Metaprogramming in Lean 4

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

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.

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, the examples provided will be simple enough for you to follow and comprehend without a super deep understanding of monadic programming.

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.

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:

```
import Lean

elab "#assertType " termStx:term " : " typeStx:term : command =>
  open Lean.Elab Command Term in
  liftTermElabM `assertTypeCmd
    try
      let tp ← elabType typeStx
      let tm ← 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 `tm : Expr`, which should have the type `tp` otherwise an error will be thrown.

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, 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 writing `syntax:75` for multiplication 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 arith " + " arith      : arith -- Arith.add
syntax:75 arith " * " arith : 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.symbol "x") (Arith.symbol "y")

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

#check ⟪ "x" + 20 ⟫
-- Arith.add (Arith.symbol "x") (Arith.int 20)

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

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

#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 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
  withMVarContext mvarId do
    let t ← elabType t
    let p ← mkFreshExprMVar t MetavarKind.syntheticOpaque n
    let (_, mvarIdNew) ← intro1P $ ← assert mvarId 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.Elab.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 → Data → Expr                            -- bound variables
  | fvar    : FVarId → Data → Expr                         -- free variables
  | mvar    : MVarId → Data → Expr                         -- meta variables
  | sort    : Level → Data → Expr                          -- Sort
  | const   : Name → List Level → Data → Expr              -- constants
  | app     : Expr → Expr → Data → Expr                    -- application
  | lam     : Name → Expr → Expr → Data → Expr             -- lambda abstraction
  | forallE : Name → Expr → Expr → Data → Expr             -- (dependent) arrow
  | letE    : Name → Expr → Expr → Expr → Data → Expr -- let expressions
  -- less essential constructors:
  | lit     : Literal → Data → Expr                        -- literals
  | mdata   : MData → Expr → Data → Expr                   -- metadata
  | proj    : Name → Nat → Expr → Data → 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 ahead. 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) → $b`). This is also sometimes called a Π-type or Π-expression. Note that the non-dependent arrow `α → β` is a special case of `(a : α) → β` where `β` doesn't depend on `a`. The `E` on the end of `forallE` is to distinguish it from the `forall` keyword.
- `letE n t v b` is a **let binder** (`let ($n : $t) := $v in $b`).
- `lit` is a **literal**, this is a number or string literal like `4` or `"hello world"`. These are not strictly necessary for the kernel, but they are kept mainly for convenience. (Ie in Lean 3, there were a load of tricks needed to store `11234 : Nat` as something more efficient than `succ $ succ $ succ ... $ succ 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).

Expression Data

If you look the constructors of `Expr`, you will see that all of them have a `Data` argument. This Data field contains some extra cached information about the expression that is useful for speeding up some common operations. These are things like: a hash of the `Expr`, whether or not the `Expr` contains free variables, metavariables or bound variables and also it is where the `BinderInfo` is stored for `forallE` and `lam`.

This data param means that you should *never* construct instances of `Expr` directly using the `Expr` constructors but instead use the helper methods (`mkLambda`, `mkApp` etc) that compute `Data` for you.

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, `Expr`s 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 `Expr`s containing bound expressions are equal.

This is why the signature of the `bvar` case is `Nat → Expr` and not `Name → Expr`. If in our `Expr`, all `bvar`s are bound, we say that the `Expr` is **closed**. The process of replacing all instances of an unbound `bvar` with an `Expr` is called **instantiation**. Going the other way is called **abstraction**.

Constructing Expressions

As mentioned above, you should *never* construct instances of `Expr` directly using the `Expr` constructors but instead use helper methods that not only compute `Data` but also take care of other things for you. Here we give examples and brief descriptions of some basic helpers.

The simplest expressions we can construct are constants. We use `mkConst` with argument a name. Below are two examples of this, both giving an expression for the natural number `0`.

The second form (with double backticks) is better, as it resolves the name to a global name, checking, in the process that it is valid.

```
open Lean

def z' := mkConst `Nat.zero
#eval z' -- Lean.Expr.const `Nat.zero [] (Expr.mkData 3114957063 (bi := Lean.BinderI

def z := mkConst ``Nat.zero
#eval z -- Lean.Expr.const `Nat.zero [] (Expr.mkData 3114957063 (bi := Lean.BinderIn
```

To illustrate the difference, here are two further examples. The first definition is unsafe as it is not valid without `open Nat` in the context. On the other hand, the second resolves correctly.

