A Practical Guide to Writing Tactics in Lean
Daniel J. Velleman

Suppose you're writing a proof in Lean, using tactic mode, and you invoke the `assumption` tactic. Lean will search through the current context for a hypothesis that matches the goal and, if it finds one, use it to complete the proof. How does Lean know how to do this?

The answer can be found in Lean's library, in the file `core/init.meta.tactic`, which contains this definition:

```
meta def assumption : tactic unit :=
do { ctx ← local_context,
     t ← target,
     H ← find_same_type t ctx,
     exact H }
<|> fail "assumption tactic failed"
```

(You can find this file at `https://leanprover-community.github.io/mathlib_docs/`. On the left, under `core`, click on `init`, then `meta`, then `tactic`.)

At the moment, this definition probably doesn't mean much to you. The purpose of this guide is to explain how tactics are defined in Lean so that you will be able to read definitions like this and write definitions to create your own tactics. Here are some other resources that you may find useful:

- A Tactic Writing Tutorial, which can be found here: `https://leanprover-community.github.io/extras/tactic_writing.html`
- Chapters 6 and 7 of *The Hitchhiker's Guide to Logical Verification*, by Baanen, Bentkamp, Blanchette, Hölzl, and Limperg (`https://github.com/blanchette/logical_verification_2021/raw/main/hitchhikers_guide.pdf`).

1 Preliminaries

Tactics are functions, so it may be helpful to review briefly some aspects of how functions are defined in Lean. Here is a simple function definition:

```
def absval (n : int) : int :=
  if n < 0 then - n else n
```

The definition is introduced by the keyword `def`. The name of the function is `absval`, it takes an argument `n` of type `int`, and it returns an `int`. The return value is given by the `if-then-else` expression in the second line.

A Lean function can call another function, so once we have defined the `absval` function, we can define this function:

```
def abs_sum (m n : int) : int :=
  (absval m) + (absval n)
```

An alternative way to define this function would be:

```
def abs_sum (m n : int) : int :=
do let avm := absval m,
   let avn := absval n,
   avm + avn
```

This version gives the illusion that the function is defined by a procedural program: first `absval m` is computed and stored in the variable `avm`, then `absval n` is computed and stored in `avn`, and finally `avm` and `avn` are added. This illusion can be helpful for writing complex functions, but it is important to understand that it is only an illusion. In particular, the syntax `let ... := ...` cannot stand on its own; it must be followed by a comma and then another expression. Thus, for example, the following is ungrammatical:

```
def abs_sum (m n : int) : int :=
do if m < 0 then let avm := - m else let avm := m,
   if n < 0 then let avn := - n else let avn := n,        WRONG!
   avm + avn
```

The problem is that the `then` clause of the `if-then-else` construct is the incomplete expression `let avm := - m`. However, one could write:

```
def abs_sum (m n : int) : int :=
do let avm := if m < 0 then - m else m,
   let avn := if n < 0 then - n else n,
   avm + avn
```

2 Tactics

The art of writing tactics is called metaprogramming, and a tactic definition is introduced by the keyword `meta`. Here is a very simple tactic:

```
meta def tactic.interactive.greet : tactic unit :=
tactic.trace "Good day"
```

This defines a tactic called `tactic.interactive.greet`. The tactic has type `tactic unit`; we'll explain what this means later. The body of the tactic consists of the line `tactic.trace "Good day"`, which tells Lean to display the message `Good day` to the user.

By putting this tactic in the `tactic.interactive` namespace, we allow a user who is writing a proof in tactic mode to invoke the tactic by typing `greet`. To see this tactic in action, type the definition above into a Lean file and then try this example:

```
example : true :=
begin
   greet,
   trivial
end
```

If you are using VS Code, then the word `greet` should have a squiggly green line under it, and if you click on it you should see the message `Good day` from Lean.

