book. And we also say that the common usage of the language syntax is done in the **object-level**.

In most systems, the meta-level activities are done in a different language to the one that we use to write code. In Isabelle, the meta-level language is ML and Scala. In Coq, it's OCaml. In Agda it's Haskell. In Lean 4, the meta code is mostly written in Lean itself, with a few components written in C++.

One cool thing about Lean, though, is that it allows us to define custom syntax nodes and to implement our own meta-level routines to elaborate those in the very same development environment that we use to perform object-level activities. So for example, one can write their own notation to instantiate a term of a certain type and use it right away, on the same file! This concept is generally called **reflection**. We can say that, in Lean, the meta-level is *reflected* to the object-level.

Since the objects defined in the meta-level are not the ones we're most interested in proving theorems about, it can sometimes be overly tedious to prove that they are type correct. For example, we don't care about proving that a recursive function to traverse an expression is well founded. Thus, we can use the `partial` keyword if we're convinced that our function terminates. In the worst case scenario, our function gets stuck in a loop but the kernel is not reached/affected.

Let's see some example use cases of metaprogramming in Lean.

Metaprogramming examples

The following examples are meant for mere illustration. Don't worry if you don't understand the details for now.

Introducing notation (defining new syntax)

Often one wants to introduce new notation, for example one more suitable for (a branch of) mathematics. For instance, in mathematics one would write the function adding 2 to a natural number as `x : Nat ↦ x + 2` or simply `x ↦ x + 2` if the domain can be inferred to be the natural numbers. The corresponding lean definitions `fun x : Nat => x + 2` and `fun x => x + 2` use `=>` which in mathematics means *implication*, so may be confusing to some.

We can introduce notation using a `macro` which transforms our syntax to lean's own syntax (or syntax we previously defined). Here we introduce the `↦` notation for functions.

```
import Lean

macro x:ident ":" t:term " ↦ " y:term : term => do
  `(fun $x : $t => $y)

#eval (x : Nat ↦ x + 2) 2 -- 4

macro x:ident " ↦ " y:term : term => do
  `(fun $x  => $y)

#eval (x ↦  x + 2) 2 -- 4
```

Building a command

Suppose we want to build a helper command `#assertType` which tells whether a given term is of a certain type. The usage will be:

```
#assertType <term> : <type>
```

Let's see the code:

```
elab "#assertType " termStx:term " : " typeStx:term : command =>
  open Lean Lean.Elab Command Term in
  liftTermElabM
    try
```

```
      let tp ← elabType typeStx
      discard $ elabTermEnsuringType termStx tp
      synthesizeSyntheticMVarsNoPostponing
      logInfo "success"
    catch | _ => throwError "failure"

#assertType 5  : Nat -- success
#assertType [] : Nat -- failure
```

We started by using `elab` to define a `command` syntax, which, when parsed by the compiler, will trigger the incoming computation.

At this point, the code should be running in the `CommandElabM` monad. We then use `liftTermElabM` to access the `TermElabM` monad, which allows us to use `elabType` and `elabTermEnsuringType` in order to build expressions out of the syntax nodes `typeStx` and `termStx`.

First we elaborate the expected type `tp : Expr` and then we use it to elaborate the term expression, which should have the type `tp` otherwise an error will be thrown. The term expression itself doesn't matter to us here, as we're calling `elabTermEnsuringType` as a sanity check.

We also add `synthesizeSyntheticMVarsNoPostponing`, which forces Lean to elaborate metavariables right away. Without that line, `#assertType 5  : ?_` would result in `success`.

If no error is thrown until now then the elaboration succeeded and we can use `logInfo` to output "success". If, instead, some error is caught, then we use `throwError` with the appropriate message.

Building a DSL and a syntax for it

Let's parse a classic grammar, the grammar of arithmetic expressions with addition, multiplication, naturals, and variables. We'll define an AST (Abstract Syntax Tree) to encode the data of our expressions, and use operators + and * to denote building an arithmetic AST. Here's the AST that we will be parsing:

```
inductive Arith : Type where
  | add : Arith → Arith → Arith -- e + f
  | mul : Arith → Arith → Arith -- e * f
  | nat : Nat → Arith           -- constant
  | var : String → Arith        -- variable
```

Now we declare a syntax category to describe the grammar that we will be parsing. Notice that we control the precedence of + and * by giving a lower precedence weight to the + syntax than to the * syntax indicating that multiplication binds tighter than addition (the higher the number, the tighter the binding). This allows us to declare *precedence* when defining new syntax.

```
declare_syntax_cat arith
syntax num                        : arith -- nat for Arith.nat
syntax str                        : arith -- strings for Arith.var
syntax:50 arith:50 " + " arith:51 : arith -- Arith.add
syntax:60 arith:60 " * " arith:61 : arith -- Arith.mul
syntax " ( " arith " ) "          : arith -- bracketed expressions

-- Auxiliary notation for translating `arith` into `term`
syntax " ⟪ " arith " ⟫ " : term

-- Our macro rules perform the "obvious" translation:
macro_rules
  | `(⟪ $s:str ⟫)              => `(Arith.var $s)
  | `(⟪ $num:num ⟫)            => `(Arith.nat $num)
  | `(⟪ $x:arith + $y:arith ⟫) => `(Arith.add ⟪ $x ⟫ ⟪ $y ⟫)
  | `(⟪ $x:arith * $y:arith ⟫) => `(Arith.mul ⟪ $x ⟫ ⟪ $y ⟫)
  | `(⟪ ( $x ) ⟫)              => `( ⟪ $x ⟫ )

#check ⟪ "x" * "y" ⟫
-- Arith.mul (Arith.var "x") (Arith.var "y") : Arith

#check ⟪ "x" + "y" ⟫
-- Arith.add (Arith.var "x") (Arith.var "y") : Arith

#check ⟪ "x" + 20 ⟫
-- Arith.add (Arith.var "x") (Arith.nat 20) : Arith

#check ⟪ "x" + "y" * "z" ⟫ -- precedence
-- Arith.add (Arith.var "x") (Arith.mul (Arith.var "y") (Arith.var "z")) : Arith

#check ⟪ "x" * "y" + "z" ⟫ -- precedence
-- Arith.add (Arith.mul (Arith.var "x") (Arith.var "y")) (Arith.var "z") : Arith

#check ⟪ ("x" + "y") * "z" ⟫ -- brackets
-- Arith.mul (Arith.add (Arith.symbol "x") (Arith.symbol "y")) (Arith.symbol "z")
```

Writing our own tactic

Let's create a tactic that adds a new hypothesis to the context with a given name and postpones the need for its proof to the very end. It's similar to the `suffices` tactic from Lean 3, except that we want to make sure that the new goal goes to the bottom of the goal list.

It's going to be called `suppose` and is used like this:

```
suppose <name> : <type>
```

So let's see the code:

```
open Lean Meta Elab Tactic Term in
elab "suppose " n:ident " : " t:term : tactic => do
  let n : Name := n.getId
  let mvarId ← getMainGoal
  mvarId.withContext do
    let t ← elabType t
    let p ← mkFreshExprMVar t MetavarKind.syntheticOpaque n
    let (_, mvarIdNew) ← intro1P $ ← mvarId.assert n t p
    replaceMainGoal [p.mvarId!, mvarIdNew]
  evalTactic $ ← `(tactic|rotate_left)

example : 0 + a = a := by
  suppose add_comm : 0 + a = a + 0
  rw [add_comm]; rfl     -- closes the initial main goal
  rw [Nat.zero_add]; rfl -- proves `add_comm`
```

We start by storing the main goal in `mvarId` and using it as a parameter of `withMVarContext` to make sure that our elaborations will work with types that depend on other variables in the context.

This time we're using `mkFreshExprMVar` to create a metavariable expression for the proof of `t`, which we can introduce to the context using `intro1P` and `assert`.

To require the proof of the new hypothesis as a goal, we call `replaceMainGoal` passing a list with `p.mvarId!` in the head. And then we can use the `rotate_left` tactic to move the recently added top goal to the bottom.

Printing Messages

In the `#assertType` example, we used `logInfo` to make our command print something. If, instead, we just want to perform a quick debug, we can use `dbg_trace`.

They behave a bit differently though, as we can see below:

```
elab "traces" : tactic => do
  let array := List.replicate 2 (List.range 3)
  Lean.logInfo m!"logInfo: {array}"
  dbg_trace f!"dbg_trace: {array}"

example : True := by -- `example` is underlined in blue, outputting:
                     -- dbg_trace: [[0, 1, 2], [0, 1, 2]]
  traces -- now `traces` is underlined in blue, outputting
         -- logInfo: [[0, 1, 2], [0, 1, 2]]
  trivial
```

Expressions

Expressions (terms of type `Expr`) carry the data used to communicate with the Lean kernel for core tasks such as type inference and definitional equality checks.

In Lean, terms and types are represented by expressions. For instance, let's consider `1` of type `Nat`. The type `Nat` is represented as a constant with the name "Nat". And then `1` is an application of the function `Nat.succ` to the term `Nat.zero`, so all this is represented as an application, given a constant named "Nat.succ" and a constant named "Nat.zero".

That example gives us an idea of what we're aiming at: we use expressions to represent all Lean terms at the meta level. Let's check the precise definition of `Expr`.

```
import Lean

namespace Playground

inductive Expr where
  | bvar    : Nat → Expr                              -- bound variables
  | fvar    : FVarId → Expr                           -- free variables
  | mvar    : MVarId → Expr                           -- meta variables
  | sort    : Level → Expr                            -- Sort
  | const   : Name → List Level → Expr                -- constants
  | app     : Expr → Expr → Expr                      -- application
  | lam     : Name → Expr → Expr → BinderInfo → Expr  -- lambda abstraction
  | forallE : Name → Expr → Expr → BinderInfo → Expr  -- (dependent) arrow
  | letE    : Name → Expr → Expr → Expr → Bool → Expr -- let expressions
  -- less essential constructors:
  | lit     : Literal → Expr                          -- literals
  | mdata   : MData → Expr → Expr                     -- metadata
  | proj    : Name → Nat → Expr → Expr                -- projection

end Playground
```

What is each of these constructors doing?

- `bvar` is a **bound variable**. For example, the `x` in `fun x => x + 2` or `∑ x, x²`.

This is any occurrence of a variable in an expression where there is a binder above it. Why is the argument a `Nat`? This is called a de Bruijn index and will be explained later. You can figure out the type of a bound variable by looking at its binder, since the binder always has the type information for it.

- `fvar` is a **free variable**. These are variables which are not bound by a binder. An example is `x` in `x + 2`. Note that you can't just look at a free variable `x` and tell what its type is, there needs to be a context which contains a declaration for `x` and its type. A free variable has an ID that tells you where to look for it in a `LocalContext`. In Lean 3, free variables were called "local constants" or "locals".
- `mvar` is a **metavariable**. There will be much more on these later, but you can think of it as a placeholder or a 'hole' in an expression that needs to be filled at a later point.
- `sort` is used for `Type u`, `Prop` etc.
- `const` is a constant that has been defined earlier in the Lean document.
- `app` is a function application. Multiple arguments are done using *partial application*: `f x y ↝ app (app f x) y`.
- `lam n t b` is a lambda expression (`fun ($n : $t) => $b`). The `b` argument is called the **body**. Note that you have to give the type of the variable you are binding.
- `forallE n t b` is a dependent arrow expression (`($n : $t) → $b`). This is also sometimes called a Π-type or Π-expression and is often written `∀ $n : $t, $b`. Note that the non-dependent arrow `α → β` is a special case of `(a : α) → β` where `β` doesn't depend on `a`. The `E` on the end of `forallE` is to distinguish it from the `forall` keyword.
- `letE n t v b` is a **let binder** (`let ($n : $t) := $v in $b`).
- `lit` is a **literal**, this is a number or string literal like `4` or `"hello world"`. Literals help with performance: we don't want to represent the expression `(10000 : Nat)` as `Nat.succ $ ... $ Nat.succ Nat.zero`.
- `mdata` is just a way of storing extra information on expressions that might be useful, without changing the nature of the expression.
- `proj` is for projection. Suppose you have a structure such as `p : α × β`, rather than storing the projection `π₁ p` as `app π₁ p`, it is expressed as `proj Prod 0 p`. This is for efficiency reasons ([todo] find link to docstring explaining this).

De Bruijn Indexes

Consider the following lambda expression `(λ f x => f x x) (λ x y => x + y) 5`, we have to be very careful when we reduce this, because we get a clash in the variable `x`.

To avoid variable name-clash carnage, `Exprs` use a nifty trick called **de Bruijn indexes**. In de Bruijn indexing, each variable bound by a `lam` or a `forallE` is converted into a number `#n`. The number says how many binders up the `Expr` tree we should look to find the binder which binds this variable. So our above example would become (putting wildcards _ in the type arguments for now for brevity): `app (app (lam `f _ (lam `x _ (app (app #1 #0) #0))) (lam `x _ (lam `y _ (app (app plus #1) #0)))) five` Now we don't need to rename variables when we perform β-reduction. We also really easily check if two `Exprs` containing bound expressions are equal. This is why the signature of the `bvar` case is `Nat → Expr` and not `Name → Expr`.

If a de Bruijn index is too large for the number of binders preceding it, we say it is a **loose `bvar`**; otherwise we say it is **bound**. For example, in the expression `lam `x _ (app #0 #1)` the `bvar #0` is bound by the preceding binder and `#1` is loose. The fact that Lean calls all de Bruijn indexes `bvars` ("bound variables") points to an important invariant: outside of some very low-level functions, Lean expects that expressions do not contain any loose `bvars`. Instead, whenever we would be tempted to introduce a loose `bvar`, we immediately convert it into an `fvar` ("free variable"). Precisely how that works is discussed in the next chapter.

If there are no loose `bvars` in an expression, we say that the expression is **closed**. The process of replacing all instances of a loose `bvar` with an `Expr` is called **instantiation**. Going the other way is called **abstraction**.

If you are familiar with the standard terminology around variables, Lean's terminology may be confusing, so here's a map: Lean's "bvars" are usually called just "variables"; Lean's "loose" is usually called "free"; and Lean's "fvars" might be called "local hypotheses".

Universe Levels

Some expressions involve universe levels, represented by the `Lean.Level` type. A universe level is a natural number, a universe parameter (introduced with a

`universe` declaration), a universe metavariable or the maximum of two universes. They are relevant for two kinds of expressions.

First, sorts are represented by `Expr.sort u`, where `u` is a `Level`. `Prop` is `sort Level.zero`; `Type` is `sort (Level.succ Level.zero)`.

Second, universe-polymorphic constants have universe arguments. A universe-polymorphic constant is one whose type contains universe parameters. For example, the `List.map` function is universe-polymorphic, as the `pp.universes` pretty-printing option shows:

```
set_option pp.universes true in
#check @List.map
```

The `.{u_1,u_2}` suffix after `List.map` means that `List.map` has two universe arguments, `u_1` and `u_2`. The `.{u_1}` suffix after `List` (which is itself a universe-polymorphic constant) means that `List` is applied to the universe argument `u_1`, and similar for `.{u_2}`.

In fact, whenever you use a universe-polymorphic constant, you must apply it to the correct universe arguments. This application is represented by the `List Level` argument of `Expr.const`. When we write regular Lean code, Lean infers the universes automatically, so we do not need think about them much. But when we construct `Expr`s, we must be careful to apply each universe-polymorphic constant to the right universe arguments.

Constructing Expressions

The simplest expressions we can construct are constants. We use the `const` constructor and give it a name and a list of universe levels. Most of our examples only involve non-universe-polymorphic constants, in which case the list is empty.

We also show a second form where we write the name with double backticks. This checks that the name in fact refers to a defined constant, which is useful to avoid typos.

```
open Lean

def z' := Expr.const `Nat.zero []
#eval z' -- Lean.Expr.const `Nat.zero []

def z := Expr.const ``Nat.zero []
#eval z -- Lean.Expr.const `Nat.zero []
```

The double-backtick variant also resolves the given name, making it fully-qualified. To illustrate this mechanism, here are two further examples. The first expression, `z₁`, is unsafe: if we use it in a context where the `Nat` namespace is not open, Lean will complain that there is no constant called `zero` in the environment. In contrast, the second expression, `z₂`, contains the fully-qualified name `Nat.zero` and does not have this problem.

```
open Nat

def z₁ := Expr.const `zero []
#eval z₁ -- Lean.Expr.const `zero []

def z₂ := Expr.const ``zero []
#eval z₂ -- Lean.Expr.const `Nat.zero []
```

The next class of expressions we consider are function applications. These can be built using the `app` constructor, with the first argument being an expression for the function and the second being an expression for the argument.

Here are two examples. The first is simply a constant applied to another. The second is a recursive definition giving an expression as a function of a natural number.

```
def one := Expr.app (.const ``Nat.succ []) z
#eval one
-- Lean.Expr.app (Lean.Expr.const `Nat.succ []) (Lean.Expr.const `Nat.zero [])

def natExpr: Nat → Expr
| 0     => z
| n + 1 => .app (.const ``Nat.succ []) (natExpr n)
```

Next we use the variant `mkAppN` which allows application with multiple arguments.

```
def sumExpr : Nat → Nat → Expr
| n, m => mkAppN (.const ``Nat.add []) #[natExpr n, natExpr m]
```

As you may have noticed, we didn't show `#eval` outputs for the two last functions. That's because the resulting expressions can grow so large that it's hard to make sense of them.

We next use the constructor `lam` to construct a simple function which takes any natural number `x` and returns `Nat.zero`. The argument `BinderInfo.default` says that `x` is an explicit argument (rather than an implicit or typeclass argument).

```
def constZero : Expr :=
  .lam `x (.const ``Nat []) (.const ``Nat.zero []) BinderInfo.default
```

```
#eval constZero
-- Lean.Expr.lam `x (Lean.Expr.const `Nat []) (Lean.Expr.const `Nat.zero [])
--   (Lean.BinderInfo.default)
```

As a more elaborate example which also involves universe levels, here is the `Expr` that represents `List.map (λ x => Nat.add x 1) []` (broken up into several definitions to make it somewhat readable):

```
def nat : Expr := .const ``Nat []

def addOne : Expr :=
  .lam `x nat
    (mkAppN (.const ``Nat.add []) #[.bvar 0, mkNatLit 1])
    BinderInfo.default

def mapAddOneNil : Expr :=
  mkAppN (.const ``List.map [levelOne, levelOne])
    #[nat, nat, addOne, .app (.const ``List.nil [levelOne]) nat]
```

With a little trick (more about which in the Elaboration chapter), we can turn our `Expr` into a Lean term, which allows us to inspect it more easily.

```
elab "mapAddOneNil" : term => return mapAddOneNil

#check mapAddOneNil
-- List.map (fun x => Nat.add x 1) [] : List Nat

set_option pp.universes true in
set_option pp.explicit true in
#check mapAddOneNil
-- @List.map.{1, 1} Nat Nat (fun x => Nat.add x 1) (@List.nil.{1} Nat) : List.{1} Nat

#reduce mapAddOneNil
-- []
```

In the next chapter we explore the `MetaM` monad, which, among many other things, allows us to more conveniently construct and destruct larger expressions.

MetaM

The Lean 4 metaprogramming API is organised around a small zoo of monads. The four main ones are:

- `CoreM` gives access to the *environment*, i.e. the set of things that have been declared or imported at the current point in the program.
- `MetaM` gives access to the *metavariable context*, i.e. the set of metavariables that are currently declared and the values assigned to them (if any).
- `TermElabM` gives access to various information used during elaboration.
- `TacticM` gives access to the list of current goals.

These monads extend each other, so a `MetaM` operation also has access to the environment and a `TermElabM` computation can use metavariables. There are also other monads which do not neatly fit into this hierarchy, e.g. `CommandElabM` extends `MetaM` but neither extends nor is extended by `TermElabM`.