```
open Nat

def z₁ := mkConst `zero
#eval z₁ -- Lean.Expr.const `zero [] (Expr.mkData 790182631 (bi := Lean.BinderInfo.d

def z₂ := mkConst ``zero
#eval z₂ -- Lean.Expr.const `Nat.zero [] (Expr.mkData 3114957063 (bi := Lean.BinderI
```

The next class of expressions we consider are function applications. These can be built using `mkApp` 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 := mkApp (mkConst ``Nat.succ) z
#eval one
-- Lean.Expr.app
-- (Lean.Expr.const `Nat.succ [] (Expr.mkData 3403344051 (bi := Lean.BinderInfo.defa
-- (Lean.Expr.const `Nat.zero [] (Expr.mkData 3114957063 (bi := Lean.BinderInfo.defa
-- (Expr.mkData 3354277877 (approxDepth := 1) (bi := Lean.BinderInfo.default))
```

```
def natExpr: Nat → Expr
| 0     => z
| n + 1 => mkApp (mkConst ``Nat.succ) (natExpr n)
```

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

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

We next consider the helper `mkLambda` to construct a simple function named `cz` which takes any natural number and returns `Nat.zero`. The argument `BinderInfo.default` for the constructor says that the argument is explicit.

```
def constZero : Expr :=
  mkLambda `cz BinderInfo.default (mkConst ``Nat) (mkConst ``Nat.zero)
```

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

In the next chapter we shall explore some functions that compute in the `MetaM` monad, opening room for more powerful tricks involving expressions. And we will start off by visiting `reduce`, a function that can simplify bigger expressions such as the ones we'd get from using the functions above.

MetaM

Smart constructors for expressions

The `MetaM` monad provides even more smart constructors to help us build expressions. The API also contains functions that help us explore certain expressions more easily. In this chapter we will visit some of those.

But first, let's recap the definition of `natExpr`

```
import Lean

open Lean Meta

def natExpr : Nat → Expr
  | 0     => mkConst ``Nat.zero
  | n + 1 => mkApp (mkConst ``Nat.succ) (natExpr n)

#eval natExpr 1
-- Lean.Expr.app
--   (Lean.Expr.const `Nat.succ [] (Expr.mkData 3403344051 (bi := Lean.BinderInfo.de
--   (Lean.Expr.const `Nat.zero [] (Expr.mkData 3114957063 (bi := Lean.BinderInfo.de
--   (Expr.mkData 3354277877 (approxDepth := 1) (bi := Lean.BinderInfo.default))
```

That's already a long expression for the natural number 1! Let's see what `reduce : Expr → MetaM Expr` can do about it

```
#eval reduce $ natExpr 1
-- Lean.Expr.lit (Lean.Literal.natVal 1) (Expr.mkData 4289331193 (bi := Lean.BinderI
```

The following example would yield an even longer expression, but `reduce` can clean it up for us:

```
def sumExprM (n m : Nat) : MetaM Expr := do
  reduce $ mkAppN (mkConst ``Nat.add) #[natExpr n, natExpr m]
```

```
#eval sumExprM 2 3 --Lean.Expr.lit (Lean.Literal.natVal 5) (Expr.mkData 1441793 (bi
```

We next construct a λ-expression for the function `double : Nat → Nat` given by `double n = n + n`. To construct such an expression, we introduce a free variable `n`, we define an expression in terms of this variable, and we construct the λ-expression.

The variable is introduced by passing the code using it as a *continuation* to `withLocalDecl`. The arguments of `withLocalDecl` are: * The name of the variable * The *binder* that determines whether it is explicit or not * The type of the variable * The function that turns an expression into something else (in our case, into another expression). This function does not need to be pure.

The λ-expression is constructed using `mkLambdaFVars`, with the first argument being an array of free variables (just one in this case) with respect to which we take λ. The second argument is the body of the λ-expression.

```
def doubleM : MetaM Expr :=
  withLocalDecl `n BinderInfo.default (mkConst ``Nat)
    fun n : Expr => mkLambdaFVars #[n] $ mkAppN (mkConst ``Nat.add) #[n, n]
```

Let's check if `doubleM` can indeed compute the expression for `n + n`

```
def appDoubleM (n : Nat) : MetaM Expr := do
  reduce $ mkApp (← doubleM) (natExpr n)

#eval appDoubleM 3 -- Lean.Expr.lit (Lean.Literal.natVal 6) (Expr.mkData 393219 (bi
```