Let's try making the tactic a little more complicated. We begin by defining a function that has a natural number argument and returns one of three different greetings:

```
def greeting (n : nat) : string :=
match n with
   | 0 := "Good morning"
   | 1 := "Good afternoon"
   | _ := "Good evening"
end
```

Now we can rewrite the tactic like this:

```
meta def tactic.interactive.greet (n : nat) : tactic unit :=
tactic.trace (greeting n)
```

Returning to the example above, we will need to specify a natural number argument for the `greet` tactic. To get Lean to greet you with `Good morning`, we would rewrite the example like this:

```
example : true :=
begin
   greet 0,
   trivial
end
```

Change `greet 0` to `greet 1` to get Lean to say `Good afternoon`, and use `greet 2` for `Good evening`.

Like function definitions, tactic definitions can also be written in a way that simulates procedural programming. To write a tactic definition in this simulated procedural style, we use the keyword do followed by a sequence of commands, separated by commas. Here is our `greet` tactic, rewritten in this style:

```
meta def tactic.interactive.greet (n : int) : tactic unit :=
do let g := greeting n,
   tactic.trace g
```

This tactic definition illustrates that a tactic can call a function and use the function's return value. A tactic can also call another tactic and use that tactic's return value, but the situation is more complicated because of some differences between tactics and ordinary functions. Unlike an ordinary function, a tactic can access the current context of the proof being written—that is, the list of goals remaining to be established to complete the proof and the hypotheses in effect for those goals. A tactic can change this context—for example, by adding or removing a hypothesis or changing a goal. And a tactic can fail—if you have been writing proofs in tactic mode, you have no doubt experienced a tactic failing. These differences make it more complicated for a tactic to call another tactic and use its return value.

Suppose, for example, that you want to define a tactic that first calls some tactic `T` that returns a value and then does some further processing with that returned value. The tactic `T` might change the proof context, and those changes must be passed on to the next step of your tactic, along with the value returned by `T`. And `T` might fail, in which case there will be no returned value and your tactic will be unable to continue.

Let's begin by illustrating the possibility that a tactic can fail. We'll rewrite our `greet` tactic, changing the `greeting` function to a tactic that may fail. This will require a number of small changes to the definitions of `greeting` and `greet`. We'll give the new definitions first and explain the changes afterwards:

```
meta def greeting (n : int) : tactic string :=
match n with
   | 0 := return "Good morning"
   | 1 := return "Good afternoon"
   | 2 := return "Good evening"
   | _ := fail "Illegal input"
end

meta def tactic.interactive.greet (n : int) : tactic unit :=
do g ← greeting n,
   tactic.trace g
```

Let's run through the changes that have been made. The definition of `greeting` now starts with the keyword `meta`, and the return type is now `tactic string` rather than `string`. Think of this return type as meaning that when the tactic is applied in a proof context, it produces a box that contains a string, together with all of the other information needed to deal with the complications described earlier that arise when one tactic calls another. We'll call this a "`tactic string` box." For example, the box must contain information about how the proof context may have been changed by the `greeting` tactic, and it must contain information about whether or not the tactic failed.

Since the return type of `greeting` is now `tactic string` rather than `string`, the cases in the `match` statement must be changed. The string values corresponding to the different values of `n` are now preceded by `return`. You could think of `return "Good morning"` as meaning "put the string `"Good morning"` inside a `tactic string` box"; this `tactic string` box is then what is returned when the `greeting` tactic is applied in a proof context. The `match` statement also now has an additional case. If the argument `n` is anything other than 0, 1, or 2, then the tactic fails, with the failure message `Illegal input`.

Finally, in the `greet` tactic, we have written `g ← greeting n` rather than `let g := greeting n`. (In VS Code, type `\l` to get the symbol ←.) You can think of this command as meaning: "Invoke the tactic `greeting n` in the current proof context. If it succeeds, extract the `string` value from the resulting `tactic string` box, assign that value to `g`, and then pass on that value, along with the other information in the box, to the next step."

There's a lot going on here that has been left out of this description. If you want to know more, I recommend Chapters 6 and 7 of *The Hitchhiker's Guide to Logical Verification*. But you don't really need to know the details to be able to write tactics. You just need to know that if you want to invoke a function `f` that returns a value of type α, then you should write `let x := f,` followed by further steps that may use `x`, and if you want to invoke a tactic `T` that returns a value of type `tactic α`, then you should write `x ← T,` followed by further steps that may use `x`; in both cases, `x` will have type α.