This chapter demonstrates a number of useful operations in the `MetaM` monad. `MetaM` is of particular importance because it allows us to give meaning to every expression: the environment (from `CoreM`) gives meaning to constants like `Nat.zero` or `List.map` and the metavariable context gives meaning to both metavariables and local hypotheses.

```
import Lean

open Lean Lean.Expr Lean.Meta
```

Metavariables

Overview

The 'Meta' in `MetaM` refers to metavariables, so we should talk about these first. Lean users do not usually interact much with metavariables – at least not consciously –

but they are used all over the place in metaprograms. There are two ways to view them: as holes in an expression or as goals.

Take the goal perspective first. When we prove things in Lean, we always operate on goals, such as

```
n m : Nat
⊢ n + m = m + n
```

These goals are internally represented by metavariables. Accordingly, each metavariable has a *local context* containing hypotheses (here `[n : Nat, m : Nat]`) and a *target type* (here `n + m = m + n`). Metavariables also have a unique name, say `m`, and we usually render them as `?m`.

To close a goal, we must give an expression `e` of the target type. The expression may contain fvars from the metavariable's local context, but no others. Internally, closing a goal in this way corresponds to *assigning* the metavariable; we write `?m := e` for this assignment.

The second, complementary view of metavariables is that they represent holes in an expression. For instance, an application of `Eq.trans` may generate two goals which look like this:

```
n m : Nat
⊢ n = ?x

n m : Nat
⊢ ?x = m
```

Here `?x` is another metavariable – a hole in the target types of both goals, to be filled in later during the proof. The type of `?x` is `Nat` and its local context is `[n : Nat, m : Nat]`. Now, if we solve the first goal by reflexivity, then `?x` must be `n`, so we assign `?x := n`. Crucially, this also affects the second goal: it is "updated" (not really, as we will see) to have target `n = m`. The metavariable `?x` represents the same expression everywhere it occurs.

Tactic Communication via Metavariables

Tactics use metavariables to communicate the current goals. To see how, consider this simple (and slightly artificial) proof:

```
example {α} (a : α) (f : α → α) (h : ∀ a, f a = a) : f (f a) = a := by
  apply Eq.trans
```

```
  apply h
  apply h
```

After we enter tactic mode, our ultimate goal is to generate an expression of type `f (f a) = a` which may involve the hypotheses `α`, `a`, `f` and `h`. So Lean generates a metavariable `?m1` with target `f (f a) = a` and a local context containing these hypotheses. This metavariable is passed to the first `apply` tactic as the current goal.

The `apply` tactic then tries to apply `Eq.trans` and succeeds, generating three new metavariables:

```
...
⊢ f (f a) = ?b

...
⊢ ?b = a

...
⊢ α
```

Call these metavariables `?m2`, `?m3` and `?b`. The last one, `?b`, stands for the intermediate element of the transitivity proof and occurs in `?m2` and `?m3`. The local contexts of all metavariables in this proof are the same, so we omit them.

Having created these metavariables, `apply` assigns

```
?m1 := @Eq.trans α (f (f a)) ?b a ?m2 ?m3
```

and reports that `?m2`, `?m3` and `?b` are now the current goals.

At this point the second `apply` tactic takes over. It receives `?m2` as the current goal and applies `h` to it. This succeeds and the tactic assigns `?m2 := h (f a)`. This assignment implies that `?b` must be `f a`, so the tactic also assigns `?b := f a`. Assigned metavariables are not considered open goals, so the only goal that remains is `?m3`.

Now the third `apply` comes in. Since `?b` has been assigned, the target of `?m3` is now `f (f a) = a`. Again, the application of `h` succeeds and the tactic assigns `?m3 := h a`.

At this point, all metavariables are assigned as follows:

```
?m1 := @Eq.trans α (f (f a)) ?b a ?m2 ?m3
?m2 := h (f a)
?m3 := h a
?b  := f a
```

Exiting the by block, Lean constructs the final proof term by taking the assignment of ?m1 and replacing each metavariable with its assignment. This yields

```
@Eq.trans α (f (f a)) (f a) a (h (f a)) (h a)
```

The example also shows how the two views of metavariables – as holes in an expression or as goals – are related: the goals we get are holes in the final proof term.

Basic Operations

Let us make these concepts concrete. When we operate in the MetaM monad, we have read-write access to a MetavarContext structure containing information about the currently declared metavariables. Each metavariable is identified by an MVarId (a unique Name). To create a new metavariable, we use Lean.Meta.mkFreshExprMVar with type

```
mkFreshExprMVar (type? : Option Expr) (kind := MetavarKind.natural)
    (userName := Name.anonymous) : MetaM Expr
```

Its arguments are:

- type?: the target type of the new metavariable. If none, the target type is Sort ?u, where ?u is a universe level metavariable. (This is a special class of metavariables for universe levels, distinct from the expression metavariables which we have been calling simply “metavariables”.)
- kind: the metavariable kind. See the Metavariable Kinds section (but the default is usually correct).
- userName: the new metavariable’s user-facing name. This is what gets printed when the metavariable appears in a goal. Unlike the MVarId, this name does not need to be unique.

The returned Expr is always a metavariable. We can use Lean.Expr.mvarId! to extract the MVarId, which is guaranteed to be unique. (Arguably mkFreshExprMVar should just return the MVarId.)

The local context of the new metavariable is inherited from the current local context, more about which in the next section. If you want to give a different local context, use Lean.Meta.mkFreshExprMVarAt.

Metavariables are initially unassigned. To assign them, use Lean.MVarId.assign with type

```
assign (mvarId : MVarId) (val : Expr) : MetaM Unit
```

This updates the `MetavarContext` with the assignment `?mvarId := val`. You must make sure that `mvarId` is not assigned yet (or that the old assignment is definitionally equal to the new assignment). You must also make sure that the assigned value, `val`, has the right type. This means (a) that `val` must have the target type of `mvarId` and (b) that `val` must only contain fvars from the local context of `mvarId`.

If you `#check Lean.MVarId.assign`, you will see that its real type is more general than the one we showed above: it works in any monad that has access to a `MetavarContext`. But `MetaM` is by far the most important such monad, so in this chapter, we specialise the types of `assign` and similar functions.

To get information about a declared metavariable, use `Lean.MVarId.getDecl`. Given an `MVarId`, this returns a `MetavarDecl` structure. (If no metavariable with the given `MVarId` is declared, the function throws an exception.) The `MetavarDecl` contains information about the metavariable, e.g. its type, local context and user-facing name. This function has some convenient variants, such as `Lean.MVarId.getType`.

To get the current assignment of a metavariable (if any), use `Lean.getExprMVarAssignment?`. To check whether a metavariable is assigned, use `Lean.MVarId.isAssigned`. However, these functions are relatively rarely used in tactic code because we usually prefer a more powerful operation: `Lean.Meta.instantiateMVars` with type

```
instantiateMVars : Expr → MetaM Expr
```

Given an expression `e`, `instantiateMVars` replaces any assigned metavariable `?m` in `e` with its assigned value. Unassigned metavariables remain as they are.

This operation should be used liberally. When we assign a metavariable, existing expressions containing this metavariable are not immediately updated. This is a problem when, for example, we match on an expression to check whether it is an equation. Without `instantiateMVars`, we might miss the fact that the expression `?m`, where `?m` happens to be assigned to `0 = n`, represents an equation. In other words, `instantiateMVars` brings our expressions up to date with the current metavariable state.

Instantiating metavariables requires a full traversal of the input expression, so it can be somewhat expensive. But if the input expression does not contain any metavariables, `instantiateMVars` is essentially free. Since this is the common case, liberal use of `instantiateMVars` is fine in most situations.

Before we go on, here is a synthetic example demonstrating how the basic metavariable operations are used. More natural examples appear in the following sections.

```
#eval show MetaM Unit from do
  -- Create two fresh metavariables of type `Nat`.
  let mvar1 ← mkFreshExprMVar (Expr.const ``Nat []) (userName := `mvar1)
  let mvar2 ← mkFreshExprMVar (Expr.const ``Nat []) (userName := `mvar2)
  -- Create a fresh metavariable of type `Nat → Nat`. The `mkArrow` function
  -- creates a function type.
  let mvar3 ← mkFreshExprMVar (← mkArrow (.const ``Nat []) (.const ``Nat []))
    (userName := `mvar3)

  -- Define a helper function that prints each metavariable.
  let printMVars : MetaM Unit := do
    IO.println s!"  meta1: {← instantiateMVars mvar1}"
    IO.println s!"  meta2: {← instantiateMVars mvar2}"
    IO.println s!"  meta3: {← instantiateMVars mvar3}"

  IO.println "Initially, all metavariables are unassigned:"
  printMVars

  -- Assign `mvar1 : Nat := ?mvar3 ?mvar2`.
  mvar1.mvarId!.assign (.app mvar3 mvar2)
  IO.println "After assigning mvar1:"
  printMVars

  -- Assign `mvar2 : Nat := 0`.
  mvar2.mvarId!.assign (.const ``Nat.zero [])
  IO.println "After assigning mvar2:"
  printMVars

  -- Assign `mvar3 : Nat → Nat := Nat.succ`.
  mvar3.mvarId!.assign (.const ``Nat.succ [])
  IO.println "After assigning mvar3:"
  printMVars
-- Initially, all metavariables are unassigned:
--   meta1: ?_uniq.1
--   meta2: ?_uniq.2
--   meta3: ?_uniq.3
-- After assigning mvar1:
--   meta1: ?_uniq.3 ?_uniq.2
--   meta2: ?_uniq.2
--   meta3: ?_uniq.3
-- After assigning mvar2:
--   meta1: ?_uniq.3 Nat.zero
--   meta2: Nat.zero
--   meta3: ?_uniq.3
-- After assigning mvar3:
--   meta1: Nat.succ Nat.zero
```

```
--   meta2: Nat.zero
--   meta3: Nat.succ
```

Local Contexts

Consider the expression `e` which refers to the free variable with unique name `h`:

```
e := .fvar (FVarId.mk `h)
```

What is the type of this expression? The answer depends on the local context in which `e` is interpreted. One local context may declare that `h` is a local hypothesis of type `Nat`; another local context may declare that `h` is a local definition with value `List.map`.

Thus, expressions are only meaningful if they are interpreted in the local context for which they were intended. And as we saw, each metavariable has its own local context. So in principle, functions which manipulate expressions should have an additional `MVarId` argument specifying the goal in which the expression should be interpreted.

That would be cumbersome, so Lean goes a slightly different route. In `MetaM`, we always have access to an ambient `LocalContext`, obtained with `Lean.getLCtx` of type

```
getLCtx : MetaM LocalContext
```

All operations involving fvars use this ambient local context.

The downside of this setup is that we always need to update the ambient local context to match the goal we are currently working on. To do this, we use `Lean.MVarId.withContext` of type

```
withContext (mvarId : MVarId) (c : MetaM α) : MetaM α
```

This function takes a metavariable `mvarId` and a `MetaM` computation `c` and executes `c` with the ambient context set to the local context of `mvarId`. A typical use case looks like this:

```
def someTactic (mvarId : MVarId) ... : ... :=
  mvarId.withContext do
    ...
```

The tactic receives the current goal as the metavariable `mvarId` and immediately sets the current local context. Any operations within the `do` block then use the local context of `mvarId`.

Once we have the local context properly set, we can manipulate fvars. Like metavariables, fvars are identified by an `FVarId` (a unique `Name`). Basic operations include:

- `Lean.FVarId.getDecl : FVarId → MetaM LocalDecl` retrieves the declaration of a local hypothesis. As with metavariables, a `LocalDecl` contains all information pertaining to the local hypothesis, e.g. its type and its user-facing name.
- `Lean.Meta.getLocalDeclFromUserName : Name → MetaM LocalDecl` retrieves the declaration of the local hypothesis with the given user-facing name. If there are multiple such hypotheses, the bottommost one is returned. If there is none, an exception is thrown.

We can also iterate over all hypotheses in the local context, using the `ForIn` instance of `LocalContext`. A typical pattern is this:

```
for ldecl in ← getLCtx do
  if ldecl.isImplementationDetail then
    continue
  -- do something with the ldecl
```

The loop iterates over every `LocalDecl` in the context. The `isImplementationDetail` check skips local hypotheses which are 'implementation details', meaning they are introduced by Lean or by tactics for bookkeeping purposes. They are not shown to users and tactics are expected to ignore them.

At this point, we can build the `MetaM` part of an `assumption` tactic:

```
def myAssumption (mvarId : MVarId) : MetaM Bool := do
  -- Check that `mvarId` is not already assigned.
  mvarId.checkNotAssigned `myAssumption
  -- Use the local context of `mvarId`.
  mvarId.withContext do
    -- The target is the type of `mvarId`.
    let target ← mvarId.getType
    -- For each hypothesis in the local context:
    for ldecl in ← getLCtx do
      -- If the hypothesis is an implementation detail, skip it.
      if ldecl.isImplementationDetail then
        continue
      -- If the type of the hypothesis is definitionally equal to the target
      -- type:
      if ← isDefEq ldecl.type target then
        -- Use the local hypothesis to prove the goal.
        mvarId.assign ldecl.toExpr
        -- Stop and return true.
```

```
        return true
    -- If we have not found any suitable local hypothesis, return false.
    return false
```

The `myAssumption` tactic contains three functions we have not seen before:

- `Lean.MVarId.checkNotAssigned` checks that a metavariable is not already assigned. The 'myAssumption' argument is the name of the current tactic. It is used to generate a nicer error message.
- `Lean.Meta.isDefEq` checks whether two definitions are definitionally equal. See the Definitional Equality section.
- `Lean.LocalDecl.toExpr` is a helper function which constructs the `fvar` expression corresponding to a local hypothesis.

Delayed Assignments

The above discussion of metavariable assignment contains a lie by omission: there are actually two ways to assign a metavariable. We have seen the regular way; the other way is called a *delayed assignment*.

We do not discuss delayed assignments in any detail here since they are rarely useful for tactic writing. If you want to learn more about them, see the comments in `MetavarContext.lean` in the Lean standard library. But they create two complications which you should be aware of.

First, delayed assignments make `Lean.MVarId.isAssigned` and `getExprMVarAssignment?` medium-calibre footguns. These functions only check for regular assignments, so you may need to use `Lean.MVarId.isDelayedAssigned` and `Lean.Meta.getDelayedMVarAssignment` as well.

Second, delayed assignments break an intuitive invariant. You may have assumed that any metavariable which remains in the output of `instantiateMVars` is unassigned, since the assigned metavariables have been substituted. But delayed metavariables can only be substituted once their assigned value contains no unassigned metavariables. So delayed-assigned metavariables can appear in an expression even after `instantiateMVars`.

Metavariable Depth

Metavariable depth is also a niche feature, but one that is occasionally useful. Any metavariable has a *depth* (a natural number), and a `MetavarContext` has a

corresponding depth as well. Lean only assigns a metavariable if its depth is equal to the depth of the current `MetavarContext`. Newly created metavariables inherit the `MetavarContext`'s depth, so by default every metavariable is assignable.

This setup can be used when a tactic needs some temporary metavariables and also needs to make sure that other, non-temporary metavariables will not be assigned. To ensure this, the tactic proceeds as follows:

1. Save the current `MetavarContext`.
2. Increase the depth of the `MetavarContext`.
3. Perform whatever computation is necessary, possibly creating and assigning metavariables. Newly created metavariables are at the current depth of the `MetavarContext` and so can be assigned. Old metavariables are at a lower depth, so cannot be assigned.
4. Restore the saved `MetavarContext`, thereby erasing all the temporary metavariables and resetting the `MetavarContext` depth.

This pattern is encapsulated in `Lean.Meta.withNewMCtxDepth`.

Computation

Computation is a core concept of dependent type theory. The terms `2`, `Nat.succ 1` and `1 + 1` are all "the same" in the sense that they compute the same value. We call them *definitionally equal*. The problem with this, from a metaprogramming perspective, is that definitionally equal terms may be represented by entirely different expressions, but our users would usually expect that a tactic which works for `2` also works for `1 + 1`. So when we write our tactics, we must do additional work to ensure that definitionally equal terms are treated similarly.

Full Normalisation

The simplest thing we can do with computation is to bring a term into normal form. With some exceptions for numeric types, the normal form of a term `t` of type `T` is a sequence of applications of `T`'s constructors. E.g. the normal form of a list is a sequence of applications of `List.cons` and `List.nil`.

The function that normalises a term (i.e. brings it into normal form) is `Lean.Meta.reduce` with type signature

```
reduce (e : Expr) (explicitOnly skipTypes skipProofs := true) : MetaM Expr
```

We can use it like this:

```
def someNumber : Nat := (· + 2) $ 3

#eval Expr.const ``someNumber []
-- Lean.Expr.const `someNumber []

#eval reduce (Expr.const ``someNumber [])
-- Lean.Expr.lit (Lean.Literal.natVal 5)
```

Incidentally, this shows that the normal form of a term of type `Nat` is not always an application of the constructors of `Nat`; it can also be a literal. Also note that `#eval` can be used not only to evaluate a term, but also to execute a `MetaM` program.

The optional arguments of `reduce` allow us to skip certain parts of an expression. E.g. `reduce e (explicitOnly := true)` does not normalise any implicit arguments in the expression `e`. This yields better performance: since normal forms can be very big, it may be a good idea to skip parts of an expression that the user is not going to see anyway.

The `#reduce` command is essentially an application of `reduce`:

```
#reduce someNumber
-- 5
```

Transparency

An ugly but important detail of Lean 4 metaprogramming is that any given expression does not have a single normal form. Rather, it has a normal form up to a given *transparency*.

A transparency is a value of `Lean.Meta.TransparencyMode`, an enumeration with four values: `reducible`, `instances`, `default` and `all`. Any `MetaM` computation has access to an ambient `TransparencyMode` which can be obtained with `Lean.Meta.getTransparency`.

The current transparency determines which constants get unfolded during normalisation, e.g. by `reduce`. (To unfold a constant means to replace it with its definition.) The four settings unfold progressively more constants:

- `reducible`: unfold only constants tagged with the `@[reducible]` attribute. Note that `abbrev` is a shorthand for `@[reducible] def`.

- `instances`: unfold reducible constants and constants tagged with the `@[instance]` attribute. Again, the `instance` command is a shorthand for `@[instance] def`.
- `default`: unfold all constants except those tagged as `@[irreducible]`.
- `all`: unfold all constants, even those tagged as `@[irreducible]`.

The ambient transparency is usually `default`. To execute an operation with a specific transparency, use `Lean.Meta.withTransparency`. There are also shorthands for specific transparencies, e.g. `Lean.Meta.withReducible`.

Putting everything together for an example (where we use `Lean.Meta.ppExpr` to pretty-print an expression):

```
def traceConstWithTransparency (md : TransparencyMode) (c : Name) :
    MetaM Format := do
  ppExpr (← withTransparency md $ reduce (.const c []))

@[irreducible] def irreducibleDef : Nat      := 1
def                defaultDef     : Nat      := irreducibleDef + 1
abbrev             reducibleDef   : Nat      := defaultDef + 1
```

We start with `reducible` transparency, which only unfolds `reducibleDef`:

```
#eval traceConstWithTransparency .reducible ``reducibleDef
-- defaultDef + 1
```

If we repeat the above command but let Lean print implicit arguments as well, we can see that the `+` notation amounts to an application of the `hAdd` function, which is a member of the `HAdd` typeclass:

```
set_option pp.explicit true
#eval traceConstWithTransparency .reducible ``reducibleDef
-- @HAdd.hAdd Nat Nat Nat (@instHAdd Nat instAddNat) defaultDef 1
```

When we reduce with `instances` transparency, this applications is unfolded and replaced by `Nat.add`:

```
#eval traceConstWithTransparency .instances ``reducibleDef
-- Nat.add defaultDef 1
```

With `default` transparency, `Nat.add` is unfolded as well:

```
#eval traceConstWithTransparency .default ``reducibleDef
-- Nat.succ (Nat.succ irreducibleDef)
```

And with `TransparencyMode.all`, we're finally able to unfold `irreducibleDef`:

```
#eval traceConstWithTransparency .all ``reducibleDef
-- 3
```

The `#eval` commands illustrate that the same term, `reducibleDef`, can have a different normal form for each transparency.