A powerful feature of lean is its unifier. There is an easy way to use this while meta-programming, namely the method `mkAppM` (and a similar method `mkAppM'`). For example, we can construct an expression for the length of a list using `mkAppM`. Recall that `List.length` has an implicit parameter `α : Type u`. This is deduced by unification, as are universe levels.

```
def lenExprM (list: Expr) : MetaM Expr := do
  reduce $ ← mkAppM ``List.length #[list]
```

We test the unification in this definition.

```
def egList := [1, 3, 7, 8]

def egLenM : MetaM Expr :=
  lenExprM (mkConst ``egList)
```

```
#eval egLenM -- Lean.Expr.lit (Lean.Literal.natVal 4) (Expr.mkData 2490367 (bi := Le
```

Analogous to the construction of λ-expressions, we can construct ∀-expressions (i.e., Π-expressions) for types. We simply replace `mkLambdaFVars` with `mkForallF-Vars`.

A special case of Π-types are function types `A → B`. These can be constructed using the function `mkArrow`. Another very useful meta-level function is `mkEq`, which constructs equalities.

We illustrate all these, as well as the construction of a λ-expression, by constructing the proposition `∀ n: Nat, f n = f (n + 1)` as a function of `f`. Formally this is `λ f, ∀ n, f n = f (n + 1)`. We break this into many steps to illustrate the different ingredients.

First we build the expression for our proposition:

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

Now let's elaborate the expression into a term so we can see the result of what we did more easily. This will be further explored in the next chapter

```
elab "myProp" : term => propM

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

Meta variables

Meta-variables are variables that can be created and assigned to only at the meta level, and not the object/term level. They are used principally as placeholders, especially for goals. They can be assigned expressions in terms of other meta variables. However, before being assigned to a pure (i.e., not meta) definition, the assignments should be resolvable to a value not involving meta-variables.

One way to create a meta-variable representing an expression is to use the `mkFreshExprMVar` function. This function creates a meta-variable that can be assigned an expression. One can optionally specify a type for the meta-variable. In the example below, we create three meta-variables, `mvar1`, `mvar2`, and `mvar3`, with `mvar1` and `mvar3` assigned type `Nat` and `mvar2` assigned the type `Nat → Nat`.

We assign expressions to the meta-variables using the `assignExprMVar` function. Like many functions dealing with meta-variables, this takes the id of the meta-variable as an argument. Below we assign to `mvar1` the result of function application of `mvar2` to `mvar3`. We then assign to `mvar2` the constant expression `Nat.succ` and to `mvar3` the constant expression `Nat.zero`. Clearly this means we have assigned `Nat.succ (Nat.zero)`, i.e., `1` to `mvar1`. We return `mvar1` in the function `metaOneM`. We can see, using an elaborator, that indeed when the expression `metaOneM` is assigned to a term, the result is `1`.

```
def oneMetaVar : MetaM Expr := do
  let zero := mkConst ``Nat.zero
  let mvar1 ← mkFreshExprMVar (some (mkConst ``Nat))
  let mvar2 ← mkFreshExprMVar (some (mkConst ``Nat))
  let funcType ← mkArrow (mkConst ``Nat) (mkConst ``Nat)
  let mvar3 ← mkFreshExprMVar (some funcType)
  IO.println "the initial state of each metavariable:"
  IO.println $ ← instantiateMVars mvar1
  IO.println $ ← instantiateMVars mvar2
  IO.println $ ← instantiateMVars mvar3
  IO.println "---------------------------------"
  assignExprMVar mvar1.mvarId! (mkApp mvar3 mvar2)
  IO.println "after turning `mvar1` into an application:"
  IO.println $ ← instantiateMVars mvar1
  IO.println $ ← instantiateMVars mvar2
  IO.println $ ← instantiateMVars mvar3
  IO.println "---------------------------------"
  assignExprMVar mvar2.mvarId! zero
  IO.println "after turning the application argument into `Nat.zero`:"
  IO.println $ ← instantiateMVars mvar1
  IO.println $ ← instantiateMVars mvar2
  IO.println $ ← instantiateMVars mvar3
  IO.println "--------------------------------"
  assignExprMVar mvar3.mvarId! (mkConst ``Nat.succ)
  IO.println "after turning the application function into `Nat.succ`:"
  IO.println $ ← instantiateMVars mvar1
  IO.println $ ← instantiateMVars mvar2
  IO.println $ ← instantiateMVars mvar3
  IO.println "--------------------------------"
  return mvar1