If you try out the new version of the `greet` tactic, you should find that if you invoke the tactic with any argument larger than 2, then the green squiggly line under `greet` will change to red, and Lean will display the message `Illegal input`. In this case the execution of the `greet` tactic stops as soon as the `greeting` tactic fails; the instruction `tactic.trace g` does not get executed.

As with the `let x := f` construction, it is important to keep in mind that `x ← T` cannot stand on its own, but must be followed by a comma and then further commands. So the following example would be ungrammatical:

```
meta def tactic.interactive.wrong (b : bool) :
     tactic unit :=
do if b then x ← greeting 0 else x ← greeting 1,
   tactic.trace x
```

WRONG!

However, if you are tempted to write this, then you can probably get the effect you intended like this:

```
meta def tactic.interactive.right (b : bool) : tactic unit :=
do x ← if b then greeting 0 else greeting 1,
   tactic.trace x
```

There is an alternative notation that is sometimes useful for defining a tactic that involves a sequence of steps. If `A` and `B` are tactics, then `A >> B` means the same thing as `do A, B`. Also, if `A` has type `tactic α` and `B` takes an argument of type α, then `A >>= B` means the same thing as `do x ← A, B x`. For example, we could rewrite the `greet` tactic like this:

```
meta def tactic.interactive.greet (n : nat) : tactic unit :=
greeting n >>= tactic.trace
```

This saves us the trouble of giving a name to the value returned by `greeting n`; that nameless value is simply passed along to `tactic.trace`. While this notation is convenient in simple cases, in more complex tactic definitions it is probably preferable to use the `do` syntax.

We can now explain the type `tactic unit` of our `greet` tactic. The term `unit` here is a type. There is only one object of type `unit`, and it contains no information; it is denoted by `()`. The `greet` tactic thus produces a `tactic unit` box that contains this informationless object (together, of course, with the information described earlier that makes metaprogramming possible). Many tactics have no information to return, and are therefore declared to have type `tactic unit`. In fact, `tactic.trace` is a tactic with type `tactic unit` (it is defined in the same file `core/init.meta.tactic` that contains the definition of `assumption`). Since the last step in our `greet` tactic is `tactic.trace`, the value produced by this call to `tactic.trace`, which is a `tactic unit` box containing `()`, is the value returned by `greet`.

3 Names, Expressions, and Pre-Expressions

So far we have not written a tactic that does anything useful. Before we can write useful tactics, we will need to know about three types: `name`, `expr`, and `pexpr`, which represent names, expressions, and pre-expressions.

Many things in Lean have names: variables, constants, functions, theorems, and so on. Examples of names include `x`, `nat`, `or.intro_left`, and `tactic.interactive.greet`. There is a special backtick notation for defining names: if you write ``let n := `x``, then `n` will have type `name`, and its value will be `x`. You can try it out as follows:

```
meta def tactic.interactive.trace_name : tactic unit :=
do let n := `x,
   tactic.trace (to_string n)

example : true :=
begin
   trace_name,
   trivial
end
```

The type `expr` represents an expression in Lean, such as a proposition or a proof. There is a similar backtick notation for defining expressions: the expression is enclosed in parentheses and then preceded by a backtick. Here is a simple example:

```
meta def tactic.interactive.trace_expr : tactic unit :=
do let e : expr := `(true ∨ ¬ true),
   tactic.trace (to_string e)

example : true :=
begin
   trace_expr,
   trivial
end
```

(Note that it is necessary to explicitly give `e` the type `expr`; otherwise it will have a type that reduces to `expr` but is not the same as `expr`.) If you check the message generated by the `trace_expr` tactic, you will see that it is `or true (not true)`. This shows that the value assigned to `e` is the *parsed form* of the expression `true ∨ ¬ true`: it is represented as the function `or` applied first to the constant `true` and then to the result of applying the function `not` to the constant `true`.