Why all this ceremony? Essentially for performance: if we allowed normalisation to always unfold every constant, operations such as type class search would become prohibitively expensive. The tradeoff is that we must choose the appropriate transparency for each operation that involves normalisation.

Weak Head Normalisation

Transparency addresses some of the performance issues with normalisation. But even more important is to recognise that for many purposes, we don't need to fully normalise terms at all. Suppose we are building a tactic that automatically splits hypotheses of the type `P ∧ Q`. We might want this tactic to recognise a hypothesis `h : X` if `X` reduces to `P ∧ Q`. But if `P` additionally reduces to `Y ∨ Z`, the specific `Y` and `Z` do not concern us. Reducing `P` would be unnecessary work.

This situation is so common that the fully normalising `reduce` is in fact rarely used. Instead, the normalisation workhorse of Lean is `whnf`, which reduces an expression to *weak head normal form* (WHNF).

Roughly speaking, an expression `e` is in weak-head normal form when it has the form

```
e = f x₁ ... xₙ   (n ≥ 0)
```

and `f` cannot be reduced (at the current transparency). To conveniently check the WHNF of an expression, we define a function `whnf'`, using some functions that will be discussed in the Elaboration chapter.

```
open Lean.Elab.Term in
def whnf' (e : TermElabM Syntax) : TermElabM Format := do
  let e ← elabTermAndSynthesize (← e) none
  ppExpr (← whnf e)
```

Now, here are some examples of expressions in WHNF.

Constructor applications are in WHNF (with some exceptions for numeric types):

```
#eval whnf' `(List.cons 1 [])
-- [1]
```

The *arguments* of an application in WHNF may or may not be in WHNF themselves:

```
#eval whnf' `(List.cons (1 + 1) [])
-- [1 + 1]
```

Applications of constants are in WHNF if the current transparency does not allow us to unfold the constants:

```
#eval withTransparency .reducible $ whnf' `(List.append [1] [2])
-- List.append [1] [2]
```

Lambdas are in WHNF:

```
#eval whnf' `(λ x : Nat => x)
-- fun x => x
```

Foralls are in WHNF:

```
#eval whnf' `(∀ x, x > 0)
-- ∀ (x : Nat), x > 0
```

Sorts are in WHNF:

```
#eval whnf' `(Type 3)
-- Type 3
```

Literals are in WHNF:

```
#eval whnf' `((15 : Nat))
-- 15
```

Here are some more expressions in WHNF which are a bit tricky to test:

```
?x 0 1  -- Assuming the metavariable `?x` is unassigned, it is in WHNF.
h 0 1   -- Assuming `h` is a local hypothesis, it is in WHNF.
```

On the flipside, here are some expressions that are not in WHNF.

Applications of constants are not in WHNF if the current transparency allows us to unfold the constants:

```
#eval whnf' `(List.append [1])
-- fun x => 1 :: List.append [] x
```

Applications of lambdas are not in WHNF:

```
#eval whnf' `((λ x y : Nat => x + y) 1)
-- `fun y => 1 + y`
```

`let` bindings are not in WHNF:

```
#eval whnf' `(let x : Nat := 1; x)
-- 1
```

And again some tricky examples:

```
?x 0 1 -- Assuming `?x` is assigned (e.g. to `Nat.add`), its application is not
          in WHNF.
h 0 1  -- Assuming `h` is a local definition (e.g. with value `Nat.add`), its
          application is not in WHNF.
```

Returning to the tactic that motivated this section, let us write a function that matches a type of the form `P ∧ Q`, avoiding extra computation. WHNF makes it easy:

```
def matchAndReducing (e : Expr) : MetaM (Option (Expr × Expr)) := do
  match ← whnf e with
  | (.app (.app (.const ``And _) P) Q) => return some (P, Q)
  | _ => return none
```

By using `whnf`, we ensure that if `e` evaluates to something of the form `P ∧ Q`, we'll notice. But at the same time, we don't perform any unnecessary computation in `P` or `Q`.

However, our 'no unnecessary computation' mantra also means that if we want to perform deeper matching on an expression, we need to use `whnf` multiple times. Suppose we want to match a type of the form `P ∧ Q ∧ R`. The correct way to do this uses `whnf` twice:

```
def matchAndReducing₂ (e : Expr) : MetaM (Option (Expr × Expr × Expr)) := do
  match ← whnf e with
  | (.app (.app (.const ``And _) P) e') =>
    match ← whnf e' with
    | (.app (.app (.const ``And _) Q) R) => return some (P, Q, R)
    | _ => return none
  | _ => return none
```

This sort of deep matching up to computation could be automated. But until someone builds this automation, we have to figure out the necessary `whnf`s ourselves.

Definitional Equality

As mentioned, definitional equality is equality up to computation. Two expressions `t` and `s` are definitionally equal or *defeq* (at the current transparency) if their normal forms (at the current transparency) are equal.

To check whether two expressions are defeq, use `Lean.Meta.isDefEq` with type signature

```
isDefEq : Expr → Expr → MetaM Bool
```

Even though definitional equality is defined in terms of normal forms, `isDefEq` does not actually compute the normal forms of its arguments, which would be very expensive. Instead, it tries to "match up" `t` and `s` using as few reductions as possible. This is a necessarily heuristic endeavour and when the heuristics misfire, `isDefEq` can become very expensive. In the worst case, it may have to reduce `s` and `t` so often that they end up in normal form anyway. But usually the heuristics are good and `isDefEq` is reasonably fast.

If expressions `t` and `u` contain assignable metavariables, `isDefEq` may assign these metavariables to make `t` defeq to `u`. We also say that `isDefEq` *unifies* `t` and `u`; such unification queries are sometimes written `t =?= u`. For instance, the unification `List ?m =?= List Nat` succeeds and assigns `?m := Nat`. The unification `Nat.succ ?m =?= n + 1` succeeds and assigns `?m := n`. The unification `?m₁ + ?m₂ + ?m₃ =?= m + n - k` fails and no metavariables are assigned (even though there is a 'partial match' between the expressions).

Whether `isDefEq` considers a metavariable assignable is determined by two factors:

1. The metavariable's depth must be equal to the current `MetavarContext` depth. See the Metavariable Depth section.
2. Each metavariable has a *kind* (a value of type `MetavarKind`) whose sole purpose is to modify the behaviour of `isDefEq`. Possible kinds are:
    - Natural: `isDefEq` may freely assign the metavariable. This is the default.
    - Synthetic: `isDefEq` may assign the metavariable, but avoids doing so if possible. For example, suppose `?n` is a natural metavariable and `?s` is a synthetic metavariable. When faced with the unification problem `?s =?= ?n`, `isDefEq` assigns `?n` rather than `?s`.
    - Synthetic opaque: `isDefEq` never assigns the metavariable.

Constructing Expressions

In the previous chapter, we saw some primitive functions for building expressions: `Expr.app`, `Expr.const`, `mkAppN` and so on. There is nothing wrong with these functions, but the additional facilities of `MetaM` often provide more convenient ways.

Applications

When we write regular Lean code, Lean helpfully infers many implicit arguments and universe levels. If it did not, our code would look rather ugly:

```
def appendAppend (xs ys : List α) := (xs.append ys).append xs

set_option pp.all true in
set_option pp.explicit true in
#print appendAppend
-- def appendAppend.{u_1} : {α : Type u_1} → List.{u_1} α → List.{u_1} α → List.{u_1} α
 ↪  :=
-- fun {α : Type u_1} (xs ys : List.{u_1} α) => @List.append.{u_1} α (@List.append.{u_1}
 ↪  α xs ys) xs
```

The `.{u_1}` suffixes are universe levels, which must be given for every polymorphic constant. And of course the type α is passed around everywhere.

Exactly the same problem occurs during metaprogramming when we construct expressions. A hand-made expression representing the right-hand side of the above definition looks like this:

```
def appendAppendRHSExpr₁ (u : Level) (α xs ys : Expr) : Expr :=
  mkAppN (.const ``List.append [u])
    #[α, mkAppN (.const ``List.append [u]) #[α, xs, ys], xs]
```

Having to specify the implicit arguments and universe levels is annoying and error-prone. So `MetaM` provides a helper function which allows us to omit implicit information: `Lean.Meta.mkAppM` of type

```
mkAppM : Name → Array Expr → MetaM Expr
```

Like `mkAppN`, `mkAppM` constructs an application. But while `mkAppN` requires us to give all universe levels and implicit arguments ourselves, `mkAppM` infers them. This means we only need to provide the explicit arguments, which makes for a much shorter example:

```
def appendAppendRHSExpr₂ (xs ys : Expr) : MetaM Expr := do
  mkAppM ``List.append #[← mkAppM ``List.append #[xs, ys], xs]
```

Note the absence of any αs and us. There is also a variant of `mkAppM`, `mkAppM'`, which takes an `Expr` instead of a `Name` as the first argument, allowing us to construct applications of expressions which are not constants.

However, `mkAppM` is not magic: if you write `mkAppM ``List.append #[]`, you will get an error at runtime. This is because `mkAppM` tries to determine what the type α is, but with no arguments given to `append`, α could be anything, so `mkAppM` fails.

Another occasionally useful variant of `mkAppM` is `Lean.Meta.mkAppOptM` of type

```
mkAppOptM : Name → Array (Option Expr) → MetaM Expr
```

Whereas `mkAppM` always infers implicit and instance arguments and always requires us to give explicit arguments, `mkAppOptM` lets us choose freely which arguments to provide and which to infer. With this, we can, for example, give instances explicitly, which we use in the following example to give a non-standard `Ord` instance.

```
def revOrd : Ord Nat where
  compare x y := compare y x

def ordExpr : MetaM Expr := do
  mkAppOptM ``compare #[none, Expr.const ``revOrd [], mkNatLit 0, mkNatLit 1]

#eval format <$> ordExpr
-- Ord.compare.{0} Nat revOrd
--   (OfNat.ofNat.{0} Nat 0 (instOfNatNat 0))
--   (OfNat.ofNat.{0} Nat 1 (instOfNatNat 1))
```

Like `mkAppM`, `mkAppOptM` has a primed variant `Lean.Meta.mkAppOptM'` which takes an `Expr` instead of a `Name` as the first argument. The file which contains `mkAppM` also contains various other helper functions, e.g. for making list literals or `sorry`s.

Lambdas and Foralls

Another common task is to construct expressions involving `λ` or `∀` binders. Suppose we want to create the expression `λ (x : Nat), Nat.add x x`. One way is to write out the lambda directly:

```
def doubleExpr₁ : Expr :=
  .lam `x (.const ``Nat []) (mkAppN (.const ``Nat.add []) #[.bvar 0, .bvar 0])
    BinderInfo.default

#eval ppExpr doubleExpr₁
-- fun x => Nat.add x x
```

This works, but the use of `bvar` is highly unidiomatic. Lean uses a so-called *locally closed* variable representation. This means that all but the lowest-level functions in the Lean API expect expressions not to contain 'loose `bvar`s', where a `bvar` is loose if it is not bound by a binder in the same expression. (Outsied of Lean, such variables are usually called 'free'. The name `bvar` – 'bound variable' – already indicates that `bvar`s are never supposed to be free.)

As a result, if in the above example we replace `mkAppN` with the slightly higher-level `mkAppM`, we get a runtime error. Adhering to the locally closed convention, `mkAppM` expects any expressions given to it to have no loose bound variables, and `.bvar 0` is precisely that.

So instead of using `bvars` directly, the Lean way is to construct expressions with bound variables in two steps:

1. Construct the body of the expression (in our example: the body of the lambda), using temporary local hypotheses (`fvars`) to stand in for the bound variables.
2. Replace these `fvars` with `bvars` and, at the same time, add the corresponding lambda binders.

This process ensures that we do not need to handle expressions with loose `bvars` at any point (except during step 2, which is performed 'atomically' by a bespoke function). Applying the process to our example:

```
def doubleExpr₂ : MetaM Expr :=
  withLocalDecl `x BinderInfo.default (.const ``Nat []) λ x => do
    let body ← mkAppM ``Nat.add #[x, x]
    mkLambdaFVars #[x] body

#eval show MetaM _ from do
  ppExpr (← doubleExpr₂)
-- fun x => Nat.add x x
```

There are two new functions. First, `Lean.Meta.withLocalDecl` has type

```
withLocalDecl (name : Name) (bi : BinderInfo) (type : Expr) (k : Expr → MetaM α) : MetaM
 ↪  α
```

Given a variable name, binder info and type, `withLocalDecl` constructs a new `fvar` and passes it to the computation `k`. The `fvar` is avaible in the local context during the execution of `k` but is deleted again afterwards.

The second new function is `Lean.Meta.mkLambdaFVars` with type (ignoring some optional arguments)

```
mkLambdaFVars : Array Expr → Expr → MetaM Expr
```

This function takes an array of `fvars` and an expression `e`. It then adds one lambda binder for each `fvar x` and replaces every occurence of `x` in `e` with a bound variable corresponding to the new lambda binder. The returned expression does not contain the `fvars` any more, which is good since they disappear after we leave the `withLocalDecl` context. (Instead of `fvars`, we can also give `mvars` to `mkLambdaFVars`, despite its name.)

Some variants of the above functions may be useful:

- `withLocalDecls` declares multiple temporary `fvars`.
- `mkForallFVars` creates `∀` binders instead of `λ` binders. `mkLetFVars` creates `let` binders.
- `mkArrow` is the non-dependent version of `mkForallFVars` which construcs a function type `X → Y`. Since the type is non-dependent, there is no need for temporary `fvars`.

Using all these functions, we can construct larger expressions such as this one:

```
λ (f : Nat → Nat), ∀ (n : Nat), f n = f (n + 1)
```

```
def somePropExpr : MetaM Expr := do
  let funcType ← mkArrow (.const ``Nat []) (.const ``Nat [])
  withLocalDecl `f BinderInfo.default funcType fun f => do
    let feqn ← withLocalDecl `n BinderInfo.default (.const ``Nat []) fun n => do
      let lhs := .app f n
      let rhs := .app f (← mkAppM ``Nat.succ #[n])
      let eqn ← mkEq lhs rhs
      mkForallFVars #[n] eqn
    mkLambdaFVars #[f] feqn
```

The next line registers `someProp` as a name for the expression we've just constructed, allowing us to play with it more easily. The mechanisms behind this are discussed in the Elaboration chapter.

```
elab "someProp" : term => somePropExpr

#check someProp
-- fun f => ∀ (n : Nat), f n = f (Nat.succ n) : (Nat → Nat) → Prop
#reduce someProp Nat.succ
-- ∀ (n : Nat), Nat.succ n = Nat.succ (Nat.succ n)
```

Deconstructing Expressions

Just like we can construct expressions more easily in `MetaM`, we can also deconstruct them more easily. Particularly useful is a family of functions for deconstructing expressions which start with `λ` and `∀` binders.

When we are given a type of the form `∀ (x₁ : T₁) ... (xₙ : Tₙ), U`, we are often interested in doing something with the conclusion `U`. For instance, the `apply` tactic, when given an expression `e : ∀ ..., U`, compares `U` with the current target to determine whether `e` can be applied.

To do this, we could repeatedly match on the type expression, removing ∀ binders until we get to `U`. But this would leave us with an `U` containing unbound `bvars`, which, as we saw, is bad. Instead, we use `Lean.Meta.forallTelescope` of type

```
forallTelescope (type : Expr) (k : Array Expr → Expr → MetaM α) : MetaM α
```

Given `type = ∀ (x₁ : T₁) ... (xₙ : Tₙ), U x₁ ... xₙ`, this function creates one fvar `fᵢ` for each ∀-bound variable `xᵢ` and replaces each `xᵢ` with `fᵢ` in the conclusion `U`. It then calls the computation `k`, passing it the `fᵢ` and the conclusion `U f₁ ... fₙ`. Within this computation, the `fᵢ` are registered in the local context; afterwards, they are deleted again (similar to `withLocalDecl`).

There are many useful variants of `forallTelescope`:

- `forallTelescopeReducing`: like `forallTelescope` but matching is performed up to computation. This means that if you have an expression `X` which is different from but defeq to `∀ x, P x`, `forallTelescopeReducing X` will deconstruct `X` into `x` and `P x`. The non-reducing `forallTelescope` would not recognise `X` as a quantified expression. The matching is performed by essentially calling `whnf` repeatedly, using the ambient transparency.
- `forallBoundedTelescope`: like `forallTelescopeReducing` (even though there is no "reducing" in the name) but stops after a specified number of ∀ binders.
- `forallMetaTelescope`, `forallMetaTelescopeReducing`, `forallMetaBoundedTelescope`: like the corresponding non-`meta` functions, but the bound variables are replaced by new `mvars` instead of `fvars`. Unlike the non-`meta` functions, the `meta` functions do not delete the new metavariables after performing some computation, so the metavariables remain in the environment indefinitely.
- `lambdaTelescope`, `lambdaTelescopeReducing`, `lambdaBoundedTelescope`, `lambdaMetaTelescope`: like the corresponding `forall` functions, but for λ binders instead of ∀.

Using one of the telescope functions, we can implement our own `apply` tactic:

```
def myApply (goal : MVarId) (e : Expr) : MetaM (List MVarId) := do
  -- Check that the goal is not yet assigned.
  goal.checkNotAssigned `myApply
  -- Operate in the local context of the goal.
  goal.withContext do
    -- Get the goal's target type.
    let target ← goal.getType
    -- Get the type of the given expression.
    let type ← inferType e
    -- If `type` has the form `∀ (x₁ : T₁) ... (xₙ : Tₙ), U`, introduce new
```

```
    -- metavariables for the `xᵢ` and obtain the conclusion `U`. (If `type` does
    -- not have this form, `args` is empty and `conclusion = type`.)
    let (args, _, conclusion) ← forallMetaTelescopeReducing type
    -- If the conclusion unifies with the target:
    if ← isDefEq target conclusion then
      -- Assign the goal to `e x₁ ... xₙ`, where the `xᵢ` are the fresh
      -- metavariables in `args`.
      goal.assign (mkAppN e args)
      -- `isDefEq` may have assigned some of the `args`. Report the rest as new
      -- goals.
      let newGoals ← args.filterMapM λ mvar => do
        let mvarId := mvar.mvarId!
        if ! (← mvarId.isAssigned) && ! (← mvarId.isDelayedAssigned) then
          return some mvarId
        else
          return none
      return newGoals.toList
    -- If the conclusion does not unify with the target, throw an error.
    else
      throwTacticEx `myApply goal m!"{e} is not applicable to goal with target {target}"
```

The real `apply` does some additional pre- and postprocessing, but the core logic is what we show here. To test our tactic, we need an elaboration incantation, more about which in the Elaboration chapter.

```
elab "myApply" e:term : tactic => do
  let e ← Elab.Term.elabTerm e none
  Elab.Tactic.liftMetaTactic (myApply · e)

example (h : α → β) (a : α) : β := by
  myApply h
  myApply a
```

Backtracking

Many tactics naturally require backtracking: the ability to go back to a previous state, as if the tactic had never been executed. A few examples:

- `first | t | u` first executes `t`. If `t` fails, it backtracks and executes `u`.
- `try t` executes `t`. If `t` fails, it backtracks to the initial state, erasing any changes made by `t`.
- `trivial` attempts to solve the goal using a number of simple tactics (e.g. `rfl` or `contradiction`). After each unsuccessful application of such a tactic, `trivial`

backtracks.