elab "one!" : term => oneMetaVar

#eval one! -- 1
-- the initial state of each metavariable:
-- ?_uniq.3411
-- ?_uniq.3412
-- ?_uniq.3413
-- --------------------------------
-- after turning `mvar1` into an application:
-- ?_uniq.3413 ?_uniq.3412
-- ?_uniq.3412
-- ?_uniq.3413
-- --------------------------------
-- after turning the application argument into `Nat.zero`:
-- ?_uniq.3413 Nat.zero
-- Nat.zero
-- ?_uniq.3413
-- --------------------------------
-- after turning the application function into `Nat.succ`:
-- Nat.succ Nat.zero
-- Nat.zero
-- Nat.succ
-- --------------------------------
```

Telescopes

Before going further, let's take a step back and think about the `Expr.lam` constructor:

```
Expr.lam : Name → Expr → Expr → Data → Expr
```

The first `Expr` is the type of the function's input and the second is its body. Then we ask ourselves: how do we build a function with multiple input variables? Well, we use the same constructor multiple times, one for each input variable.

As an example, let's see an approximation of how we'd build the function `fun (x : Nat) (y : Nat) => x + y`:

```
Expr.lam `x (mkConst ``Nat) (Expr.lam `y (mkConst ``Nat) b) d') d
```

It's done by nesting a new `Expr.lam` as the body of another `Expr.lam`. Thus, if we wanted to, say, perform a computation that involves all the input types of a function as well as its body, we would have to unfold the expression recursively until the last nested `Expr.lam` just to gather everything we need to do what we want. And that's when `lambdaTelescope` comes into play.

```
def lambdaTelescope (e : Expr) (k : Array Expr → Expr → m α) : m α
```

It makes it easier for us to do our computation with the data that we need. All we need to do is provide a `k` function, whose first argument is an array of input types and the second argument is the function body.

There are multiple telescopes in the API and we don't intend to be exhaustive here. Something to note is that `m` is not necessarily the `MetaM` monad, but we are covering this subject here because telescopes are defined in `Lean.Meta` and also because we are already in `MetaM` when we want to use more powerful tools to deal with expressions.

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 three unintuitive parts about these two are: - The fact that we are leaving spaces around our operators: " ⊕ "," XOR ". 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 you 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. This forces left associativity like `infixl` above. To understand this, let's compare two hypothetical parses:

```
-- a LXOR b LXOR c
(a:10 LXOR b:11):10 LXOR c
a LXOR (b:10 LXOR c:11):10
```

In the parse tree of `(a:10 LXOR b:11):10 LXOR c`, we see that the right argument `(b LXOR c)` is given the precedence 10, because a rule is always given the lowest precedence of any of its subrules. However, the rule for `LXOR` expects the right argument to have a precedence of at least 11, as witnessed by the `r:11` at the right-hand-side of `notation:10 l:10 " LXOR " r:11`. Thus this rule ensures that `LXOR` is left associative.

Can you make it right associative?

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, is topic of the elaboration and macro chapter. An example of one we have already seen however would be the `notation` and `infix` command.

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 pa

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 boolean_expr " OR " boolean_expr : boolean_expr
syntax 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 p
```

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 - repetetive 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 a way to do all of these.

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

```
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. 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, arbitrarily 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 repetetion 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 se
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 parsin
#check bin() -- fails to parse because `binNumber` is "one or many": expected 'O' or

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
```

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

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

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 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 is. 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, they 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 ` infront of the name
#eval Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumLit "1"] --
#eval mkNode `«term_+_» #[Syntax.mkNumLit "1", Syntax.mkNumLit "1"] -- is the syntax

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

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

Note that x and y in this example are of type Syntax not Nat. This is simply because we are still at the Syntax level: the concept of a type doesn't quite exist yet. What we can however do is limit the parsers/categories we want to match on, for example if we only want to match on number literals in order to implement some constant folding:

```
def isLitAdd : Syntax → Option Nat
  | `(Nat.add $x:num $y:num) => some (x.toNat + y.toNat)
  | _ => none

#eval isLitAdd (Syntax.mkApp (mkIdent `Nat.add) #[Syntax.mkNumLit "1", Syntax.mkNumL
#eval isLitAdd (Syntax.mkApp (mkIdent `Nat.add) #[mkIdent `foo, Syntax.mkNumLit "1"]
```

As you can see in the code even though we explicitly matched on the `num` parser we still have to explicitly convert x and y to Nat because again, we are on Syntax level, types do not exist.