There is an important subtlety to the backtick notation for expressions. We can illustrate it using the functions `or.intro_left` and `or.inl`. Recall that if `ha` is a proof of some proposition `a` and `b` is another proposition, then `or.intro_left b ha` is a proof of `a ∨ b`; in other words, it is an expression of type `a ∨ b`. For example, `or.intro_left false trivial` has type `true ∨ false`, because `trivial` is a proof of `true`. You can confirm this with the Lean command `#check or.intro_left false trivial`, which reports the type of the expression.

It may appear that the function `or.intro_left` takes two arguments, but it actually takes three. You can see this by giving the command `#check @or.intro_left`, which reports that the type of `or.intro_left` is `∀ {a : Prop} (b : Prop), a → a ∨ b`. In other words, `or.intro_left` takes a proposition `a`, a proposition `b`, and a proof of `a`, and returns a proof of `a ∨ b`. The curly braces around the first argument, the proposition `a`, indicate that this argument is *implicit*. In the expression `or.intro_left false trivial`, that argument has been left out, and when Lean interprets the expression, it infers the missing first argument by a process called *elaboration*.

In many cases, Lean would be able to infer the second argument as well, so Lean has a version of the or-introduction rule in which both of the first two arguments are implicit. The command `#check or.inl` shows that the function `or.inl` has type `∀ {a b : Prop}, a`

→ a ∨ b. Thus, if `ha` is a proof of `a`, then `or.inl ha` will be a proof of `a ∨ b`, where Lean will try to infer `b` from the context. If you give the command `#check or.inl trivial`, Lean reports that the expression `or.inl trivial` has type `true ∨ ?M_1`. The `?M_1` here is a *metavariable*; it is a placeholder, indicating that Lean was unable to infer the proposition `b`, so it has left it unspecified.

With that preparation, let's investigate what happens if we use these expressions in our `trace_expr` tactic. First try replacing `` `(true ∨ ¬ true) `` in the definition of the `trace_expr` tactic with `` `(or.intro_left false trivial) ``. When the `trace_expr` tactic is invoked in the example proof, the message generated by the `tactic.trace` command is `or.intro_left true false trivial`. Thus we see that Lean has elaborated the expression and inferred (correctly) that the missing first argument is `true`.

Now try putting in `` `(or.inl trivial) ``. In VS Code, there will be a squiggly red line underneath `or`, and if you click on it, you will see the error message `don't know how to synthesize placeholder`. What's going on here is that Lean tries to elaborate the expression inside the `` `(...) `` notation *when it parses the definition of the tactic*. If that elaboration produces a metavariable, Lean reports an error. The lesson here is that the backtick notation for expressions can only be used for expressions whose elaboration does not produce metavariables.

Does this mean that we can't work with the expression `or.inl trivial` in a tactic? Not at all. We just have to tell Lean not to try to elaborate it when it parses the definition of the tactic. To do that, we put *two* backticks before the open parenthesis. Instead of an `expr`, we get a `pexpr`; this type represents a *pre-expression*, which is just an unelaborated expression. We can ask Lean to elaborate the pre-expression *when the tactic is executed* by invoking the tactic `tactic.to_expr`, which takes an argument of type `pexpr`, elaborates it, and returns the resulting `expr`. Here's a version of our `trace_expr` tactic that illustrates this:

```
meta def tactic.interactive.trace_expr : tactic unit :=
do let p : pexpr := ``(or.inl trivial),
   e ← tactic.to_expr p,
   tactic.trace ("The pre-expression is " ++ (to_string p)
       ++ " and the expression is " ++ (to_string e))

example : true :=
begin
   trace_expr,
   trivial,
   exact false
end
```

Take a look at the message generated by this version of the `trace_expr` tactic. It should be something like `The pre-expression is or.inl trivial and the expression is or.inl true ?_mlocal._fresh.800.574 trivial`. We see that when Lean elaborated the pre-expression, it was able to infer that the first implicit argument was `true`, but it generated a metavariable for the second argument, which it was unable to infer. And notice something else that has happened: after the `trace_expr` tactic is invoked, there are *two* goals required to finish the proof: `⊢ true`, which was the original goal of the example, and `⊢ Prop`, which is asking for a proposition to be supplied as the value of the metavariable. The `trivial` tactic accomplishes the first goal, and `exact false` takes care of the second.