Good thing, then, that Lean's core data structures are designed to enable easy and efficient backtracking. The corresponding API is provided by the `Lean.MonadBacktrack` class. `MetaM`, `TermElabM` and `TacticM` are all instances of this class. (`CoreM` is not but could be.)

`MonadBacktrack` provides two fundamental operations:

- `Lean.saveState : m s` returns a representation of the current state, where `m` is the monad we are in and `s` is the state type. E.g. for `MetaM`, `saveState` returns a `Lean.Meta.SavedState` containing the current environment, the current `MetavarContext` and various other pieces of information.
- `Lean.restoreState : s → m Unit` takes a previously saved state and restores it. This effectively resets the compiler state to the previous point.

With this, we can roll our own `MetaM` version of the `try` tactic:

```
def tryM (x : MetaM Unit) : MetaM Unit := do
  let s ← saveState
  try
    x
  catch _ =>
    restoreState s
```

We first save the state, then execute `x`. If `x` fails, we backtrack the state.

The standard library defines many combinators like `tryM`. Here are the most useful ones:

- `Lean.withoutModifyingState (x : m α) : m α` executes the action `x`, then resets the state and returns `x`'s result. You can use this, for example, to check for definitional equality without assigning metavariables:

  ```
  withoutModifyingState $ isDefEq x y
  ```

  If `isDefEq` succeeds, it may assign metavariables in `x` and `y`. Using `withoutModifyingState`, we can make sure this does not happen.

- `Lean.observing? (x : m α) : m (Option α)` executes the action `x`. If `x` succeeds, `observing?` returns its result. If `x` fails (throws an exception), `observing?` backtracks the state and returns `none`. This is a more informative version of our `tryM` combinator.

- `Lean.commitIfNoEx (x : α) : m α` executes `x`. If `x` succeeds, `commitIfNoEx` returns its result. If `x` throws an exception, `commitIfNoEx` backtracks the state and rethrows the exception.

Note that the builtin `try ... catch ... finally` does not perform any backtracking. So code which looks like this is probably wrong:

```
try
  doSomething
catch e =>
  doSomethingElse
```

The `catch` branch, `doSomethingElse`, is executed in a state containing whatever modifications `doSomething` made before it failed. Since we probably want to erase these modifications, we should write instead:

```
try
  commitIfNoEx doSomething
catch e =>
  doSomethingElse
```

Another `MonadBacktrack` gotcha is that `restoreState` does not backtrack the *entire* state. Caches, trace messages and the global name generator, among other things, are not backtracked, so changes made to these parts of the state are not reset by `restoreState`. This is usually what we want: if a tactic executed by `observing?` produces some trace messages, we want to see them even if the tactic fails. See `Lean.Meta.SavedState.restore` and `Lean.Core.restore` for details on what is and is not backtracked.

In the next chapter, we move towards the topic of elaboration, of which you've already seen several glimpses in this chapter. We start by discussing Lean's syntax system, which allows you to add custom syntactic constructs to the Lean parser.

Syntax

This chapter is concerned with the means to declare and operate on syntax in Lean. Since there are a multitude of ways to operate on it, we will not go into great detail about this yet and postpone quite a bit of this to later chapters.

Declaring Syntax

Declaration helpers

Some readers might be familiar with the `infix` or even the `notation` commands, for those that are not here is a brief recap:

```
import Lean

-- XOR, denoted \oplus
infixl:60 " ⊕ " => fun l r => (!l && r) || (l && !r)

#eval true ⊕ true -- false
#eval true ⊕ false -- true
#eval false ⊕ true -- true
#eval false ⊕ false -- false

-- with `notation`, "left XOR"
notation:10 l:10 " LXOR " r:11 => (!l && r)

#eval true LXOR true -- false
#eval true LXOR false -- false
#eval false LXOR true -- true
#eval false LXOR false -- false
```

As we can see the `infixl` command allows us to declare a notation for a binary operation that is infix, meaning that the operator is in between the operands (as opposed to e.g. before which would be done using the `prefix` command). The `l` at the end of `infixl` means that the notation is left associative so `a ⊕ b ⊕ c` gets parsed as `(a ⊕ b) ⊕ c` as opposed to `a ⊕ (b ⊕ c)` (which would be achieved by

`infixr`). On the right hand side, it expects a function that operates on these two parameters and returns some value. The `notation` command, on the other hand, allows us some more freedom: we can just "mention" the parameters right in the syntax definition and operate on them on the right hand side. It gets even better though, we can in theory create syntax with 0 up to as many parameters as we wish using the `notation` command, it is hence also often referred to as "mixfix" notation.

The two unintuitive parts about these two are: - The fact that we are leaving spaces around our operators: `" ⊕ "`," LXOR ". This is so that, when Lean pretty prints our syntax later on, it also uses spaces around the operators, otherwise the syntax would just be presented as `l⊕r` as opposed to `l ⊕ r`. - The `60` and `10` right after the respective commands – these denote the operator precedence, meaning how strong they bind to their arguments, let's see this in action:

```
#eval true ⊕ false LXOR false -- false
#eval (true ⊕ false) LXOR false -- false
#eval true ⊕ (false LXOR false) -- true
```

As we can see, the Lean interpreter analyzed the first term without parentheses like the second instead of the third one. This is because the `⊕` notation has higher precedence than `LXOR` (`60 > 10` after all) and is thus evaluated before it. This is also how you might implement rules like * being evaluated before +.

Lastly at the `notation` example there are also these `:precedence` bindings at the arguments: `l:10` and `r:11`. This conveys that the left argument must have precedence at least 10 or greater, and the right argument must have precedence at 11 or greater. The way the arguments are assigned their respective precedence is by looking at the precedence of the rule that was used to parse them. Consider for example `a LXOR b LXOR c`. Theoretically speaking this could be parsed in two ways: 1. `(a LXOR b) LXOR c` 2. `a LXOR (b LXOR c)`

Since the arguments in parentheses are parsed by the `LXOR` rule with precedence 10 they will appear as arguments with precedence 10 to the outer `LXOR` rule: 1. `(a LXOR b):10 LXOR c` 2. `a LXOR (b LXOR c):10`

However if we check the definition of `LXOR`: `notation:10 l:10 " LXOR " r:11` we can see that the right hand side argument requires a precedence of at least 11 or greater, thus the second parse is invalid and we remain with: `(a LXOR b) LXOR c` assuming that: - `a` has precedence 10 or higher - `b` has precedence 11 or higher - `c` has precedence 11 or higher

Thus `LXOR` is a left associative notation. Can you make it right associative?

NOTE: If parameters of a notation are not explicitly given a precedence they will implicitly be tagged with precedence 0.

As a last remark for this section: Lean will always attempt to obtain the longest matching parse possible, this has three important implications. First a very intuitive one, if we have a right associative operator ^ and Lean sees something like `a ^ b ^ c`, it will first parse the `a ^ b` and then attempt to keep parsing (as long as precedence allows it) until it cannot continue anymore. Hence Lean will parse this expression as `a ^ (b ^ c)` (as we would expect it to).

Secondly, if we have a notation where precedence does not allow to figure out how the expression should be parenthesized, for example:

```
notation:65 lhs:65 " ~ " rhs:65 => (lhs - rhs)
```

An expression like `a ~ b ~ c` will be parsed as `a ~ (b ~ c)` because Lean attempts to find the longest parse possible. As a general rule of thumb: If precedence is ambiguous Lean will default to right associativity.

```
#eval 5 ~ 3 ~ 3 -- 5 because this is parsed as 5 - (3 - 3)
```

Lastly, if we define overlapping notation such as:

```
-- define `a ~ b mod rel` to mean that a and b are equivalent with respect to some
 ↪  equivalance relation rel
notation:65 a:65 " ~ " b:65 " mod " rel:65 => rel a b
```

Lean will prefer this notation over parsing `a ~ b` as defined above and then erroring because it doesn't know what to do with `mod` and the relation argument:

```
#check 0 ~ 0 mod Eq -- 0 = 0 : Prop
```

This is again because it is looking for the longest possible parser which in this case involves also consuming `mod` and the relation argument.

Free form syntax declarations

With the above `infix` and `notation` commands, you can get quite far with declaring ordinary mathematical syntax already. Lean does however allow you to introduce arbitrarily complex syntax as well. This is done using two main commands `syntax` and `declare_syntax_cat`. A `syntax` command allows you add a new syntax rule to an already existing so-called "syntax category". The most common syntax categories are: - `term`, this category will be discussed in detail in the elaboration chapter, for

now you can think of it as “the syntax of everything that has a value” - `command`, this is the category for top-level commands like `#check`, `def` etc. - TODO: …

Let’s see this in action:

```
syntax "MyTerm" : term
```

We can now write `MyTerm` in place of things like `1 + 1` and it will be *syntactically* valid, this does not mean the code will compile yet, it just means that the Lean parser can understand it:

```
def Playground1.test := MyTerm
-- elaboration function for 'termMyTerm' has not been implemented
--   MyTerm
```

Implementing this so-called “elaboration function”, which will actually give meaning to this syntax in terms of Lean’s fundamental `Expr` type, is topic of the elaboration chapter.

The `notation` and `infix` commands are utilities that conveniently bundle syntax declaration with macro definition (for more on macros, see the macro chapter), where the contents left of the `=>` declare the syntax. All the previously mentioned principles from `notation` and `infix` regarding precedence fully apply to `syntax` as well.

We can, of course, also involve other syntax into our own declarations in order to build up syntax trees. For example, we could try to build our own little boolean expression language:

```
namespace Playground2

-- The scoped modifier makes sure the syntax declarations remain in this `namespace`
-- because we will keep modifying this along the chapter
scoped syntax "⊥" : term -- ⊥ for false
scoped syntax "⊤" : term -- ⊤ for true
scoped syntax:40 term " OR " term : term
scoped syntax:50 term " AND " term : term
#check ⊥ OR (⊤ AND ⊥) -- elaboration function hasn't been implemented but parsing passes

end Playground2
```

While this does work, it allows arbitrary terms to the left and right of our `AND` and `OR` operation. If we want to write a mini language that only accepts our boolean language on a syntax level we will have to declare our own syntax category on top. This is done using the `declare_syntax_cat` command:

```
declare_syntax_cat boolean_expr
syntax "⊥" : boolean_expr -- ⊥ for false
syntax "⊤" : boolean_expr -- ⊤ for true
syntax:40 boolean_expr " OR " boolean_expr : boolean_expr
syntax:50 boolean_expr " AND " boolean_expr : boolean_expr
```

Now that we are working in our own syntax category, we are completely disconnected from the rest of the system. And these cannot be used in place of terms anymore:

```
#check ⊥ AND ⊤ -- expected term
```

In order to integrate our syntax category into the rest of the system we will have to extend an already existing one with new syntax, in this case we will re-embed it into the `term` category:

```
syntax "[Bool|" boolean_expr "]" : term
#check [Bool| ⊥ AND ⊤] -- elaboration function hasn't been implemented but parsing
 ↪  passes
```

Syntax combinators

In order to declare more complex syntax, it is often very desirable to have some basic operations on syntax already built-in, these include: - helper parsers without syntax categories (i.e. not extendable) - alternatives - repetitive parts - optional parts While all of these do have an encoding based on syntax categories, this can make things quite ugly at times, so Lean provides an easier way to do all of these.

In order to see all of these in action, we will briefly define a simple binary expression syntax. First things first, declaring named parsers that don't belong to a syntax category is quite similar to ordinary `def`s:

```
syntax binOne := "O"
syntax binZero := "Z"
```

These named parsers can be used in the same positions as syntax categories from above, their only difference to them is, that they are not extensible. That is, they are directly expanded within syntax declarations, and we cannot define new patterns for them as we would with proper syntax categories. There does also exist a number of built-in named parsers that are generally useful, most notably: - `str` for string literals - `num` for number literals - `ident` for identifiers - ... TODO: better list or link to compiler docs

Next up we want to declare a parser that understands digits, a binary digit is either 0 or 1 so we can write:

```
syntax binDigit := binZero <|> binOne
```

Where the <|> operator implements the "accept the left or the right" behaviour. We can also chain them to achieve parsers that accept arbitrarily many, arbitrarly complex other ones. Now we will define the concept of a binary number, usually this would be written as digits directly after each other but we will instead use comma separated ones to showcase the repetition feature:

```
-- the "+" denotes "one or many", in order to achieve "zero or many" use "*" instead
-- the "," denotes the separator between the `binDigit`s, if left out the default
 ↪  separator is a space
syntax binNumber := binDigit,+
```

Since we can just use named parsers in place of syntax categories, we can now easily add this to the `term` category:

```
syntax "bin(" binNumber ")" : term
#check bin(Z, O, Z, Z, O) -- elaboration function hasn't been implemented but parsing
 ↪  passes
#check bin() -- fails to parse because `binNumber` is "one or many": expected 'O' or 'Z'

syntax binNumber' := binDigit,* -- note the *
syntax "emptyBin(" binNumber' ")" : term
#check emptyBin() -- elaboration function hasn't been implemented but parsing passes
```

Note that nothing is limiting us to only using one syntax combinator per parser, we could also have written all of this inline:

```
syntax "binCompact(" ("Z" <|> "O"),+ ")" : term
#check binCompact(Z, O, Z, Z, O) -- elaboration function hasn't been implemented but
 ↪  parsing passes
```

As a final feature, let's add an optional string comment that explains the binary literal being declared:

```
-- The (...)? syntax means that the part in parentheses is optional
syntax "binDoc(" (str ";")? binNumber ")" : term
#check binDoc(Z, O, Z, Z, O) -- elaboration function hasn't been implemented but parsing
 ↪  passes
#check binDoc("mycomment"; Z, O, Z, Z, O) -- elaboration function hasn't been
 ↪  implemented but parsing passes
```

Operating on Syntax

As explained above, we will not go into detail in this chapter on how to teach Lean about the meaning you want to give your syntax. We will, however, take a look at how to write functions that operate on it. Like all things in Lean, syntax is represented by the inductive type `Lean.Syntax`, on which we can operate. It does contain quite some information, but most of what we are interested in, we can condense in the following simplified view:

```
namespace Playground2

inductive Syntax where
  | missing : Syntax
  | node (kind : Lean.SyntaxNodeKind) (args : Array Syntax) : Syntax
  | atom : String -> Syntax
  | ident : Lean.Name -> Syntax

end Playground2
```

Lets go through the definition one constructor at a time: - `missing` is used when there is something the Lean compiler cannot parse, it is what allows Lean to have a syntax error in one part of the file but recover from it and try to understand the rest of it. This also means we pretty much don't care about this constructor. - `node` is, as the name suggests, a node in the syntax tree. It has a so called `kind : SyntaxNodeKind` where `SyntaxNodeKind` is just a `Lean.Name`. Basically, each of our `syntax` declarations receives an automatically generated `SyntaxNodeKind` (we can also explicitly specify the name with `syntax (name := foo) ... : cat`) so we can tell Lean "this function is responsible for processing this specific syntax construct". Furthermore, like all nodes in a tree, it has children, in this case in the form of an `Array Syntax`. - `atom` represents (with the exception of one) every syntax object that is at the bottom of the hierarchy. For example, our operators `⊕` and `LXOR` from above will be represented as atoms. - `ident` is the mentioned exception to this rule. The difference between `ident` and `atom` is also quite obvious: an identifier has a `Lean.Name` instead of a `String` that represents it. Why a `Lean.Name` is not just a `String` is related to a concept called macro hygiene that will be discussed in detail in the macro chapter. For now, you can consider them basically equivalent.

Constructing new Syntax

Now that we know how syntax is represented in Lean, we could of course write programs that generate all of these inductive trees by hand, which would be incredibly tedious and is something we most definitely want to avoid. Luckily for us there is quite an extensive API hidden inside the `Lean.Syntax` namespace we can explore:

```
open Lean
#check Syntax -- Syntax. autocomplete
```

The interesting functions for creating `Syntax` are the `Syntax.mk*` ones that allow us to create both very basic `Syntax` objects like `idents` but also more complex ones like `Syntax.mkApp` which we can use to create the `Syntax` object that would amount to applying the function from the first argument to the argument list (all given as `Syntax`) in the second one. Let's see a few examples:

```
-- Name literals are written with this little ` in front of the name
#eval Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumLit "1"] -- is
 ↪  the syntax of `Nat.add 1 1`
#eval mkNode `«term_+_» #[Syntax.mkNumLit "1", mkAtom "+", Syntax.mkNumLit "1"] -- is
 ↪  the syntax for `1 + 1`

-- note that the `«term_+_» is the auto-generated SyntaxNodeKind for the + syntax
```

If you don't like this way of creating `Syntax` at all you are not alone. However, there are a few things involved with the machinery of doing this in a pretty and correct (the machinery is mostly about the correct part) way which will be explained in the macro chapter.

Matching on Syntax

Just like constructing `Syntax` is an important topic, especially with macros, matching on syntax is equally (or in fact even more) interesting. Luckily we don't have to match on the inductive type itself either: we can instead use so-called "syntax patterns". They are quite simple, their syntax is just `` `(the syntax I want to match on) ``. Let's see one in action:

```
def isAdd11 : Syntax → Bool
  | `(Nat.add 1 1) => true
  | _ => false
```

```
#eval isAdd11 (Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumLit
↪  "1"]) -- true
#eval isAdd11 (Syntax.mkApp (mkIdent `Nat.add) #[mkIdent `foo, Syntax.mkNumLit "1"]) --
↪  false
```

The next level with matches is to capture variables from the input instead of just matching on literals, this is done with a slightly fancier-looking syntax:

```
def isAdd : Syntax → Option (Syntax × Syntax)
  | `(Nat.add $x $y) => some (x, y)
  | _ => none

#eval isAdd (Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumLit
↪  "1"]) -- some ...
#eval isAdd (Syntax.mkApp (mkIdent `Nat.add) #[mkIdent `foo, Syntax.mkNumLit "1"]) --
↪  some ...
#eval isAdd (Syntax.mkApp (mkIdent `Nat.add) #[mkIdent `foo]) -- none
```

Typed Syntax

Note that `x` and `y` in this example are of type `TSyntax `term`, not `Syntax`. Even though we are pattern matching on `Syntax` which, as we can see in the constructors, is purely composed of types that are not `TSyntax`, so what is going on? Basically the `` `() `` Syntax is smart enough to figure out the most general syntax category the syntax we are matching might be coming from (in this case `term`). It will then use the typed syntax type `TSyntax` which is parameterized by the `Name` of the syntax category it came from. This is not only more convenient for the programmer to see what is going on, it also has other benefits. For Example if we limit the syntax category to just `num` in the next example Lean will allow us to call `getNat` on the resulting `TSyntax `num` directly without pattern matching or the option to panic:

```
-- Now we are also explicitly marking the function to operate on term syntax
def isLitAdd : TSyntax `term → Option Nat
  | `(Nat.add $x:num $y:num) => some (x.getNat + y.getNat)
  | _ => none

#eval isLitAdd (Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumLit
↪  "1"]) -- some 2
#eval isLitAdd (Syntax.mkApp (mkIdent `Nat.add) #[mkIdent `foo, Syntax.mkNumLit "1"]) --
↪  none
```

If you want to access the `Syntax` behind a `TSyntax` you can do this using `TSyntax.raw` although the coercion machinery should just work most of the time. We will see some further benefits of the `TSyntax` system in the macro chapter.

One last important note about the matching on syntax: In this basic form it only works on syntax from the `term` category. If you want to use it to match on your own syntax categories you will have to use `` `(category| ...) ``.