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 : Syntax → Nat
  | `(arith| $x:num) => x.toNat
  | `(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.

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 quoting mechanism to c
  | _ => 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 
```

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

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.

MonadQuotation and MonadRef

This macro hygiene mechanism is 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: someone has to keep track of macro scopes for us. In this case, this is done by the MacroM monad, but it can be done by any monad that implements Lean.MonadQuotation. For this reason, it's worth to take a brief look at it:

```
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 and - evaluates a function of type Syntax → m α to m α by the return value of getRef to this Syntax parameter and evaluating the m α parameter with that new state.

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 mac
  | `([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.

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 of 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 `Expr`s 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 `commandElabAttribute`. 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 `commandElabAttribute`

Let’s see how this is done:

```
import Lean

open Lean Elab Command Term Meta

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

@[commandElab mycommand1]
def mycommand1Impl : CommandElab := fun stx => do -- declare and register the elabor
  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.

```
@[commandElab 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`

```
@[commandElab Lean.Parser.Command.check] def mySpecialCheck : CommandElab := fun stx
  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.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.dec

#findCElab def lala := 12 -- Your syntax may be elaborated by: [Lean.Elab.Command.el
#findCElab abbrev lolo := 12 -- Your syntax may be elaborated by: [Lean.Elab.Command
#findCElab #check foo -- even our own syntax!: Your syntax may be elaborated by: [my
#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.ela
#findCElab #findCElab open Bar -- even itself!: Your syntax may be elaborated by: [«
```

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 `Syn-`

tax via the termElabAttribute (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 meta variables. As you know from before, metavariables are holes in Exprs that are waiting to be filled in. Synthetic meta variables 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 meta variable should be solved with typeclass synthesis - coe, the meta variable should be solved via coercion (a special case of typeclass) - tactic, the meta variable 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 meta variable is created, the next higher level term elaborator will continue. At some point, execution of postponed meta variables 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.pos
```

What happened here is that the elaborator for function applications started at List.foldr which is a generic function so it created meta variables 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 meta variable) the elaborator for .add does need to know the actual value of ?m2 so the term elaborator postpones execution (by internally creating a synthetic meta variable 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 ...
  -- ?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]

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

#eval myterm 1 -- 1

-- Also works with `elab`
elab "myterm 2" : term => do
  mkAppM ``List.get! #[mkConst ``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 []

@[termElab myanon]
def myanonImpl : TermElab := fun stx typ? => do
  -- Attempt to postpone execution if the type is not known or is a meta variable.
  -- Meta variables 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
  let [ctor] ← getCtors base | throwError "type doesn't have exactly one constructor
  let args := 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.

Let's begin with the usual incantations, where we import `Lean` and open `Lean`, `Lean.Elab`, and `Lean.Meta`.

```lean
import Lean

open Lean Elab Meta
```

We begin by defining our atomic literal value.

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

This is our only unary operator

```lean
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
```

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.toNat]
  -- `mkConst` creates an `Expr.const` given the constant `Name`
  | `(imp_lit| true   ) => mkAppM ``IMPLit.bool #[mkConst ``Bool.true]
  | `(imp_lit| false  ) => mkAppM ``IMPLit.bool #[mkConst ``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
```

Now we can elaborate our (only) unary operator

```
declare_syntax_cat imp_unop
syntax "not"     : imp_unop

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

And our binary operators:

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

The following function could very well be pure (`Syntax → Expr`), but we're staying in `MetaM` because it allows us to easily throw an error for match completion.

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

The operators are needed for our expressions. See below:

```
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 t
```

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 mkConst ``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) => d
    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 (I
--       (IMPProgram.Assign "c" (IMPExpr.lit (IMPLit.nat 5)))
--       (IMPProgram.Assign "a" (IMPExpr.bin IMPBinOp.add (IMPExpr.var "a") (IMPExpr
--     (IMPProgram.Assign "b" (IMPExpr.lit (IMPLit.nat 10))))
```

Tactics

We've finally come to what may be considered by many the end goal of this book. The reason why this chapter is placed after the DSL chapter is because the tactic mode in Lean 4 is itself a DSL.

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

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

We will start by implementing tactics that compute in `TacticM` and then we shall see how some tactics can be implemented as macros.