Let's do one more experiment. In the last version of the `trace_expr` tactic, try changing ``` ``(or.inl trivial) ``` to ``` ``(or.inl true) ```. Now the invocation of `trace_expr` in the example will have a red squiggle under it, and if you click on it you will see the error message `type mismatch at application`. This indicates that the call to `tactic.to_expr` failed, because `true` does not have the right type to be an argument of the function `or.inl`.

There is one more subtle point that is worth mentioning. An expression can consist of just a single constant, as in the following example:

```
meta def tactic.interactive.trace_name_and_expr : tactic unit :=
do let n := `not,
   let e : expr := `(not),
   tactic.trace ("The name is " ++ (to_string n)
       ++ " and the expression is " ++ (to_string e))

example : true :=
begin
   trace_name_and_expr,
   trivial
end
```

The output is `The name is not and the expression is not`. It may appear that `n` and `e` are the same, but they are different: `n` is the name `not`, whereas `e` is an expression consisting of a single constant (whose name is `not`). This distinction will be important in the next section.

In our last example, the argument of `tactic.trace` was a string constructed by combining explicit strings, which appear in quotation marks, with expressions outside of those quotation marks that compute strings. There is a similar mechanism, called *antiquotation*, that allows us to compute expressions in a similar way. If `x` has type `expr` or `pexpr`, then in a double-backtick specification of a `pexpr` we can use the notation `%%x` to indicate a place where the expression that is the value of `x` should be inserted. Here is an example:

```
meta def tactic.interactive.trace_expr (l b : bool) : tactic unit :=
do let o : pexpr := if l then ``(or.intro_left)
                    else ``(or.intro_right),
   let d : expr := if b then `(true) else `(false),
   let p : pexpr := ``(%%o %%d trivial),
   e ← tactic.to_expr p,
   tactic.trace (to_string e)

example : true :=
begin
   trace_expr tt ff,
   trivial
end
```

When the tactic is invoked with the arguments `tt ff`, as in the example above, `o` gets the value ``` ``(or.intro_left) ``` and `d` gets the value `` `(false) ``. And when these values are inserted at the positions marked by `%%o` and `%%d`, the pre-expression assigned to `p` ends up being `or.intro_left false trivial`. This pre-expression is then elaborated to produce `e`, which is the expression `or.intro_left true false trivial`. Try varying the arguments passed to `trace_expr` to see how the results change.

Note that `o` had to be a pre-expression, because the elaboration of `or.intro_left` or `or.intro_right` would introduce metavariables, and `p` had to be a pre-expression, because `p` cannot be elaborated until execution time. In some cases, the antiquotation notation can be used in a single-backtick specification of an `expr`, but usually we will want to use this notation when defining a `pexpr`.

4 Accessing and Altering the Proof Context

We are finally ready to write tactics that do something useful. It may be helpful to have a simple example in front of us, so consider this example:

```
example (p q : Prop) (h1 : ¬ (p ∧ q)) : ¬ p ∨ ¬ q
```

When you start writing this proof in tactic mode, the state looks like this:

```
p q : Prop
h1 : ¬ (p ∧ q)
⊢ ¬ p ∨ ¬ q
```

We begin by discussing tactics in Lean's library that allow us to access the proof context. The tactic `tactic.target` returns an `expr` that is the current goal—the proposition that needs to be proven. To access a hypothesis, we must use the name of the hypothesis. The tactic `tactic.get_local` takes an argument `n` of type `name` and returns an `expr` that is the constant whose name is `n`. (Actually, in this case the `expr` is what is called a *local constant*. As we saw in the last section, the name and the expression are not the same thing.) The tactic `tactic.infer_type` takes an argument `e` of type `expr` and returns an `expr` that is the type of `e`. Here is an example that illustrates the use of these tactics:

```
meta def tactic.interactive.access_context : tactic unit :=
do P ← tactic.get_local `p,
   pt ← tactic.infer_type P,
   H ← tactic.get_local `h1,
   ht ← tactic.infer_type H,
   g ← tactic.target,
   tactic.trace ("The type of p is " ++ (to_string pt)
     ++ ", the type of h1 is " ++ (to_string ht)
     ++ ", and the goal is " ++ (to_string g))
```

Now we can try using the tactic in the example above:

```
example (p q : Prop) (h1 : ¬ (p ∧ q)) : ¬ p ∨ ¬ q :=
begin
   access_context
end
```

The output is `The type of p is Prop, the type of h1 is not (and p q), and the goal is or (not p) (not q)`. (Since the types are returned as expressions, they are in parsed form.)