Mini Project

As a final mini project for this chapter we will declare the syntax of a mini arithmetic expression language and a function of type `Syntax → Nat` to evaluate it. We will see more about some of the concepts presented below in future chapters.

```
declare_syntax_cat arith

syntax num : arith
syntax arith "-" arith : arith
syntax arith "+" arith : arith
syntax "(" arith ")" : arith

partial def denoteArith : TSyntax `arith → Nat
  | `(arith| $x:num) => x.getNat
  | `(arith| $x:arith + $y:arith) => denoteArith x + denoteArith y
  | `(arith| $x:arith - $y:arith) => denoteArith x - denoteArith y
  | `(arith| ($x:arith)) => denoteArith x
  | _ => 0

-- You can ignore Elab.TermElabM, what is important for us is that it allows
-- us to use the ``(arith| (12 + 3) - 4)` notation to construct `Syntax`
-- instead of only being able to match on it like this.
def test : Elab.TermElabM Nat := do
  let stx ← `(arith| (12 + 3) - 4)
  pure (denoteArith stx)

#eval test -- 11
```

Feel free to play around with this example and extend it in whatever way you want to. The next chapters will mostly be about functions that operate on `Syntax` in some way.

More elaborate examples

Using type classes for notations

We can use type classes in order to add notation that is extensible via the type instead of the syntax system, this is for example how + using the typeclasses `HAdd` and `Add` and other common operators in Lean are generically defined.

For example, we might want to have a generic notation for subset notation. The first thing we have to do is define a type class that captures the function we want to build notation for.

```
class Subset (α : Type u) where
  subset : α → α → Prop
```

The second step is to define the notation, what we can do here is simply turn every instance of a ⊆ appearing in the code to a call to `Subset.subset` because the type class resolution should be able to figure out which `Subset` instance is referred to. Thus the notation will be a simple:

```
-- precedence is arbitrary for this example
infix:50 " ⊆ " => Subset.subset
```

Let's define a simple theory of sets to test it:

```
-- a `Set` is defined by the elements it contains
-- -> a simple predicate on the type of its elements
def Set (α : Type u) := α → Prop

def Set.mem (x : α) (X : Set α) : Prop := X x

-- Integrate into the already existing typeclass for membership notation
instance : Membership α (Set α) where
  mem := Set.mem

def Set.empty : Set α := λ _ => False

instance : Subset (Set α) where
  subset X Y := ∀ (x : α), x ∈ X → x ∈ Y

example : ∀ (X : Set α), Set.empty ⊆ X := by
  intro X x
  -- ⊢ x ∈ Set.empty → x ∈ X
  intro h
  exact False.elim h -- empty set has no members
```

Binders

Because declaring syntax that uses variable binders used to be a rather unintuitive thing to do in Lean 3, we'll take a brief look at how naturally this can be done in Lean 4.

For this example we will define the well-known notation for the set that contains all elements `x` such that some property holds: `{x ∈ ℕ | x < 10}` for example.

First things first we need to extend the theory of sets from above slightly:

```
-- the basic "all elements such that" function for the notation
def setOf {α : Type} (p : α → Prop) : Set α := p
```

Equipped with this function, we can now attempt to intuitively define a basic version of our notation:

```
notation "{ " x " | " p " }" => setOf (fun x => p)

#check { (x : Nat) | x ≤ 1 } -- { x | x ≤ 1 } : Set Nat

example : 1 ∈ { (y : Nat) | y ≤ 1 } := by simp[Membership.mem, Set.mem, setOf]
example : 2 ∈ { (y : Nat) | y ≤ 3 ∧ 1 ≤ y } := by simp[Membership.mem, Set.mem, setOf]
```

This intuitive notation will indeed deal with what we could throw at it in the way we would expect it.

As to how one might extend this notation to allow more set-theoretic things such as `{x ∈ X | p x}` and leave out the parentheses around the bound variables, we refer the reader to the macro chapter.

Macros

What is a macro

Macros in Lean are `Syntax → MacroM Syntax` functions. `MacroM` is the macro monad which allows macros to have some static guarantees we will discuss in the next section, you can mostly ignore it for now.

Macros are registered as handlers for a specific syntax declaration using the `macro` attribute. The compiler will take care of applying these function to the syntax for us before performing actual analysis of the input. This means that the only thing we have to do is declare our syntax with a specific name and bind a function of type `Lean.Macro` to it. Let's try to reproduce the `LXOR` notation from the `Syntax` chapter:

```
import Lean

open Lean

syntax:10 (name := lxor) term:10 " LXOR " term:11 : term

@[macro lxor] def lxorImpl : Macro
  | `($l:term LXOR $r:term) => `(!$l && $r) -- we can use the quotation mechanism to
   ↪  create `Syntax` in macros
  | _ => Macro.throwUnsupported

#eval true LXOR true -- false
#eval true LXOR false -- false
#eval false LXOR true -- true
#eval false LXOR false -- false
```

That was quite easy! The `Macro.throwUnsupported` function can be used by a macro to indicate that "it doesn't feel responsible for this syntax". In this case it's merely used to fill a wildcard pattern that should never be reached anyways.

However we can in fact register multiple macros for the same syntax this way if we desire, they will be tried one after another (the later registered ones have

higher priority) – is "higher" correct? until one throws either a real error using `Macro.throwError` or succeeds, that is it does not `Macro.throwUnsupported`. Let's see this in action:

```
@[macro lxor] def lxorImpl2 : Macro
  -- special case that changes behaviour of the case where the left and
  -- right hand side are these specific identifiers
  | `(true LXOR true) => `(true)
  | _ => Macro.throwUnsupported

#eval true LXOR true -- true, handled by new macro
#eval true LXOR false -- false, still handled by the old
```

This capability is obviously *very* powerful! It should not be used lightly and without careful thinking since it can introduce weird behaviour while writing code later on. The following example illustrates this weird behaviour:

```
#eval true LXOR true -- true, handled by new macro

def foo := true
#eval foo LXOR foo -- false, handled by old macro, after all the identifiers have a
 ↪  different name
```

Without knowing exactly how this macro is implemented this behaviour will be very confusing to whoever might be debugging an issue based on this. The rule of thumb for when to use a macro vs. other mechanisms like elaboration is that as soon as you are building real logic like in the 2nd macro above, it should most likely not be a macro but an elaborator (explained in the elaboration chapter). This means ideally we want to use macros for simple syntax to syntax translations, that a human could easily write out themselves as well but is too lazy to.

Simplifying macro declaration

Now that we know the basics of what a macro is and how to register it we can take a look at slightly more automated ways to do this (in fact all of the ways about to be presented are implemented as macros themselves).

First things first there is `macro_rules` which basically desugars to functions like the ones we wrote above, for example:

```
syntax:10 term:10 " RXOR " term:11 : term

macro_rules
  | `($l:term RXOR $r:term) => `($l && !$r)
```

As you can see, it figures out lot's of things on its own for us: - the name of the syntax declaration - the `macro` attribute registration - the `throwUnsupported` wildcard

apart from this it just works like a function that is using pattern matching syntax, we can in theory encode arbitrarily complex macro functions on the right hand side.

If this is still not short enough for you, there is a next step using the `macro` macro:

```
macro l:term:10 " ⊕ " r:term:11 : term => `((!$l && $r) || ($l && !$r))

#eval true ⊕ true -- false
#eval true ⊕ false -- true
#eval false ⊕ true -- true
#eval false ⊕ false -- false
```

As you can see, `macro` is quite close to `notation` already: - it performed syntax declaration for us - it automatically wrote a `macro_rules` style function to match on it

The are of course differences as well: - `notation` is limited to the `term` syntax category - `notation` cannot have arbitrary macro code on the right hand side

`Syntax` Quotations

The basics

So far we've handwaved the `` `(foo $bar) `` syntax to both create and match on `Syntax` objects but it's time for a full explanation since it will be essential to all non trivial things that are syntax related.

First things first we call the `` `() `` syntax a `Syntax` quotation. When we plug variables into a syntax quotation like this: `` `($x) `` we call the `$x` part an anti-quotation. When we insert `x` like this it is required that `x` is of type `TSyntax x` where `x` is some `Name` of a syntax category. The Lean compiler is actually smart enough to figure the syntax categories that are allowed in this place out. Hence you might sometimes see errors of the form:

```
application type mismatch
  x.raw
argument
  x
has type
```

```
  TSyntax `a : Type
but is expected to have type
  TSyntax `b : Type
```

If you are sure that your thing from the `a` syntax category can be used as a `b` here you can declare a coercion of the form:

```
instance : Coe (TSyntax `a) (TSyntax `b) where
  coe s := ⟨s.raw⟩
```

Which will allow Lean to perform the type cast automatically. If you notice that your `a` can not be used in place of the `b` here congrats, you just discovered a bug in your `Syntax` function. Similar to the Lean compiler you could can also declare functions that are specific to certain `TSynax` variants. For example as we have seen in the syntax chapter there exists the function:

```
#check TSyntax.getNat -- TSyntax.getNat : TSyntax numLitKind → Nat
```

Which is guaranteed to not panic because we know that the `Syntax` that the function is receiving is a numeric literal and can thus naturally be converted to a `Nat`.

If we use the antiquotation syntax in pattern matching it will, as discussed in the syntax chapter, give us a a variable `x` of type `TSyntax y` where `y` is the `Name` of the syntax category that fits in the spot where we pattern matched. If we wish to insert a literal `$x` into the `Syntax` for some reason, for example macro creating macros, we can escape the anti quotation using: `` `($$x) ``.

If we want to specify the syntax kind we wish `x` to be interpreted as we can make this explicit using: `` `($x:term) `` where `term` can be replaced with any other valid syntax category (e.g. `command`) or parser (e.g. `ident`).

So far this is only a more formal explanation of the intuitive things we've already seen in the syntax chapter and up to now in this chapter, next we'll discuss some more advanced anti-quotations.

Advanced anti-quotations

For convenince we can also use anti-quotations in a way similar to format strings: `` `($(mkIdent `c)) `` is the same as: `let x := mkIdent `c; `($x)`.

Furthermore there are sometimes situations in which we are not working with basic `Syntax` but `Syntax` wrapped in more complex datastructures, most notably `Array (TSyntax c)` or `TSepArray c s`. Where `TSepArray c s`, is a `Syntax` specific

type, it is what we get if we pattern match on some `Syntax` that users a separator `s` to separate things from the category `c`. For example if we match using: `$xs,*`, `xs` will have type `TSepArray c ","`. With the special case of matching on no specific separator (i.e. whitespace): `$xs*` in which we will receive an `Array (TSyntax c)`.

If we are dealing with `xs : Array (TSyntax c)` and want to insert it into a quotation we have two main ways to achieve this: 1. Insert it using a separator, most commonly `,`: `` `($xs,*) ``. This is also the way to insert a `TSepArray c ","` 2. Insert it point blank without a separator (TODO): `` `() ``

For example:

```
-- syntactically cut away the first element of a tuple if possible
syntax "cut_tuple " "(" term ", " term,+ ")" : term

macro_rules
  -- cutting away one element of a pair isn't possible, it would not result in a tuple
  | `(cut_tuple ($x, $y)) => `(($x, $y))
  | `(cut_tuple ($x, $y, $xs,*)) => `(($y, $xs,*))

#check cut_tuple (1, 2) -- (1, 2) : Nat × Nat
#check cut_tuple (1, 2, 3) -- (2, 3) : Nat × Nat
```

The last thing for this section will be so called “anti-quotation splices”. There are two kinds of anti quotation splices, first the so called optional ones. For example we might declare a syntax with an optional argument, say our own `let` (in real projects this would most likely be a `let` in some functional language we are writing a theory about):

```
syntax "mylet " ident (" : " term)? " := " term " in " term : term
```

There is this optional `(" : " term)?` argument involved which can let the user define the type of the term to the left of it. With the methods we know so far we’d have to write two `macro_rules` now, one for the case with, one for the case without the optional argument. However the rest of the syntactic translation works exactly the same with and without the optional argument so what we can do using a splice here is to essentially define both cases at once:

```
macro_rules
  | `(mylet $x $[: $ty]? := $val in $body) => `(let $x $[: $ty]? := $val; $body)
```

The `$[...]?` part is the splice here, it basically says “if this part of the syntax isn’t there, just ignore the parts on the right hand side that involve anti quotation variables involved here”. So now we can run this syntax both with and without type ascription:

```
#eval mylet x := 5 in x - 10 -- 0, due to subtraction behaviour of `Nat`
#eval mylet x : Int := 5 in x - 10 -- -5, after all it is an `Int` now
```

The second and last splice might remind readers of list comprehension as seen for example in Python. We will demonstrate it using an implementation of `map` as a macro:

```
-- run the function given at the end for each element of the list
syntax "foreach " "[" term,* "]" term : term

macro_rules
  | `(foreach [ $[$x:term],* ] $func:term) => `(let f := $func; [ $[f $x],* ])

#eval foreach [1,2,3,4] (Nat.add 2) -- [3, 4, 5, 6]
```

In this case the `$[...],*` part is the splice. On the match side it tries to match the pattern we define inside of it repetetively (given the seperator we tell it to). However unlike regular separator matching it does not give us an `Array` or `SepArray`, instead it allows us to write another splice on the right hand side that gets evaluated for each time the pattern we specified matched, with the specific values from the match per iteration.

Hygiene issues and how to solve them

If you are familiar with macro systems in other languages like C you probably know about so called macro hygiene issues already. A hygiene issue is when a macro introduces an identifier that collides with an identifier from some syntax that it is including. For example:

```
-- Applying this macro produces a function that binds a new identifier `x`.
macro "const" e:term : term => `(fun x => $e)

-- But `x` can also be defined by a user
def x : Nat := 42

-- Which `x` should be used by the compiler in place of `$e`?
#eval (const x) 10 -- 42
```

Given the fact that macros perform only syntactic translations one might expect the above `eval` to return 10 instead of 42: after all, the resulting syntax should be `(fun x => x) 10`. While this was of course not the intention of the author, this is what would happen in more primitive macro systems like the one of C. So how does

Lean avoid these hygiene issues? You can read about this in detail in the excellent Beyond Notations paper which discusses the idea and implementation in Lean in detail. We will merely give an overview of the topic, since the details are not that interesting for practical uses. The idea described in Beyond Notations comes down to a concept called "macro scopes". Whenever a new macro is invoked, a new macro scope (basically a unique number) is added to a list of all the macro scopes that are active right now. When the current macro introduces a new identifier what is actually getting added is an identifier of the form:

```
<actual name>._@.(<module_name>.<scopes>)*.<module_name>._hyg.<scopes>
```

For example, if the module name is `Init.Data.List.Basic`, the name is `foo.bla`, and macros scopes are [2, 5] we get:

```
foo.bla._@.Init.Data.List.Basic._hyg.2.5
```

Since macro scopes are unique numbers the list of macro scopes appended in the end of the name will always be unique across all macro invocations, hence macro hygiene issues like the ones above are not possible.

If you are wondering why there is more than just the macro scopes to this name generation, that is because we may have to combine scopes from different files/modules. The main module being processed is always the right most one. This situation may happen when we execute a macro generated in a file imported in the current file.

```
foo.bla._@.Init.Data.List.Basic.2.1.Init.Lean.Expr_hyg.4
```

The delimiter `_hyg` at the end is used just to improve performance of the function `Lean.Name.hasMacroScopes` – the format could also work without it.

This was a lot of technical details. You do not have to understand them in order to use macros, if you want you can just keep in mind that Lean will not allow name clashes like the one in the `const` example.

Note that this extends to *all* names that are introduced using syntax quotations, that is if you write a macro that produces: `` `(def foo := 1) ``, the user will not be able to access `foo` because the name will subject to hygienie. Luckily there is a way to circumvent this. You can use `mkIdent` to generate a raw identifier, for example: `` `(def $(mkIdent `foo) := 1) ``. In this case it won't be subject to hygiene and accessible to the user.

`MonadQuotation` and `MonadRef`

Based on this description of the hygiene mechanism one interesting question pops up, how do we know what the current list of macro scopes actually is? After all in the macro functions that were defined above there is never any explicit passing around of the scopes happening. As is quite common in functional programming, as soon as we start having some additional state that we need to bookkeep (like the macro scopes) this is done with a monad, this is the case here as well with a slight twist.

Instead of implementing this for only a single monad `MacroM` the general concept of keeping track of macro scopes in monadic way is abstracted away using a type class called `MonadQuotation`. This allows any other monad to also easily provide this hygienic `Syntax` creation mechanism by simply implementing this type class.

This is also the reason that while we are able to use pattern matching on syntax with `` `(syntax) `` we cannot just create `Syntax` with the same syntax in pure functions: there is no `Monad` implementing `MonadQuotation` involved in order to keep track of the macro scopes.

Now let's take a brief look at the `MonadQuotation` type class:

```
namespace Playground

class MonadRef (m : Type → Type) where
  getRef      : m Syntax
  withRef {α} : Syntax → m α → m α

class MonadQuotation (m : Type → Type) extends MonadRef m where
  getCurrMacroScope : m MacroScope
  getMainModule     : m Name
  withFreshMacroScope {α : Type} : m α → m α

end Playground
```

Since `MonadQuotation` is based on `MonadRef`, let's take a look at `MonadRef` first. The idea here is quite simple: `MonadRef` is meant to be seen as an extension to the `Monad` typeclass which - gives us a reference to a `Syntax` value with `getRef` - can evaluate a certain monadic action `m α` with a new reference to a `Syntax` using `withRef`

On it's own `MonadRef` isn't exactly interesting, but once it is combined with `MonadQuotation` it makes sense.

As you can see `MonadQuotation` extends `MonadRef` and adds 3 new functions:

- `getCurrMacroScope` which obtains the latest `MacroScope` that was created - `getMainModule` which (obviously) obtains the name of the main module, both of these are used to create these hygienic identifiers explained above - `withFreshMacroScope` which will compute the next macro scope and run some computation `m α` that performs syntax quotation with this new macro scope in order to avoid name clashes. While this is mostly meant to be used internally whenever a new macro invocation happens, it can sometimes make sense to use this in our own macros, for example when we are generating some syntax block repeatedly and want to avoid name clashes.

How `MonadRef` comes into play here is that Lean requires a way to indicate errors at certain positions to the user. One thing that wasn't introduced in the `Syntax` chapter is that values of type `Syntax` actually carry their position in the file around as well. When an error is detected, it is usually bound to a `Syntax` value which tells Lean where to indicate the error in the file. What Lean will do when using `withFreshMacroScope` is to apply the position of the result of `getRef` to each introduced symbol, which then results in better error positions than not applying any position.

To see error positioning in action, we can write a little macro that makes use of it:

```
syntax "error_position" ident : term

macro_rules
  | `(error_position all) => Macro.throwError "Ahhh"
  -- the `%$tk` syntax gives us the Syntax of the thing before the %,
  -- in this case `error_position`, giving it the name `tk`
  | `(error_position%$tk first) => withRef tk (Macro.throwError "Ahhh")

#eval error_position all -- the error is indicated at `error_position all`
#eval error_position first -- the error is only indicated at `error_position`
```

Obviously controlling the positions of errors in this way is quite important for a good user experience.

Mini project

As a final mini project for this section we will re-build the arithmetic DSL from the syntax chapter in a slightly more advanced way, using a macro this time so we can actually fully integrate it into the Lean syntax.

```
declare_syntax_cat arith

syntax num : arith
syntax arith "-" arith : arith
syntax arith "+" arith : arith
syntax "(" arith ")" : arith
syntax "[Arith|" arith "]" : term

macro_rules
  | `([Arith| $x:num]) => `($x)
  | `([Arith| $x:arith + $y:arith]) => `([Arith| $x] + [Arith| $y]) -- recursive macros
   ↪ are possible
  | `([Arith| $x:arith - $y:arith]) => `([Arith| $x] - [Arith| $y])
  | `([Arith| ($x:arith)]) => `([Arith| $x])

#eval [Arith| (12 + 3) - 4] -- 11
```

Again feel free to play around with it. If you want to build more complex things, like expressions with variables, maybe consider building an inductive type using macros instead. Once you got your arithmetic expression term as an inductive, you could then write a function that takes some form of variable assignment and evaluates the given expression for this assignment. You could also try to embed arbitrary `terms` into your arith language using some special syntax or whatever else comes to your mind.