The simplest tactic: `sorry`

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

```
theorem wrong : 1 = 2 := by
  custom_sorry

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

We begin by declaring such a tactic:

```
import Lean.Elab.Tactic

elab "custom_sorry_0" : tactic => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  dbg_trace f!"1) goal: {goal.name}"
```

```
theorem wrong : 1 = 2 := by
  custom_sorry_0
-- 1) goal: _uniq.461
-- unsolved goals: ⊢ 1 = 2
```

This defines a syntax extension to Lean, where we are naming the piece of syntax `admit` as living in `tactic` syntax category. This informs the elaborator that in the context of elaborating `tactic`s, the piece of syntax `admit` must be elaborated as what we write to the right-hand-side of the `=>` (we fill the `...` with the body of the tactic).

Next, we write a term in `TacticM Unit` which fills in the goal with a `sorryAx _`. To do this, we first access the goal, and then we fill the goal in with a `sorryAx`. We 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`.

```
elab "custom_sorry_1" : tactic => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  dbg_trace f!"1) goal: {goal.name}"
  let goal_declaration ← Lean.Meta.getMVarDecl goal
  let goal_type := goal_declaration.type
  dbg_trace f!"2) goal type: {goal_type}"

theorem wrong_1 : 1 = 2 := by
  custom_sorry_1
-- 1) goal: _uniq.757
-- 2) 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 => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  let goal_declaration ← Lean.Meta.getMVarDecl goal
  let goal_type := goal_declaration.type
  Lean.Elab.admitGoal goal

theorem wrong_2 : 1 = 2 := by
  custom_sorry_2

#print wrong_2
-- theorem wrong_2 : 1 = 2 :=
-- sorryAx (1 = 2)
```

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

The `custom_trivial` 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_trivial`, 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_trivial` to use `(H2 : 2 = 2)` to solve the goal `(2 = 2)`:

```
theorem trivial_correct (H1 : 1 = 1) (H2 : 2 = 2): 2 = 2 := by
  custom_trivial

#print trivial_correct
-- theorem trivial_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_trivial` to throw an error, telling us that we cannot find a hypothesis of the type we are looking for:

```
theorem trivial_wrong (H1 : 1 = 1): 2 = 2 := by
  custom_trivial

#print trivial_wrong
-- tactic 'custom_trivial' 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:

```
elab "custom_trivial_0" : tactic => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  dbg_trace f!"1) goal: {goal.name}"
  let goal_type ← Lean.Elab.Tactic.getMainTarget
  dbg_trace f!"2) goal type: {goal_type}"

theorem trivial_correct_0 (H1 : 1 = 1) (H2 : 2 = 2): 2 = 2 := by
  custom_trivial_0
-- 1) goal: _uniq.638
-- 2) goal type: Eq.{1} Nat (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)) (OfNat.ofNat.{0
-- unsolved goals
-- H1 : 1 = 1
-- H2 : 2 = 2
-- ⊢ 2 = 2

#print trivial_correct_0
-- theorem trivial_correct_0 : 1 = 1 → 2 = 2 → 2 = 2 :=
-- fun H1 H2 => sorryAx (2 = 2)

theorem trivial_wrong_0 (H1 : 1 = 1): 2 = 2 := by
  custom_trivial_0
-- 1) goal: _uniq.713
-- 2) goal type: Eq.{1} Nat (OfNat.ofNat.{0} Nat 2 (instOfNatNat 2)) (OfNat.ofNat.{0
-- unsolved goals
-- H1 : 1 = 1
-- ⊢ 2 = 2

#print trivial_wrong_0
-- theorem trivial_wrong : 1 = 1 → 2 = 2 :=
-- fun H1 => sorryAx (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 => do
  let lctx ← Lean.MonadLCtx.getLCtx -- get the local context.
  lctx.forM fun ldecl: Lean.LocalDecl => do
    let ldecl_expr := ldecl.toExpr -- Find the expression of the declaration.
    let ldecl_name := ldecl.userName -- Find the name of the declaration.
    dbg_trace f!"+ local decl: name: {ldecl_name} | expr: {ldecl_expr}"