Of course, it would be better if we didn't have to hard code the names `p and `h1 in this tactic. So you might try rewriting it like this:

```
meta def tactic.interactive.access_context
     (h : name) : tactic unit :=
do H ← tactic.get_local h,
   ht ← tactic.infer_type H,
   g ← tactic.target,
   tactic.trace ("The type of " ++ (to_string h)
     ++ " is " ++ (to_string ht)
     ++ " and the goal is " ++ (to_string g))
```

Unfortunately, if we invoke the tactic with `access_context h1`, we get the error message `unknown identifier ‘h1’`. It would work to write `access_context `h1`, but we don't want users to have to know about backticks. A better idea is to tell Lean that the argument to the `access_context` tactic should not be *interpreted* as an identifier; rather, we want the `name` that results from *parsing* the argument as an identifier to be passed to the tactic. To do this, we need to rewrite the declaration of the tactic. We can also save a little typing by opening the `tactic` namespace:

```
open tactic
meta def tactic.interactive.access_context
     (h : interactive.parse lean.parser.ident) : tactic unit :=
do H ← get_local h,
   ht ← infer_type H,
   g ← target,
   trace ("The type of " ++ (to_string h)
     ++ " is " ++ (to_string ht)
     ++ " and the goal is " ++ (to_string g))
```

Now `access_context h1` generates the output `The type of h1 is not (and p q) and the goal is or (not p) (not q)`.

To understand what's going on here, it might be helpful note that the command `#reduce interactive.parse lean.parser.ident` produces the output `name`. Thus, the argument `h` in this tactic is a name, but by writing the type in this way we have told Lean to do the parsing we want.

Now that we can *access* the context of the proof, how do we *alter* it? The easiest way is to invoke the tactics that you are already familiar with from writing proofs in tactic mode. For instance, consider the tactic `have h : t := p`, which adds the hypothesis `h : t`, if `p` is a term of type `t`. If `h` is left out, then the new hypothesis is labeled `this`; if `t` is left out then it is inferred from `p`; and if `p` is left out then `t` becomes a new goal that must be proven. In the example above, the command `have : ¬ p ∨ ¬ q := not_and_distrib.mp h1` would cause Lean to infer the new hypothesis `this : ¬ p ∨ ¬ q`. (Note that you must `import logic.basic` for the function `not_and_distrib` to be recognized.)

The tactic `have` is defined in Lean's library (in the `tactic.interactive` namespace, of course). It has three arguments, corresponding to `h`, `t`, and `p` in the last paragraph, and their types are `option name`, `option pexpr`, and `option pexpr`, respectively. Using the `have` tactic, we can finally write a tactic that does something useful:

```
open tactic
```

```
meta def tactic.interactive.dm
    (h : interactive.parse lean.parser.ident) : tactic unit :=
do H ← get_local h,
   t ← tactic.infer_type H,
   match t with
    | `(¬(%%l ∧ %%r)) := tactic.interactive.have none
              ``(¬ %%l ∨ ¬ %%r) ``(not_and_distrib.mp %%H)
    | `(¬(%%l ∨ %%r)) := tactic.interactive.have none
              ``(¬ %%l ∧ ¬ %%r) ``(not_or_distrib.mp %%H)
    | _ := fail "De Morgan's laws don't apply"
end
```

This tactic uses backtick notation and antiquotation for pattern matching. In the first case of the match statement, if `t` has the form of the negation of a conjunction, then `l` and `r`, which have type `expr`, are set equal to the left and right sides of the conjunction. These expressions are then used to construct the pre-expressions that are passed to `have`. Similarly, the second case handles negations of disjunctions. Each call to `have` infers a new hypothesis labeled `this` by applying one of De Morgan's laws. We can use this tactic to complete the example at the beginning of this section:

```
example (p q : Prop) (h1 : ¬ (p ∧ q)) : ¬ p ∨ ¬ q :=
begin
   dm h1,
   exact this
end
```