More elaborate examples

Binders 2.0

As promised in the syntax chapter here is Binders 2.0. We'll start by reintroducing our theory of sets:

```
def Set (α : Type u) := α → Prop
def Set.mem (x : α) (X : Set α) : Prop := X x

-- Integrate into the already existing typeclass for membership notation
instance : Membership α (Set α) where
  mem := Set.mem

def Set.empty : Set α := λ _ => False

-- the basic "all elements such that" function for the notation
def setOf {α : Type} (p : α → Prop) : Set α := p
```

The goal for this section will be to allow for both `{x : X | p x}` and `{x ∈ X, p x}` notations. In principle there are two ways to do this: 1. Define a syntax and macro for each way to bind a variable we might think of 2. Define a syntax cateogry of binders that we could reuse across other binder constructs such as Σ or Π as well and implement macros for the `{ | }` case

In this section we will use approach 2 because it is more easily reusable.

```
declare_syntax_cat binder_construct
syntax "{" binder_construct "|" term "}" : term
```

Now let's define the two binders constructs we are interested in:

```
syntax ident " : " term : binder_construct
syntax ident " ∈ " term : binder_construct
```

And finally the macros to expand our syntax:

```
macro_rules
  | `({ $var:ident : $ty:term | $body:term }) => `(setOf (fun ($var : $ty) => $body))
  | `({ $var:ident ∈ $s:term | $body:term }) => `(setOf (fun $var => $var ∈ $s ∧ $body))

-- Old examples with better syntax:
#check { x : Nat | x ≤ 1 } -- setOf fun x => x ≤ 1 : Set Nat

example : 1 ∈ { y : Nat | y ≤ 1 } := by simp[Membership.mem, Set.mem, setOf]
example : 2 ∈ { y : Nat | y ≤ 3 ∧ 1 ≤ y } := by simp[Membership.mem, Set.mem, setOf]

-- New examples:
def oneSet : Set Nat := λ x => x = 1
#check { x ∈ oneSet | 10 ≤ x } -- setOf fun x => x ∈ oneSet ∧ 10 ≤ x : Set Nat

example : ∀ x, ¬(x ∈ { y ∈ oneSet | y ≠ 1 }) := by
  intro x h
  -- h : x ∈ setOf fun y => y ∈ oneSet ∧ y ≠ 1
  -- ⊢ False
  cases h
  -- : x ∈ oneSet
  -- : x ≠ 1
  contradiction
```

Reading further

If you want to know more about macros you can read: - the API docs: TODO link - the source code: the lower parts of Init.Prelude as you can see they are declared quite

early in Lean because of their importance to building up syntax - the aforementioned Beyond Notations paper

Elaboration

The elaborator is the component in charge of turning the user facing `Syntax` into something with which the rest of the compiler can work. Most of the time, this means translating `Syntax` into `Exprs` but there are also other use cases such as `#check` or `#eval`. Hence the elaborator is quite a large piece of code, it lives here.

Command elaboration

A command is the highest level of `Syntax`, a Lean file is made up of a list of commands. The most commonly used commands are declarations, for example: - `def` - `inductive` - `structure`

but there are also other ones, most notably `#check`, `#eval` and friends. All commands live in the `command` syntax category so in order to declare custom commands, their syntax has to be registered in that category.

Giving meaning to commands

The next step is giving some semantics to the syntax. With commands, this is done by registering a so called command elaborator.

Command elaborators have type `CommandElab` which is an alias for: `Syntax → CommandElabM Unit`. What they do, is take the `Syntax` that represents whatever the user wants to call the command and produce some sort of side effect on the `CommandElabM` monad, after all the return value is always `Unit`. The `CommandElabM` monad has 4 main kinds of side effects: 1. Logging messages to the user via the `Monad` extensions `MonadLog` and `AddMessageContext`, like `#check`. This is done via functions that can be found in `Lean.Elab.Log`, the most notable ones being: `logInfo`, `logWarning` and `logError`. 2. Interacting with the `Environment` via the `Monad` extension `MonadEnv`. This is the place where all of the relevant information for the compiler is stored, all known declarations, their types, doc-strings, values etc.

The current environment can be obtained via `getEnv` and set via `setEnv` once it has been modified. Note that quite often wrappers around `setEnv` like `addDecl` are the correct way to add information to the `Environment`. 3. Performing `IO`, `CommandElabM` is capable of running any `IO` operation. For example reading from files and based on their contents perform declarations. 4. Throwing errors, since it can run any kind of `IO`, it is only natural that it can throw errors via `throwError`.

Furthermore there are a bunch of other `Monad` extensions that are supported by `CommandElabM`: - `MonadRef` and `MonadQuotation` for `Syntax` quotations like in macros - `MonadOptions` to interact with the options framework - `MonadTrace` for debug trace information - TODO: There are a few others though I'm not sure whether they are relevant, see the instance in `Lean.Elab.Command`

Command elaboration

Now that we understand the type of command elaborators let's take a brief look at how the elaboration process actually works: 1. Check whether any macros can be applied to the current `Syntax`. If there is a macro that does apply and does not throw an error the resulting `Syntax` is recursively elaborated as a command again. 2. If no macro can be applied, we search for all `CommandElab`s that have been registered for the `SyntaxKind` of the `Syntax` we are elaborating, using the `command_elab` attribute. 3. All of these `CommandElab` are then tried in order until one of them does not throw an `unsupportedSyntaxException`, Lean's way of indicating that the elaborator "feels responsible" for this specific `Syntax` construct. Note that it can still throw a regular error to indicate to the user that something is wrong. If no responsible elaborator is found, then the command elaboration is aborted with an `unexpected syntax` error message.

As you can see the general idea behind the procedure is quite similar to ordinary macro expansion.

Making our own

Now that we know both what a `CommandElab` is and how they are used, we can start looking into writing our own. The steps for this, as we learned above, are: 1. Declaring the syntax 2. Declaring the elaborator 3. Registering the elaborator as responsible for the syntax via the `command_elab` attribute.

Let's see how this is done:

```
import Lean

open Lean Elab Command Term Meta

syntax (name := mycommand1) "#mycommand1" : command -- declare the syntax

@[command_elab mycommand1]
def mycommand1Impl : CommandElab := fun stx => do -- declare and register the elaborator
  logInfo "Hello World"

#mycommand1 -- Hello World
```

You might think that this is a little boiler-platey and it turns out the Lean devs did as well so they added a macro for this!

```
elab "#mycommand2" : command =>
  logInfo "Hello World"

#mycommand2 -- Hello World
```

Note that, due to the fact that command elaboration supports multiple registered elaborators for the same syntax, we can in fact overload syntax, if we want to.

```
@[command_elab mycommand1]
def myNewImpl : CommandElab := fun stx => do
  logInfo "new!"

#mycommand1 -- new!
```

Furthermore it is also possible to only overload parts of syntax by throwing an `unsupportedSyntaxException` in the cases we want the default handler to deal with it or just letting the `elab` command handle it.

In the following example, we are not extending the original `#check` syntax, but adding a new `SyntaxKind` for this specific syntax construct. However, from the point of view of the user, the effect is basically the same.

```
elab "#check" "mycheck" : command => do
  logInfo "Got ya!"
```

This is actually extending the original `#check`

```
@[command_elab Lean.Parser.Command.check] def mySpecialCheck : CommandElab := fun stx =>
 ↪  do
  if let some str := stx[1].isStrLit? then
    logInfo s!"Specially elaborated string literal!: {str} : String"
  else
```

```
    throwUnsupportedSyntax

#check mycheck -- Got ya!
#check "Hello" -- Specially elaborated string literal!: Hello : String
#check Nat.add -- Nat.add : Nat → Nat → Nat
```

Mini project

As a final mini project for this section let's build a command elaborator that is actually useful. It will take a command and use the same mechanisms as `elabCommand` (the entry point for command elaboration) to tell us which macros or elaborators are relevant to the command we gave it.

We will not go through the effort of actually reimplementing `elabCommand` though

```
elab "#findCElab " c:command : command => do
  let macroRes ← liftMacroM <| expandMacroImpl? (←getEnv) c
  match macroRes with
  | some (name, _) => logInfo s!"Next step is a macro: {name.toString}"
  | none =>
    let kind := c.raw.getKind
    let elabs := commandElabAttribute.getEntries (←getEnv) kind
    match elabs with
    | [] => logInfo s!"There is no elaborators for your syntax, looks like its bad :("
    | _ => logInfo s!"Your syntax may be elaborated by: {elabs.map (fun el =>
     ↪  el.declName.toString)}"

#findCElab def lala := 12 -- Your syntax may be elaborated by:
 ↪  [Lean.Elab.Command.elabDeclaration]
#findCElab abbrev lolo := 12 -- Your syntax may be elaborated by:
 ↪  [Lean.Elab.Command.elabDeclaration]
#findCElab #check foo -- even our own syntax!: Your syntax may be elaborated by:
 ↪  [mySpecialCheck, Lean.Elab.Command.elabCheck]
#findCElab open Hi -- Your syntax may be elaborated by: [Lean.Elab.Command.elabOpen]
#findCElab namespace Foo -- Your syntax may be elaborated by:
 ↪  [Lean.Elab.Command.elabNamespace]
#findCElab #findCElab open Bar -- even itself!: Your syntax may be elaborated by:
 ↪  [«_aux_lean_elaboration___elabRules_command#findCElab__1»]
```

TODO: Maybe we should also add a mini project that demonstrates a non # style command aka a declaration, although nothing comes to mind right now. TODO: Define a `conjecture` declaration, similar to `lemma/theorem`, except that it is automatically sorried. The `sorry` could be a custom one, to reflect that the "conjecture" might be expected to be true.

Term elaboration

A term is a `Syntax` object that represents some sort of `Expr`. Term elaborators are the ones that do the work for most of the code we write. Most notably they elaborate all the values of things like definitions, types (since these are also just `Expr`) etc.

All terms live in the `term` syntax category (which we have seen in action in the macro chapter already). So, in order to declare custom terms, their syntax needs to be registered in that category.

Giving meaning to terms

As with command elaboration, the next step is giving some semantics to the syntax. With terms, this is done by registering a so called term elaborator.

Term elaborators have type `TermElab` which is an alias for: `Syntax → Option Expr → TermElabM Expr`. This type is already quite different from command elaboration: - As with command elaboration the `Syntax` is whatever the user used to create this term - The `Option Expr` is the expected type of the term, since this cannot always be known it is only an `Option` argument - Unlike command elaboration, term elaboration is not only executed because of its side effects – the `TermElabM Expr` return value does actually contain something of interest, namely, the `Expr` that represents the `Syntax` object.

`TermElabM` is basically an upgrade of `CommandElabM` in every regard: it supports all the capabilities we mentioned above, plus two more. The first one is quite simple: On top of running `IO` code it is also capable of running `MetaM` code, so `Expr`s can be constructed nicely. The second one is very specific to the term elaboration loop.

Term elaboration

The basic idea of term elaboration is the same as command elaboration: expand macros and recurse or run term elaborators that have been registered for the `Syntax` via the `term_elab` attribute (they might in turn run term elaboration) until we are done. There is, however, one special action that a term elaborator can do during its execution.

A term elaborator may throw `Except.postpone`. This indicates that the term elaborator requires more information to continue its work. In order to represent this

missing information, Lean uses so called synthetic metavariables. As you know from before, metavariables are holes in `Exprs` that are waiting to be filled in. Synthetic metavariables are different in that they have special methods that are used to solve them, registered in `SyntheticMVarKind`. Right now, there are four of these: - `typeClass`, the metavariable should be solved with typeclass synthesis - `coe`, the metavariable should be solved via coercion (a special case of typeclass) - `tactic`, the metavariable is a tactic term that should be solved by running a tactic - `postponed`, the ones that are created at `Except.postpone`

Once such a synthetic metavariable is created, the next higher level term elaborator will continue. At some point, execution of postponed metavariables will be resumed by the term elaborator, in hopes that it can now complete its execution. We can try to see this in action with the following example:

```
#check set_option trace.Elab.postpone true in List.foldr .add 0 [1,2,3] --
 ↪  [Elab.postpone] .add : ?m.5695 → ?m.5696 → ?m.5696
```

What happened here is that the elaborator for function applications started at `List.foldr` which is a generic function so it created metavariables for the implicit type parameters. Then, it attempted to elaborate the first argument `.add`.

In case you don't know how `.name` works, the basic idea is that quite often (like in this case) Lean should be able to infer the output type (in this case `Nat`) of a function (in this case `Nat.add`). In such cases, the `.name` feature will then simply search for a function named `name` in the namespace `Nat`. This is especially useful when you want to use constructors of a type without referring to its namespace or opening it, but can also be used like above.

Now back to our example, while Lean does at this point already know that `.add` needs to have type: `?m1 → ?m2 → ?m2` (where `?x` is notation for a metavariable) the elaborator for `.add` does need to know the actual value of `?m2` so the term elaborator postpones execution (by internally creating a synthetic metavariable in place of `.add`), the elaboration of the other two arguments then yields the fact that `?m2` has to be `Nat` so once the `.add` elaborator is continued it can work with this information to complete elaboration.

We can also easily provoke cases where this does not work out. For example:

```
#check set_option trace.Elab.postpone true in List.foldr .add
-- [Elab.postpone] .add : ?m.5808 → ?m.5809 → ?m.5809
-- invalid dotted identifier notation, expected type is not of the form (... → C ...)
 ↪  where C is a constant
  -- ?m.5808 → ?m.5809 → ?m.5809
```

In this case `.add` first postponed its execution, then got called again but didn't have enough information to finish elaboration and thus failed.

Making our own

Adding new term elaborators works basically the same way as adding new command elaborators so we'll only take a very brief look:

```
syntax (name := myterm1) "myterm 1" : term

def mytermValues := [1, 2]

@[term_elab myterm1]
def myTerm1Impl : TermElab := fun stx type? =>
  mkAppM ``List.get! #[.const ``mytermValues [], mkNatLit 0] -- `MetaM` code

#eval myterm 1 -- 1

-- Also works with `elab`
elab "myterm 2" : term => do
  mkAppM ``List.get! #[.const ``mytermValues [], mkNatLit 1] -- `MetaM` code

#eval myterm 2 -- 2
```

Mini project

As a final mini project for this chapter we will recreate one of the most commonly used Lean syntax sugars, the `⟨a,b,c⟩` notation as a short hand for single constructor types:

```
-- slightly different notation so no ambiguity happens
syntax (name := myanon) "⟨⟨" term,* "⟩⟩" : term

def getCtors (typ : Name) : MetaM (List Name) := do
  let env ← getEnv
  match env.find? typ with
  | some (ConstantInfo.inductInfo val) =>
    pure val.ctors
  | _ => pure []

@[term_elab myanon]
def myanonImpl : TermElab := fun stx typ? => do
  -- Attempt to postpone execution if the type is not known or is a metavariable.
  -- Metavariables are used by things like the function elaborator to fill
```

```
  -- out the values of implicit parameters when they haven't gained enough
  -- information to figure them out yet.
  -- Term elaborators can only postpone execution once, so the elaborator
  -- doesn't end up in an infinite loop. Hence, we only try to postpone it,
  -- otherwise we may cause an error.
  tryPostponeIfNoneOrMVar typ?
  -- If we haven't found the type after postponing just error
  let some typ := typ? | throwError "expected type must be known"
  if typ.isMVar then
    throwError "expected type must be known"
  let Expr.const base .. := typ.getAppFn | throwError s!"type is not of the expected
   ↪  form: {typ}"
  let [ctor] ← getCtors base | throwError "type doesn't have exactly one constructor"
  let args := TSyntaxArray.mk stx[1].getSepArgs
  let stx ← `($(mkIdent ctor) $args*) -- syntax quotations
  elabTerm stx typ -- call term elaboration recursively

#check (⟨1, sorry⟩ : Fin 12) -- { val := 1, isLt := (_ : 1 < 12) } : Fin 12
#check ⟨1, sorry⟩ -- expected type must be known
#check (⟨0⟩ : Nat) -- type doesn't have exactly one constructor
#check (⟨⟩ : Nat → Nat) -- type is not of the expected form: Nat -> Nat
```

As a final note, we can shorten the postponing act by using an additional syntax sugar of the `elab` syntax instead:

```
-- This `t` syntax will effectively perform the first two lines of `myanonImpl`
elab "⟨⟨" args:term,* "⟩⟩" : term <= t => do
  sorry
```

Embedding DSLs By Elaboration

In this chapter we will learn how to use elaboration to build a DSL. We will not explore the full power of `MetaM`, and simply gesture at how to get access to this low-level machinery.

More precisely, we shall enable Lean to understand the syntax of IMP, which is a simple imperative language, often used for teaching operational and denotational semantics.

We are not going to define everything with the same encoding that the book does. For instance, the book defines arithmetic expressions and boolean expressions. We, will take a different path and just define generic expressions that take unary or binary operators.

This means that we will allow weirdnesses like `1 + true`! But it will simplify the encoding, the grammar and consequently the metaprogramming didactic.

Defining our AST

We begin by defining our atomic literal value.

```
import Lean

open Lean Elab Meta

inductive IMPLit
  | nat  : Nat  → IMPLit
  | bool : Bool → IMPLit
```

This is our only unary operator

```
inductive IMPUnOp
  | not
```

These are our binary operations.

```
inductive IMPBinOp
  | and | add | less
```

Now we define the expressions that we want to handle.

```
inductive IMPExpr
  | lit : IMPLit → IMPExpr
  | var : String → IMPExpr
  | un  : IMPUnOp → IMPExpr → IMPExpr
  | bin : IMPBinOp → IMPExpr → IMPExpr → IMPExpr
```

And finally the commands of our language. Let's follow the book and say that "each piece of a program is also a program":

```
inductive IMPProgram
  | Skip   : IMPProgram
  | Assign : String → IMPExpr → IMPProgram
  | Seq    : IMPProgram → IMPProgram → IMPProgram
  | If     : IMPExpr → IMPProgram → IMPProgram → IMPProgram
  | While  : IMPExpr → IMPProgram → IMPProgram
```

Elaborating literals

Now that we have our data types, let's elaborate terms of `Syntax` into terms of `Expr`. We begin by defining the syntax and an elaboration function for literals.

```
declare_syntax_cat imp_lit
syntax num       : imp_lit
syntax "true"    : imp_lit
syntax "false"   : imp_lit

def elabIMPLit : Syntax → MetaM Expr
  -- `mkAppM` creates an `Expr.app`, given the function `Name` and the args
  -- `mkNatLit` creates an `Expr` from a `Nat`
  | `(imp_lit| $n:num) => mkAppM ``IMPLit.nat  #[mkNatLit n.getNat]
  | `(imp_lit| true  ) => mkAppM ``IMPLit.bool #[.const ``Bool.true []]
  | `(imp_lit| false ) => mkAppM ``IMPLit.bool #[.const ``Bool.false []]
  | _ => throwUnsupportedSyntax

elab "test_elabIMPLit " l:imp_lit : term => elabIMPLit l

#reduce test_elabIMPLit 4     -- IMPLit.nat 4
#reduce test_elabIMPLit true  -- IMPLit.bool true
#reduce test_elabIMPLit false -- IMPLit.bool true
```

Elaborating expressions

In order to elaborate expressions, we also need a way to elaborate our unary and binary operators.