theorem test_list_local_decls_1 (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 `LocalDefinition` by calling `Lean.Meta.inferType` on the local declaration's expression.

```
elab "list_local_decls_2" : tactic => do
  let lctx ← Lean.MonadLCtx.getLCtx -- get the local context.
  lctx.forM fun ldecl: Lean.LocalDecl => do
    let ldecl_expr := ldecl.toExpr -- Find the expression of the declaration.
    let ldecl_name := ldecl.userName -- Find the name of the declaration.
    let ldecl_type ← Lean.Meta.inferType ldecl_expr -- **NEW:** Find the type.
    dbg_trace f!"+ local decl: name: {ldecl_name} | expr: {ldecl_expr} | type: {ldec

theorem test_list_local_decls_2 (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}
  -- + 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 `LocalDefinition` 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 => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  let goal_declaration ← Lean.Meta.getMVarDecl goal
  let goal_type := goal_declaration.type
  let lctx ← Lean.MonadLCtx.getLCtx -- get the local context.
  lctx.forM fun ldecl: Lean.LocalDecl => do
    let ldecl_expr := ldecl.toExpr -- Find the expression of the declaration.
    let ldecl_name := ldecl.userName -- Find the name of the declaration.
    let ldecl_type ← Lean.Meta.inferType ldecl_expr -- Find the type.
    let eq? ← Lean.Meta.isExprDefEq ldecl_type goal_type -- **NEW** Check if type eq
    dbg_trace f!"+ local decl[EQUAL? {eq?}]: name: {ldecl_name}"

theorem test_list_local_decls_3 (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_trivial_1" : tactic => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  let goal_type ← 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 ldecl_expr := ldecl.toExpr -- Find the expression of the declaration.
    let ldecl_type ← Lean.Meta.inferType ldecl_expr -- Find the type.
    if (← Lean.Meta.isExprDefEq ldecl_type goal_type) -- Check if type equals goal t
    then return Option.some ldecl_expr -- If equal, success!
    else return Option.none -- Not found.
  dbg_trace f!"matching_expr: {option_matching_expr}"

theorem trivial_correct_1 (H1 : 1 = 1) (H2 : 2 = 2): 2 = 2 := by
```

```
  custom_trivial_1
-- matching_expr: some _uniq.6241
  rfl

#print trivial_correct_1
-- theorem trivial_correct_1 : 1 = 1 → 2 = 2 → 2 = 2 :=
-- fun H1 H2 => sorryAx (2 = 2) false

theorem trivial_wrong_1 (H1 : 1 = 1): 2 = 2 := by
  custom_trivial_1
-- matching_expr: none
  rfl

#print trivial_wrong_1
-- theorem trivial_wrong_1 : 1 = 1 → 2 = 2 :=
-- fun H1 => sorryAx (2 = 2) false
```

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_trivial_2" : tactic => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  let goal_type ← Lean.Elab.Tactic.getMainTarget
  let lctx ← Lean.MonadLCtx.getLCtx
  let option_matching_expr ← lctx.findDeclM? fun ldecl: Lean.LocalDecl => do
    let ldecl_expr := ldecl.toExpr
    let ldecl_type ← Lean.Meta.inferType ldecl_expr
    if ← Lean.Meta.isExprDefEq ldecl_type goal_type
      then return Option.some ldecl_expr
      else return Option.none
  match option_matching_expr with
  | some e => Lean.Elab.Tactic.closeMainGoal e
  | none => do
    Lean.Meta.throwTacticEx `custom_trivial_2 goal (m!"unable to find matching hypot

theorem trivial_correct_2 (H1 : 1 = 1) (H2 : 2 = 2): 2 = 2 := by
  custom_trivial_2

#print trivial_correct_2
-- theorem trivial_correct_2 : 1 = 1 → 2 = 2 → 2 = 2 :=
-- fun H1 H2 => H2

theorem trivial_wrong_2 (H1 : 1 = 1): 2 = 2 := by
  custom_trivial_2
-- tactic 'custom_trivial_2' failed, unable to find matching hypothesis of type (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).

For the first task, we can use `Lean.Elab.Tactic.getGoals` and `Lean.Elab.Tactic.setGoals`:

```
elab "reverse_goals" : tactic => 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
```

Now let's try to simulate a `let` and a `have`. For this task, first we will need to use `Lean.Elab.Tactic.withMainContext`, which can run commands taking into consideration the entire goal state. This is important because if the user has some `n : Nat` in the context and wants to do `custom_have h : n = n := rfl` then our tactic will need to elaborate the type `n = n` while knowing what `n` is.