The declaration of the `have` tactic (which can be found in the Lean library, in the file `core/init.meta.interactive`) is:

```
meta def tactic.interactive.have
   (h : interactive.parse lean.parser.ident?)
   (q₁ : interactive.parse
      (lean.parser.tk ":" *> interactive.types.texpr)?)
   (q₂ : interactive.parse
      (lean.parser.tk ":=" *> interactive.types.texpr)?)
   : tactic unit
```

We have already explained that if a tactic has an argument that is declared to have type `interactive.parse lean.parser.ident`, then the argument is simply a name, with the complicated type expression instructing Lean to parse the user's input to produce the name. Putting a question mark after `lean.parser.ident` is defined in the library file to be a shorthand for `optional lean.parser.ident`, so the type declaration for `h` means

```
(h : interactive.parse (optional lean.parser.ident))
```

As before, the command `#reduce interactive.parse (optional lean.parser.ident)` can be used to determine that the type of `h` is `option name`.

To figure out the type of q_1, you can give the command `#reduce interactive.parse (lean.parser.tk ":" *> interactive.types.texpr)`. The output is `expr ff`, which means "expression that has not been elaborated." This is the same as `pexpr`; indeed, `#reduce pexpr` also produces the output `expr ff`. Again, adding a question mark makes the argument optional, so the type of q_1 is `option pexpr`. This time the instructions to the parser tell the parser to look for the token "`:`" followed by a pre-expression. Similarly, the type of q_2 tells the parser to look for the token "`:=`" followed by a pre-expression, and the argument is optional, so the type of q_2 is also `option pexpr`.

If you are trying to use another tactic from the Lean library, look at the declaration in the library file to determine what types of arguments to pass to the tactic. If necessary, you can use the `#reduce` command to help you determine the types. For example, looking in the library file `core/init.meta.interactive` we find that the declaration of the `apply` tactic is:

```
meta def tactic.interactive.apply
   (q : interactive.parse interactive.types.texpr) : tactic unit
```

This tells us that the argument has type `pexpr`. We will use this in our next version of the `dm` tactic, which will apply one of De Morgan's laws to either a hypothesis or the conclusion:

```
meta def tactic.interactive.dm
   (n : interactive.parse
      (optional (lean.parser.tk "at" *> lean.parser.ident)))
   (l : interactive.parse
      (optional (lean.parser.tk "with" *> lean.parser.ident)))
   : tactic unit :=
match n with
  | some h := do H ← get_local h,
      t ← infer_type H,
      let label := match l with
        | some h := h
        | none := `this
      end,
      match t with
        | `(¬(%%l ∧ %%r)) := tactic.interactive.have label
                  ``(¬ %%l ∨ ¬ %%r) ``(not_and_distrib.mp %%H)
        | `(¬(%%l ∨ %%r)) := tactic.interactive.have label
                  ``(¬ %%l ∧ ¬ %%r) ``(not_or_distrib.mp %%H)
        | _ := fail "De Morgan's laws don't apply"
      end
  | none := do t ← target,
      match t with
        | `(¬(%%l ∧ %%r)) := tactic.interactive.apply
                     ``(not_and_distrib.mpr)
        | `(¬(%%l ∨ %%r)) := tactic.interactive.apply
                     ``(not_or_distrib.mpr)
        | _ := fail "De Morgan's laws don't apply"
      end
end
```

The tactic now takes two arguments. The first argument `n` is an optional name, preceded by the token "`at`", specifying the hypothesis to which De Morgan's law should be applied. If this name is not included, the tactic operates on the goal. The second argument `l` is an optional name preceded by the token "`with`". If this name is supplied and the tactic is applied to a hypothesis, then the name is used instead of `this` as the label of the inferred statement. The tactic checks whether or not the first argument `n` was supplied. If so, it uses the `have` tactic as before; if not, it uses the `apply` tactic to rewrite the goal using one of De Morgan's laws.