Notice that these could very much be pure functions (`Syntax → Expr`), but we're staying in `MetaM` because it allows us to easily throw an error for match completion.

```
declare_syntax_cat imp_unop
syntax "not"     : imp_unop

def elabIMPUnOp : Syntax → MetaM Expr
  | `(imp_unop| not) => return .const ``IMPUnOp.not []
  | _ => throwUnsupportedSyntax

declare_syntax_cat imp_binop
syntax "+"       : imp_binop
syntax "and"     : imp_binop
syntax "<"       : imp_binop

def elabIMPBinOp : Syntax → MetaM Expr
  | `(imp_binop| +)   => return .const ``IMPBinOp.add []
  | `(imp_binop| and) => return .const ``IMPBinOp.and []
  | `(imp_binop| <)   => return .const ``IMPBinOp.less []
  | _ => throwUnsupportedSyntax
```

Now we define the syntax for expressions:

```
declare_syntax_cat                  imp_expr
syntax imp_lit                    : imp_expr
syntax ident                      : imp_expr
syntax imp_unop imp_expr          : imp_expr
syntax imp_expr imp_binop imp_expr : imp_expr
```

Let's also allow parentheses so the IMP programmer can denote their parsing precedence.

```
syntax "(" imp_expr ")" : imp_expr
```

Now we can elaborate our expressions. Note that expressions can be recursive. This means that our `elabIMPExpr` function will need to be recursive! We'll need to use `partial` because Lean can't prove the termination of `Syntax` consumption alone.

```
partial def elabIMPExpr : Syntax → MetaM Expr
  | `(imp_expr| $l:imp_lit) => do
    let l ← elabIMPLit l
```

```
    mkAppM ``IMPExpr.lit #[l]
  -- `mkStrLit` creates an `Expr` from a `String`
  | `(imp_expr| $i:ident) => mkAppM ``IMPExpr.var #[mkStrLit i.getId.toString]
  | `(imp_expr| $b:imp_unop $e:imp_expr) => do
    let b ← elabIMPUnOp b
    let e ← elabIMPExpr e -- recurse!
    mkAppM ``IMPExpr.un #[b, e]
  | `(imp_expr| $l:imp_expr $b:imp_binop $r:imp_expr) => do
    let l ← elabIMPExpr l -- recurse!
    let r ← elabIMPExpr r -- recurse!
    let b ← elabIMPBinOp b
    mkAppM ``IMPExpr.bin #[b, l, r]
  | `(imp_expr| ($e:imp_expr)) => elabIMPExpr e
  | _ => throwUnsupportedSyntax

elab "test_elabIMPExpr " e:imp_expr : term => elabIMPExpr e

#reduce test_elabIMPExpr a
-- IMPExpr.var "a"

#reduce test_elabIMPExpr a + 5
-- IMPExpr.bin IMPBinOp.add (IMPExpr.var "a") (IMPExpr.lit (IMPLit.nat 5))

#reduce test_elabIMPExpr 1 + true
-- IMPExpr.bin IMPBinOp.add (IMPExpr.lit (IMPLit.nat 1)) (IMPExpr.lit (IMPLit.bool
 ↪  true))
```

Elaborating programs

And now we have everything we need to elaborate our IMP programs!

```
declare_syntax_cat           imp_program
syntax "skip"               : imp_program
syntax ident ":=" imp_expr : imp_program

syntax imp_program ";;" imp_program : imp_program

syntax "if" imp_expr "then" imp_program "else" imp_program "fi" : imp_program
syntax "while" imp_expr "do" imp_program "od" : imp_program

partial def elabIMPProgram : Syntax → MetaM Expr
  | `(imp_program| skip) => return .const ``IMPProgram.Skip []
  | `(imp_program| $i:ident := $e:imp_expr) => do
    let i : Expr := mkStrLit i.getId.toString
    let e ← elabIMPExpr e
```

```
    mkAppM ``IMPProgram.Assign #[i, e]
  | `(imp_program| $p₁:imp_program ;; $p₂:imp_program) => do
    let p₁ ← elabIMPProgram p₁
    let p₂ ← elabIMPProgram p₂
    mkAppM ``IMPProgram.Seq #[p₁, p₂]
  | `(imp_program| if $e:imp_expr then $pT:imp_program else $pF:imp_program fi) => do
    let e ← elabIMPExpr e
    let pT ← elabIMPProgram pT
    let pF ← elabIMPProgram pF
    mkAppM ``IMPProgram.If #[e, pT, pF]
  | `(imp_program| while $e:imp_expr do $pT:imp_program od) => do
    let e ← elabIMPExpr e
    let pT ← elabIMPProgram pT
    mkAppM ``IMPProgram.While #[e, pT]
  | _ => throwUnsupportedSyntax
```

And we can finally test our full elaboration pipeline. Let's use the following syntax:

```
elab ">>" p:imp_program "<<" : term => elabIMPProgram p

#reduce >>
a := 5;;
if not a and 3 < 4 then
  c := 5
else
  a := a + 1
fi;;
b := 10
<<
-- IMPProgram.Seq (IMPProgram.Assign "a" (IMPExpr.lit (IMPLit.nat 5)))
--   (IMPProgram.Seq
--     (IMPProgram.If
--       (IMPExpr.un IMPUnOp.not
--         (IMPExpr.bin IMPBinOp.and (IMPExpr.var "a")
--           (IMPExpr.bin IMPBinOp.less (IMPExpr.lit (IMPLit.nat 3)) (IMPExpr.lit
 ↪ (IMPLit.nat 4)))))
--       (IMPProgram.Assign "c" (IMPExpr.lit (IMPLit.nat 5)))
--       (IMPProgram.Assign "a" (IMPExpr.bin IMPBinOp.add (IMPExpr.var "a") (IMPExpr.lit
 ↪ (IMPLit.nat 1)))))
--     (IMPProgram.Assign "b" (IMPExpr.lit (IMPLit.nat 10))))
```

Tactics

Tactics are Lean programs that manipulate a custom state. All tactics are, in the end, of type `TacticM Unit`. This has the type:

```
-- from Lean/Elab/Tactic/Basic.lean
TacticM = ReaderT Context $ StateRefT State TermElabM
```

But before demonstrating how to use `TacticM`, we shall explore macro-based tactics.

Tactics by Macro Expansion

Just like many other parts of the Lean 4 infrastructure, tactics too can be declared by lightweight macro expansion.

For example, we build an example of a `custom_sorry_macro` that elaborates into a `sorry`. We write this as a macro expansion, which expands the piece of syntax `custom_sorry_macro` into the piece of syntax `sorry`:

```
import Lean.Elab.Tactic

macro "custom_sorry_macro" : tactic => `(tactic| sorry)

example : 1 = 42 := by
  custom_sorry_macro
```

Implementing `trivial`: Extensible Tactics by Macro Expansion

As more complex examples, we can write a tactic such as `custom_tactic`, which is initially completely unimplemented, and can be extended with more tactics. We start by simply declaring the tactic with no implementation:

```
syntax "custom_tactic" : tactic
```

```
example : 42 = 42 := by
  custom_tactic
-- tactic 'tacticCustom_tactic' has not been implemented
  sorry
```

We will now add the `rfl` tactic into `custom_tactic`, which will allow us to prove the previous theorem

```
macro_rules
| `(tactic| custom_tactic) => `(tactic| rfl)

example : 42 = 42 := by
   custom_tactic
-- Goals accomplished 🎉
```

We can now try a harder problem, that cannot be immediately dispatched by `rfl`:

```
example : 43 = 43 ∧ 42 = 42:= by
  custom_tactic
-- tactic 'rfl' failed, equality expected
--   43 = 43 ∧ 42 = 42
-- ⊢ 43 = 43 ∧ 42 = 42
```

We extend the `custom_tactic` tactic with a tactic that tries to break `And` down with `apply And.intro`, and then (recursively (!)) applies `custom_tactic` to the two cases with `(<;> trivial)` to solve the generated subcases `43 = 43`, `42 = 42`.

```
macro_rules
| `(tactic| custom_tactic) => `(tactic| apply And.intro <;> custom_tactic)
```

The above declaration uses `<;>` which is a *tactic combinator*. Here, `a <;> b` means "run tactic `a`, and apply"b" to each goal produced by `a`". Thus, `And.intro <;> custom_tactic` means"run `And.intro`, and then run `custom_tactic` on each goal". We test it out on our previous theorem and see that we dispatch the theorem.

```
example : 43 = 43 ∧ 42 = 42 := by
  custom_tactic
-- Goals accomplished 🎉
```

In summary, we declared an extensible tactic called `custom_tactic`. It initially had no elaboration at all. We added the `rfl` as an elaboration of `custom_tactic`, which allowed it to solve the goal `42 = 42`. We then tried a harder theorem, `43 = 43 ∧ 42 = 42` which `custom_tactic` was unable to solve. We were then able to enrich `custom_tactic` to split "and" with `And.intro`, and also *recursively* call `custom_tactic` in the two subcases.

Implementing <;>: Tactic Combinators by Macro Expansion

Recall that in the previous section, we said that `a <;> b` meant "run `a`, and then run `b` for all goals". In fact, `<;>` itself is a tactic macro. In this section, we will implement the syntax `a and_then b` which will stand for "run `a`, and then run `b` for all goals".

```
-- 1. We declare the syntax `and_then`
syntax tactic " and_then " tactic : tactic

-- 2. We write the expander that expands the tactic
--    into running `a`, and then running `b` on all goals produced by `a`.
macro_rules
| `(tactic| $a:tactic and_then $b:tactic) =>
    `(tactic| $a:tactic; all_goals $b:tactic)

-- 3. We test this tactic.
theorem test_and_then: 1 = 1 ∧ 2 = 2 := by
  apply And.intro and_then rfl

#print test_and_then
-- theorem test_and_then : 1 = 1 ∧ 2 = 2 :=
-- { left := Eq.refl 1, right := Eq.refl 2 }
```

Exploring `TacticM`

The simplest tactic: `sorry`

In this section, we wish to write a tactic that fills the proof with sorry:

```
example : 1 = 2 := by
  custom_sorry
```

We begin by declaring such a tactic:

```
elab "custom_sorry_0" : tactic => do
  return

example : 1 = 2 := by
  custom_sorry_0
-- unsolved goals: ⊢ 1 = 2
```

This defines a syntax extension to Lean, where we are naming the piece of syntax `custom_sorry_0` as living in `tactic` syntax category. This informs the elaborator

that, in the context of elaborating `tactics`, the piece of syntax `custom_sorry_0` must be elaborated as what we write to the right-hand-side of the `=>` (the actual implementation of the tactic).

Next, we write a term in `TacticM Unit` to fill in the goal with `sorryAx α`, which can synthesize an artificial term of type `α`. To do this, we first access the goal with `Lean.Elab.Tactic.getMainGoal : Tactic MVarId`, which returns the main goal, represented as a metavariable. Recall that under types-as-propositions, the type of our goal must be the proposition that `1 = 2`. We check this by printing the type of `goal`.

But first we need to start our tactic with `Lean.Elab.Tactic.withMainContext`, which computes in `TacticM` with an updated context.

```
elab "custom_sorry_1" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goal ← Lean.Elab.Tactic.getMainGoal
    let goalDecl ← goal.getDecl
    let goalType := goalDecl.type
    dbg_trace f!"goal type: {goalType}"

example : 1 = 2 := by
  custom_sorry_1
-- goal type: Eq.{1} Nat (OfNat.ofNat.{0} Nat 1 (instOfNatNat 1)) (OfNat.ofNat.{0} Nat 2
 ↪  (instOfNatNat 2))
-- unsolved goals: ⊢ 1 = 2
```

To `sorry` the goal, we can use the helper `Lean.Elab.admitGoal`:

```
elab "custom_sorry_2" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goal ← Lean.Elab.Tactic.getMainGoal
    Lean.Elab.admitGoal goal

theorem test_custom_sorry : 1 = 2 := by
  custom_sorry_2

#print test_custom_sorry
-- theorem test_custom_sorry : 1 = 2 :=
-- sorryAx (1 = 2) true
```

And we no longer have the error `unsolved goals: ⊢ 1 = 2`.

The `custom_assump` tactic: Accessing Hypotheses

In this section, we will learn how to access the hypotheses to prove a goal. In particular, we shall attempt to implement a tactic `custom_assump`, which looks for an exact match of the goal among the hypotheses, and solves the theorem if possible.

In the example below, we expect `custom_assump` to use `(H2 : 2 = 2)` to solve the goal `(2 = 2)`:

```
theorem assump_correct (H1 : 1 = 1) (H2 : 2 = 2) : 2 = 2 := by
  custom_assump

#print assump_correct
-- theorem assump_correct : 1 = 1 → 2 = 2 → 2 = 2 :=
-- fun H1 H2 => H2
```

When we do not have a matching hypothesis to the goal, we expect the tactic `custom_assump` to throw an error, telling us that we cannot find a hypothesis of the type we are looking for:

```
theorem assump_wrong (H1 : 1 = 1) : 2 = 2 := by
  custom_assump

#print assump_wrong
-- tactic 'custom_assump' failed, unable to find matching hypothesis of type (2 = 2)
-- H1 : 1 = 1
-- ⊢ 2 = 2
```

We begin by accessing the goal and the type of the goal so we know what we are trying to prove. The `goal` variable will soon be used to help us create error messages.

```
elab "custom_assump_0" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goalType ← Lean.Elab.Tactic.getMainTarget
    dbg_trace f!"goal type: {goalType}"

example (H1 : 1 = 1) (H2 : 2 = 2): 2 = 2 := by
  custom_assump_0
-- goal type: Eq.{1} Nat (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)) (OfNat.ofNat.{0} Nat 2
 ↪  (instOfNatNat 2))
-- unsolved goals
-- H1 : 1 = 1
-- H2 : 2 = 2
-- ⊢ 2 = 2
```

```
example (H1 : 1 = 1): 2 = 2 := by
  custom_assump_0
-- goal type: Eq.{1} Nat (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)) (OfNat.ofNat.{0} Nat 2
 ↪  (instOfNatNat 2))
-- unsolved goals
-- H1 : 1 = 1
-- ⊢ 2 = 2
```

Next, we access the list of hypotheses, which are stored in a data structure called `LocalContext`. This is accessed via `Lean.MonadLCtx.getLCtx`. The `LocalContext` contains `LocalDeclarations`, from which we can extract information such as the name that is given to declarations (`.userName`), the expression of the declaration (`.toExpr`). Let's write a tactic called `list_local_decls` that prints the local declarations:

```
elab "list_local_decls_1" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let ctx ← Lean.MonadLCtx.getLCtx -- get the local context.
    ctx.forM fun decl: Lean.LocalDecl => do
      let declExpr := decl.toExpr -- Find the expression of the declaration.
      let declName := decl.userName -- Find the name of the declaration.
      dbg_trace f!"+ local decl: name: {declName} | expr: {declExpr}"

example (H1 : 1 = 1) (H2 : 2 = 2): 1 = 1 := by
  list_local_decls_1
-- + local decl: name: test_list_local_decls_1 | expr: _uniq.3339
-- + local decl: name: H1 | expr: _uniq.3340
-- + local decl: name: H2 | expr: _uniq.3341
  rfl
```

Recall that we are looking for a local declaration that has the same type as the hypothesis. We get the type of `LocalDecl` by calling `Lean.Meta.inferType` on the local declaration's expression.

```
elab "list_local_decls_2" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let ctx ← Lean.MonadLCtx.getLCtx -- get the local context.
    ctx.forM fun decl: Lean.LocalDecl => do
      let declExpr := decl.toExpr -- Find the expression of the declaration.
      let declName := decl.userName -- Find the name of the declaration.
      let declType ← Lean.Meta.inferType declExpr -- **NEW:** Find the type.
      dbg_trace f!"+ local decl: name: {declName} | expr: {declExpr} | type: {declType}"

example (H1 : 1 = 1) (H2 : 2 = 2): 1 = 1 := by
  list_local_decls_2
```

```
  -- + local decl: name: test_list_local_decls_2 | expr: _uniq.4263 | type: (Eq.{1} Nat
   ↪  ...)
  -- + local decl: name: H1 | expr: _uniq.4264 | type: Eq.{1} Nat ...)
  -- + local decl: name: H2 | expr: _uniq.4265 | type: Eq.{1} Nat ...)
  rfl
```

We check if the type of the `LocalDecl` is equal to the goal type with `Lean.Meta.isExprDefEq`. See that we check if the types are equal at `eq?`, and we print that `H1` has the same type as the goal (`local decl[EQUAL? true]: name: H1`), and we print that `H2` does not have the same type (`local decl[EQUAL? false]: name: H2`):

```
elab "list_local_decls_3" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goalType ← Lean.Elab.Tactic.getMainTarget
    let ctx ← Lean.MonadLCtx.getLCtx -- get the local context.
    ctx.forM fun decl: Lean.LocalDecl => do
      let declExpr := decl.toExpr -- Find the expression of the declaration.
      let declName := decl.userName -- Find the name of the declaration.
      let declType ← Lean.Meta.inferType declExpr -- Find the type.
      let eq? ← Lean.Meta.isExprDefEq declType goalType -- **NEW** Check if type equals
       ↪  goal type.
      dbg_trace f!"+ local decl[EQUAL? {eq?}]: name: {declName}"

example (H1 : 1 = 1) (H2 : 2 = 2): 1 = 1 := by
  list_local_decls_3
-- + local decl[EQUAL? false]: name: test_list_local_decls_3
-- + local decl[EQUAL? true]: name: H1
-- + local decl[EQUAL? false]: name: H2
  rfl
```

Finally, we put all of these parts together to write a tactic that loops over all declarations and finds one with the correct type. We loop over declarations with `lctx.findDeclM?`. We infer the type of declarations with `Lean.Meta.inferType`. We check that the declaration has the same type as the goal with `Lean.Meta.isExprDefEq`:

```
elab "custom_assump_1" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goalType ← Lean.Elab.Tactic.getMainTarget
    let lctx ← Lean.MonadLCtx.getLCtx
    -- Iterate over the local declarations...
    let option_matching_expr ← lctx.findDeclM? fun ldecl: Lean.LocalDecl => do
      let declExpr := ldecl.toExpr -- Find the expression of the declaration.
      let declType ← Lean.Meta.inferType declExpr -- Find the type.
      if (← Lean.Meta.isExprDefEq declType goalType) -- Check if type equals goal type.
      then return some declExpr -- If equal, success!
```

```
      else return none          -- Not found.
    dbg_trace f!"matching_expr: {option_matching_expr}"

example (H1 : 1 = 1) (H2 : 2 = 2) : 2 = 2 := by
  custom_assump_1
-- matching_expr: some _uniq.6241
  rfl

example (H1 : 1 = 1) : 2 = 2 := by
  custom_assump_1
-- matching_expr: none
  rfl
```

Now that we are able to find the matching expression, we need to close the theorem by using the match. We do this with `Lean.Elab.Tactic.closeMainGoal`. When we do not have a matching expression, we throw an error with `Lean.Meta.throwTacticEx`, which allows us to report an error corresponding to a given goal. When throwing this error, we format the error using `m!"..."` which builds a `MessageData`. This provides nicer error messages than using `f!"..."` which builds a `Format`. This is because `MessageData` also runs *delaboration*, which allows it to convert raw Lean terms like `(Eq.{1} Nat (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)) (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)))` into readable strings like`(2 = 2)`. The full code listing given below shows how to do this:

```
elab "custom_assump_2" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goal ← Lean.Elab.Tactic.getMainGoal
    let goalType ← Lean.Elab.Tactic.getMainTarget
    let ctx ← Lean.MonadLCtx.getLCtx
    let option_matching_expr ← ctx.findDeclM? fun decl: Lean.LocalDecl => do
      let declExpr := decl.toExpr
      let declType ← Lean.Meta.inferType declExpr
      if ← Lean.Meta.isExprDefEq declType goalType
        then return Option.some declExpr
        else return Option.none
    match option_matching_expr with
    | some e => Lean.Elab.Tactic.closeMainGoal e
    | none =>
      Lean.Meta.throwTacticEx `custom_assump_2 goal
        (m!"unable to find matching hypothesis of type ({goalType})")

example (H1 : 1 = 1) (H2 : 2 = 2) : 2 = 2 := by
  custom_assump_2

example (H1 : 1 = 1): 2 = 2 := by
```

```
  custom_assump_2
-- tactic 'custom_assump_2' failed, unable to find matching hypothesis of type (2 = 2)
-- H1 : 1 = 1
-- ⊢ 2 = 2
```

Tweaking the context

Until now, we've only performed read-like operations with the context. But what if we want to change it? In this section we will see how to change the order of goals and how to add content to it (new hypotheses).