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.Meta.define` and the later uses `Lean.Meta.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 ← Lean.Meta.define mvarId n.getId t v
      let (_, mvarIdNew) ← Lean.Meta.intro1P mvarIdNew
      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 ← Lean.Meta.assert mvarId n.getId t v
      let (_, mvarIdNew) ← Lean.Meta.intro1P mvarIdNew
      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
```

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`:

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

theorem test_sorry_custom_macro: 1 = 42 := by
  custom_sorry

#print test_sorry_custom_macro
-- theorem test_sorry_custom_macro : 1 = 42 :=
--   sorryAx (1 = 42) false
```

Implementing `trivial`: Extensible Tactics by Macro Expansion

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

```
syntax "custom_trivial" : tactic
```

```
theorem test_custom_trivial_macro_0: 42 = 42 := by
  custom_trivial
-- tactic 'tacticCustom_trivial' has not been implemented
  sorry
```

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

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

theorem test_custom_trivial_macro_1: 42 = 42 := by
   custom_trivial
-- Goals accomplished 🎉
```

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

```
theorem test_custom_trivial_macro_2: 43 = 43 ∧ 42 = 42:= by
  custom_trivial
-- tactic 'rfl' failed, equality expected{indentExpr targetType}
-- ⊢ 43 = 43 ∧ 42 = 42
```

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

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

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

```
theorem test_custom_trivial_macro_3 : 43 = 43 ∧ 42 = 42 := by
  custom_trivial
-- Goals accomplished 🎉
```

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

Implementing <;>: Tactic Combinators by Macro Expansion

Recall that in the previous section, we say that `a <;> b` meant "run `a`, and then run `b` for all goals". In fact, `<;>` itself is a tactic combinator. 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.
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 }
```

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 => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  dbg_trace f!"goal: {goal.name}"

theorem test_faq_main_goal: 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 => do
  let goals ← Lean.Elab.Tactic.getGoals
  goals.forM $ fun goal => do
    let goal_type ← Lean.Meta.getMVarType goal
    dbg_trace f!"goal: {goal.name} | type: {goal_type}"

theorem test_faq_get_goals (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 => do
  let lctx ← Lean.MonadLCtx.getLCtx -- get the local context.
  lctx.forM (fun (ldecl: Lean.LocalDecl) => do
      let ldecl_expr := ldecl.toExpr -- Find the expression of the declaration.
      let ldecl_type := ldecl.type -- Find the expression of the declaration.
      let ldecl_name := ldecl.userName -- Find the name of the declaration.
      dbg_trace f!" local decl: name: {ldecl_name} | expr: {ldecl_expr} | type: {lde
  )

theorem test_faq_get_hypotheses (H1 : 1 = 1) (H2 : 2 = 2): 3 = 3 := by
  faq_get_hypotheses
  -- local decl: name: test_faq_get_hypotheses | 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 => do
  let goal ← Lean.Elab.Tactic.getMainGoal
  Lean.Meta.throwTacticEx `faq_throw_error goal "throwing an error at the current go

theorem test_faq_throw_error (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 meta-variable: `Lean.Meta.getMVarDecl mvar` when `mvar : Lean.MVarId` is in context. For instance, `mvar` could be the goal extracted using `getMainGoal`

- Extract the type of a meta-variable: `Lean.MetavarDecl.type mvdecl` when `mvdecl : Lean.MetavarDecl` 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_decl ← Lean.Meta.getMVarDecl goal
  let goal_type := goal_decl.type
  ```

- 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.isExprDefEq ex1 ex2` when `ex1 ex2 : Lean.Expr` are in context. Returns a `Lean.MetaM Bool`

  `isDefEq ex1 ex2` appears to be a synonym
- 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.Elab.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 aMessageData'` for better printing of terms

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

```
import Lean
open Lean
```

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:

```
#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 cl
```

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
  Elab.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.

```
import Lean
open Lean PrettyPrinter Delaborator SubExpr
```

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:

```
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 `@[builtinDelab` (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 `appUnexpand` 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

@[appUnexpander 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 (t

-- 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:

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

@[appUnexpander LangExpr.ident]
def unexpandIdent : Unexpander
  | `(LangExpr.ident $x:str) =>
    if let some str := x.isStrLit? then
      let name := mkIdent $ Name.mkSimple str
      `([Lang| $name])
    else
      throw ()
  | _ => throw ()

@[appUnexpander LangExpr.letE]
def unexpandLet : Unexpander
  | `(LangExpr.letE $x:str [Lang| $v:lang] [Lang| $b:lang]) =>
    if let some str := x.isStrLit? then
      let name := mkIdent $ Name.mkSimple str
      `([Lang| let $name := $v in $b])
    else
      throw ()
  | _ => 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.