Then, after elaborating our terms, we will need to use the helper function `Lean.Elab.Tactic.liftMetaTactic`, which allows us to run computations in `MetaM` while also giving us the goal `MVarId` for us to play with. In the end of our computation, `liftMetaTactic` expects us to return a `List MVarId` as the resulting list of goals.

The only substantial difference between `custom_let` and `custom_have` is that the former uses `Lean.MVarId.define` and the later uses `Lean.MVarId.assert`:

```
open Lean.Elab.Tactic in
elab "custom_let " n:ident " : " t:term " := " v:term : tactic =>
  withMainContext do
    let t ← elabTerm t none
    let v ← elabTermEnsuringType v t
    liftMetaTactic fun mvarId => do
      let mvarIdNew ← mvarId.define n.getId t v
      let (_, mvarIdNew) ← mvarIdNew.intro1P
      return [mvarIdNew]

open Lean.Elab.Tactic in
elab "custom_have " n:ident " : " t:term " := " v:term : tactic =>
  withMainContext do
    let t ← elabTerm t none
    let v ← elabTermEnsuringType v t
    liftMetaTactic fun mvarId => do
      let mvarIdNew ← mvarId.assert n.getId t v
      let (_, mvarIdNew) ← mvarIdNew.intro1P
      return [mvarIdNew]

theorem test_faq_have : True := by
  custom_let n : Nat := 5
  custom_have h : n = n := rfl
-- n : Nat := 5
-- h : n = n
```

```
-- ⊢ True
  trivial
```

"Getting" and "Setting" the list of goals

To illustrate these, let's build a tactic that can reverse the list of goals. We can use `Lean.Elab.Tactic.getGoals` and `Lean.Elab.Tactic.setGoals`:

```
elab "reverse_goals" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goals : List Lean.MVarId ← Lean.Elab.Tactic.getGoals
    Lean.Elab.Tactic.setGoals goals.reverse

theorem test_reverse_goals : (1 = 2 ∧ 3 = 4) ∧ 5 = 6 := by
  constructor
  constructor
-- case left.left
-- ⊢ 1 = 2
-- case left.right
-- ⊢ 3 = 4
-- case right
-- ⊢ 5 = 6
  reverse_goals
-- case right
-- ⊢ 5 = 6
-- case left.right
-- ⊢ 3 = 4
-- case left.left
-- ⊢ 1 = 2
```

FAQ

In this section, we collect common patterns that are used during writing tactics, to make it easy to find common patterns.

Q: How do I use goals?

A: Goals are represented as metavariables. The module `Lean.Elab.Tactic.Basic` has many functions to add new goals, switch goals, etc.

Q: How do I get the main goal?

A: Use `Lean.Elab.Tactic.getMainGoal`.

```
elab "faq_main_goal" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goal ← Lean.Elab.Tactic.getMainGoal
    dbg_trace f!"goal: {goal.name}"

example : 1 = 1 := by
  faq_main_goal
-- goal: _uniq.9298
  rfl
```

Q: How do I get the list of goals?

A: Use `getGoals`.

```
elab "faq_get_goals" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goals ← Lean.Elab.Tactic.getGoals
    goals.forM $ fun goal => do
      let goalType ← goal.getType
      dbg_trace f!"goal: {goal.name} | type: {goalType}"

example (b : Bool) : b = true := by
  cases b
  faq_get_goals
-- goal: _uniq.10067 | type: Eq.{1} Bool Bool.false Bool.true
-- goal: _uniq.10078 | type: Eq.{1} Bool Bool.true Bool.true
  sorry
  rfl
```

Q: How do I get the current hypotheses for a goal?

A: Use `Lean.MonadLCtx.getLCtx` which provides the local context, and then iterate on the `LocalDeclarations` of the `LocalContext` with accessors such as `foldlM` and `forM`.

```
elab "faq_get_hypotheses" : tactic =>
  Lean.Elab.Tactic.withMainContext do
  let ctx ← Lean.MonadLCtx.getLCtx -- get the local context.
  ctx.forM (fun (decl : Lean.LocalDecl) => do
    let declExpr := decl.toExpr -- Find the expression of the declaration.
    let declType := decl.type -- Find the type of the declaration.
    let declName := decl.userName -- Find the name of the declaration.
    dbg_trace f!" local decl: name: {declName} | expr: {declExpr} | type: {declType}"
  )

example (H1 : 1 = 1) (H2 : 2 = 2): 3 = 3 := by
  faq_get_hypotheses
  -- local decl: name: _example | expr: _uniq.10814 | type: ...
```

```
  -- local decl: name: H1 | expr: _uniq.10815 | type: ...
  -- local decl: name: H2 | expr: _uniq.10816 | type: ...
  rfl
```

Q: How do I evaluate a tactic?

A: Use `Lean.Elab.Tactic.evalTactic: Syntax → TacticM Unit` which evaluates a given tactic syntax. One can create tactic syntax using the macro `` `(tactic| …) ``.

For example, one could call `try rfl` with the piece of code:

```
Lean.Elab.Tactic.evalTactic (← `(tactic| try rfl))
```

Q: How do I check if two expressions are equal?

A: Use `Lean.Meta.isExprDefEq <expr-1> <expr-2>`.

```
#check Lean.Meta.isExprDefEq
-- Lean.Meta.isExprDefEq : Lean.Expr → Lean.Expr → Lean.MetaM Bool
```

Q: How do I throw an error from a tactic?

A: Use `throwTacticEx <tactic-name> <goal-mvar> <error>`.

```
elab "faq_throw_error" : tactic =>
  Lean.Elab.Tactic.withMainContext do
    let goal ← Lean.Elab.Tactic.getMainGoal
    Lean.Meta.throwTacticEx `faq_throw_error goal "throwing an error at the current
     ↪  goal"

example (b : Bool): b = true := by
  cases b;
  faq_throw_error
  -- case true
  -- ⊢ true = true
  -- tactic 'faq_throw_error' failed, throwing an error at the current goal
  -- case false
  -- ⊢ false = true
```

Q: What is the difference between `Lean.Elab.Tactic.*` and `Lean.Meta.Tactic.*`?

A: `Lean.Meta.Tactic.*` contains low level code that uses the `Meta` monad to implement basic features such as rewriting. `Lean.Elab.Tactic.*` contains high-level code that connects the low level development in `Lean.Meta` to the tactic infrastructure and the parsing front-end.

Lean4 Cheat-sheet

Extracting information

- Extract the goal: `Lean.Elab.Tactic.getMainGoal`

  Use as `let goal ← Lean.Elab.Tactic.getMainGoal`
- Extract the declaration out of a metavariable: `mvarId.getDecl` when `mvarId : Lean.MVarId` is in context. For instance, `mvarId` could be the goal extracted using `getMainGoal`
- Extract the type of a metavariable: `mvarId.getType` when `mvarId : Lean.MVarId` is in context.
- Extract the type of the main goal: `Lean.Elab.Tactic.getMainTarget`

  Use as `let goal_type ← Lean.Elab.Tactic.getMainTarget`

  Achieves the same as

  ```
  let goal ← Lean.Elab.Tactic.getMainGoal
  let goal_type ← goal.getType
  ```
- Extract local context: `Lean.MonadLCtx.getLCtx`

  Use as `let lctx ← Lean.MonadLCtx.getLCtx`
- Extract the name of a declaration: `Lean.LocalDecl.userName ldecl` when `ldecl : Lean.LocalDecl` is in context
- Extract the type of an expression: `Lean.Meta.inferType expr` when `expr : Lean.Expr` is an expression in context

  Use as `let expr_type ← Lean.Meta.inferType expr`

Playing around with expressions

- Convert a declaration into an expression: `Lean.LocalDecl.toExpr`

Use as `ldecl.toExpr`, when `ldecl : Lean.LocalDecl` is in context

For instance, `ldecl` could be `let ldecl ← Lean.MonadLCtx.getLCtx`

- Check whether two expressions are definitionally equal: `Lean.Meta.isDefEq ex1 ex2` when `ex1 ex2 : Lean.Expr` are in context. Returns a `Lean.MetaM Bool`
- Close a goal: `Lean.Elab.Tactic.closeMainGoal expr` when `expr : Lean.Expr` is in context

Further commands

- meta-sorry: `Lean.Elab.admitGoal goal`, when `goal : Lean.MVarId` is the current goal

Printing and errors

- Print a "permanent" message in normal usage:

  `Lean.logInfo f!"Hi, I will print\n{Syntax}"`

- Print a message while debugging:

  `dbg_trace f!"1) goal: {Syntax_that_will_be_interpreted}"`.

- Throw an error: `Lean.Meta.throwTacticEx name mvar message_data` where `name : Lean.Name` is the name of a tactic and `mvar` contains error data.

  Use as `Lean.Meta.throwTacticEx`tac goal (m¡'unable to find matching hypothesis of type ({goal_type})")`where them!formatting builds a`MessageData' for better printing of terms

TODO: Add? * Lean.LocalContext.forM * Lean.LocalContext.findDeclM?

Extra: Options

Options are a way to communicate some special configuration to both your meta programs and the Lean compiler itself. Basically it's just a `KVMap` which is a simple map from `Name` to a `Lean.DataValue`. Right now there are 6 kinds of data values: - `String` - `Bool` - `Name` - `Nat` - `Int` - `Syntax`

Setting an option to tell the Lean compiler to do something different with your program is quite simple with the `set_option` command:

```
import Lean
open Lean

#check 1 + 1 -- 1 + 1 : Nat

set_option pp.explicit true -- No custom syntax in pretty printing

#check 1 + 1 -- @HAdd.hAdd Nat Nat Nat (@instHAdd Nat instAddNat) 1 1 : Nat

set_option pp.explicit false
```

You can furthermore limit an option value to just the next command or term:

```
set_option pp.explicit true in
#check 1 + 1 -- @HAdd.hAdd Nat Nat Nat (@instHAdd Nat instAddNat) 1 1 : Nat

#check 1 + 1 -- 1 + 1 : Nat

#check set_option trace.Meta.synthInstance true in 1 + 1 -- the trace of the type class
 ↪  synthesis for `OfNat` and `HAdd`
```

If you want to know which options are available out of the Box right now you can simply write out the `set_option` command and move your cursor to where the name is written, it should give you a list of them as auto completion suggestions. The most useful group of options when you are debugging some meta thing is the `trace.` one.

Options in meta programming

Now that we know how to set options, let's take a look at how a meta program can get access to them. The most common way to do this is via the `MonadOptions` type class, an extension to `Monad` that provides a function `getOptions : m Options`. As of now, it is implemented by: - `CoreM` - `CommandElabM` - `LevelElabM` - all monads to which you can lift operations of one of the above (e.g. `MetaM` from `CoreM`)

Once we have an `Options` object, we can query the information via `Options.get`. To show this, let's write a command that prints the value of `pp.explicit`.

```
elab "#getPPExplicit" : command => do
  let opts ← getOptions
  -- defValue = default value
  logInfo s!"pp.explicit : {opts.get pp.explicit.name pp.explicit.defValue}"

#getPPExplicit -- pp.explicit : false

set_option pp.explicit true in
#getPPExplicit -- pp.explicit : true
```

Note that the real implementation of getting `pp.explicit`, `Lean.getPPExplicit`, uses whether `pp.all` is set as a default value instead.

Making our own

Declaring our own option is quite easy as well. The Lean compiler provides a macro `register_option` for this. Let's see it in action:

```
register_option book.myGreeting : String := {
  defValue := "Hello World"
  group := "pp"
  descr := "just a friendly greeting"
}
```

However, we cannot just use an option that we just declared in the same file it was declared in because of initialization restrictions.

Pretty Printing

The pretty printer is what Lean uses to present terms that have been elaborated to the user. This is done by converting the `Expr`s back into `Syntax` and then even higher level pretty printing datastructures. This means Lean does not actually recall the `Syntax` it used to create some `Expr`: there has to be code that tells it how to do that. In the big picture, the pretty printer consists of three parts run in the order they are listed in: - the delaborator this will be our main interest since we can easily extend it with our own code. Its job is to turn `Expr` back into `Syntax`. - the parenthesizer responsible for adding parenthesis into the `Syntax` tree, where it thinks they would be useful - the formatter responsible for turning the parenthesized `Syntax` tree into a `Format` object that contains more pretty printing information like explicit whitespaces

Delaboration

As its name suggests, the delaborator is in a sense the opposite of the elaborator. The job of the elaborator is to take an `Expr` produced by the elaborator and turn it back into a `Syntax` which, if elaborated, should produce an `Expr` that behaves equally to the input one.

Delaborators have the type `Lean.PrettyPrinter.Delaborator.Delab`. This is an alias for `DelabM Syntax`, where `DelabM` is the delaboration monad. All of this machinery is defined here. `DelabM` provides us with quite a lot of options you can look up in the documentation (TODO: Docs link). We will merely highlight the most relevant parts here. - It has a `MonadQuotation` instance which allows us to declare `Syntax` objects using the familiar quotation syntax. - It can run `MetaM` code. - It has a `MonadExcept` instance for throwing errors. - It can interact with `pp` options using functions like `whenPPOption`. - You can obtain the current subexpression using `SubExpr.getExpr`. There is also an entire API defined around this concept in the `SubExpr` module.

Making our own

Like so many things in metaprogramming the elaborator is based on an attribute, in this case the `delab` one. `delab` expects a `Name` as an argument, this name has to start with the name of an `Expr` constructor, most commonly `const` or `app`. This constructor name is then followed by the name of the constant we want to delaborate. For example, if we want to delaborate a function `foo` in a special way we would use `app.foo`. Let's see this in action:

```
import Lean

open Lean PrettyPrinter Delaborator SubExpr

def foo : Nat → Nat := fun x => 42

@[delab app.foo]
def delabFoo : Delab := do
  `(1)

#check foo -- 1 : Nat → Nat
#check foo 13 -- 1 : Nat, full applications are also pretty printed this way
```

This is obviously not a good delaborator since reelaborating this `Syntax` will not yield the same `Expr`. Like with many other metaprogramming attributes we can also overload delaborators:

```
@[delab app.foo]
def delabfoo2 : Delab := do
  `(2)

#check foo -- 2 : Nat → Nat
```

The mechanism for figuring out which one to use is the same as well. The delaborators are tried in order, in reverse order of registering, until one does not throw an error, indicating that it "feels unresponsible for the `Expr`". In the case of delaborators, this is done using `failure`:

```
@[delab app.foo]
def delabfoo3 : Delab := do
  failure
  `(3)

#check foo -- 2 : Nat → Nat, still 2 since 3 failed
```

In order to write a proper delaborator for `foo`, we will have to use some slightly more advanced machinery though:

```
@[delab app.foo]
def delabfooFinal : Delab := do
  let e ← getExpr
  guard $ e.isAppOfArity' `foo 1 -- only delab full applications this way
  let fn := mkIdent `fooSpecial
  let arg ← withAppArg delab
  `($fn $arg)

#check foo 42 -- fooSpecial 42 : Nat
#check foo -- 2 : Nat → Nat, still 2 since 3 failed
```

Can you extend `delabFooFinal` to also account for non full applications?

Unexpanders

While delaborators are obviously quite powerful it is quite often not necessary to use them. If you look in the Lean compiler for `@[delab` or rather `@[builtin_delab` (a special version of the `delab` attribute for compiler use, we don't care about it), you will see there are quite few occurences of it. This is because the majority of pretty printing is in fact done by so called unexpanders. Unlike delaborators they are of type `Lean.PrettyPrinter.Unexpander` which in turn is an alias for `Syntax → Lean.PrettyPrinter.UnexpandM Syntax`. As you can see, they are `Syntax` to `Syntax` translations, quite similar to macros, except that they are supposed to be the inverse of macros. The `UnexpandM` monad is quite a lot weaker than `DelabM` but it still has: - `MonadQuotation` for syntax quotations - The ability to throw errors, although not very informative ones: `throw ()` is the only valid one

Unexpanders are always specific to applications of one constant. They are registered using the `app_unexpander` attribute, followed by the name of said constant. The unexpander is passed the entire application of the constant after the `Expr` has been delaborated, without implicit arguments. Let's see this in action:

```
def myid {α : Type} (x : α) := x

@[app_unexpander myid]
def unexpMyId : Unexpander
  -- hygiene disabled so we can actually return `id` without macro scopes etc.
  | `(myid $arg) => set_option hygiene false in `(id $arg)
  | `(myid) => pure $ mkIdent `id
  | _ => throw ()
```

```
#check myid 12 -- id 12 : Nat
#check myid -- id : ?m.3870 → ?m.3870
```

For a few nice examples of unexpanders you can take a look at NotationExtra

Mini project

As per usual, we will tackle a little mini project at the end of the chapter. This time we build our own unexpander for a mini programming language. Note that many ways to define syntax already have generation of the required pretty printer code built-in, e.g. `infix`, and `notation` (however not `macro_rules`). So, for easy syntax, you will never have to do this yourself.

```
declare_syntax_cat lang
syntax num   : lang
syntax ident : lang
syntax "let " ident " := " lang " in " lang: lang
syntax "[Lang| " lang "]" : term

inductive LangExpr
  | numConst : Nat → LangExpr
  | ident    : String → LangExpr
  | letE     : String → LangExpr → LangExpr → LangExpr

macro_rules
  | `([Lang| $x:num ]) => `(LangExpr.numConst $x)
  | `([Lang| $x:ident]) => `(LangExpr.ident $(Lean.quote (toString x.getId)))
  | `([Lang| let $x:ident := $v:lang in $b:lang]) => `(LangExpr.letE $(Lean.quote
   ↪  (toString x.getId)) [Lang| $v] [Lang| $b])

instance : Coe NumLit (TSyntax `lang) where
  coe s := ⟨s.raw⟩

instance : Coe Ident (TSyntax `lang) where
  coe s := ⟨s.raw⟩

-- LangExpr.letE "foo" (LangExpr.numConst 12)
--   (LangExpr.letE "bar" (LangExpr.ident "foo") (LangExpr.ident "foo")) : LangExpr
#check [Lang|
  let foo := 12 in
  let bar := foo in
  foo
]
```

As you can see, the pretty printing output right now is rather ugly to look at. We

can do better with an unexpander:

```
@[app_unexpander LangExpr.numConst]
def unexpandNumConst : Unexpander
  | `(LangExpr.numConst $x:num) => `([Lang| $x])
  | _ => throw ()

@[app_unexpander LangExpr.ident]
def unexpandIdent : Unexpander
  | `(LangExpr.ident $x:str) =>
    let str := x.getString
    let name := mkIdent $ Name.mkSimple str
    `([Lang| $name])
  | _ => throw ()

@[app_unexpander LangExpr.letE]
def unexpandLet : Unexpander
  | `(LangExpr.letE $x:str [Lang| $v:lang] [Lang| $b:lang]) =>
    let str := x.getString
    let name := mkIdent $ Name.mkSimple str
    `([Lang| let $name := $v in $b])
  | _ => throw ()

-- [Lang| let foo := 12 in foo] : LangExpr
#check [Lang|
  let foo := 12 in foo
]

-- [Lang| let foo := 12 in let bar := foo in foo] : LangExpr
#check [Lang|
  let foo := 12 in
  let bar := foo in
  foo
]
```

That's much better! As always, we encourage you to extend the language yourself with things like parenthesized expressions, more data values, quotations for `term` or whatever else comes to your mind.