SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

HOARE

HOARE LOGIC, PART I

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Imp.
Require Import Maps.
```

In the final chaper of *Logical Foundations* (*Software Foundations*, volume 1), we began applying the mathematical tools developed in the first part of the course to studying the theory of a small programming language, Imp.

- We defined a type of *abstract syntax trees* for Imp, together with an *evaluation relation* (a partial function on states) that specifies the *operational semantics* of programs.

  The language we defined, though small, captures some of the key features of full-blown languages like C, C++, and Java, including the fundamental notion of mutable state and some common control structures.

- We proved a number of *metatheoretic properties* — "meta" in the sense that they are properties of the language as a whole, rather than of particular programs in the language. These included:
  - determinism of evaluation
  - equivalence of some different ways of writing down the definitions (e.g., functional and relational definitions of arithmetic expression evaluation)
  - guaranteed termination of certain classes of programs
  - correctness (in the sense of preserving meaning) of a number of useful program transformations
  - behavioral equivalence of programs (in the Equiv chapter).

If we stopped here, we would already have something useful: a set of tools for defining and discussing programming languages and language features that are mathematically precise, flexible, and easy to work with, applied to a set of key properties. All of these properties are things that language designers, compiler writers, and users might care about knowing. Indeed, many of them are so fundamental to our understanding of the programming languages we deal with that we might not consciously recognize them as "theorems." But properties that seem intuitively obvious can sometimes be quite subtle (sometimes also subtly wrong!).

We'll return to the theme of metatheoretic properties of whole languages later in this volume when we discuss *types* and *type soundness*. In this chapter, though, we turn to a different set of issues.

Our goal is to carry out some simple examples of *program verification* — i.e., to use the precise definition of Imp to prove formally that particular programs satisfy particular specifications of their behavior. We'll develop a reasoning system called *Floyd-Hoare Logic* — often shortened to just *Hoare Logic* — in which each of the syntactic constructs of Imp is equipped with a generic "proof rule" that can be used to reason compositionally about the correctness of programs involving this construct.

Hoare Logic originated in the 1960s, and it continues to be the subject of intensive research right up to the present day. It lies at the core of a multitude of tools that are being used in academia and industry to specify and verify real software systems.

Hoare Logic combines two beautiful ideas: a natural way of writing down *specifications* of programs, and a *compositional proof technique* for proving that programs are correct with respect to such specifications — where by "compositional" we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

This chapter:

- A systematic method for reasoning about the *functional correctness* of programs in Imp

Goals:

- a natural notation for *program specifications* and
- a *compositional* proof technique for program correctness

Plan:

- specifications (assertions / Hoare triples)
- proof rules
- loop invariants
- decorated programs
- examples

Assertions

To talk about specifications of programs, the first thing we need is a way of making *assertions* about properties that hold at particular points during a program's execution

— i.e., claims about the current state of the memory when execution reaches that point. Formally, an assertion is just a family of propositions indexed by a `state`.

```
Definition Assertion := state → Prop.
```

Exercise: 1 star, optional (assertions)

Paraphrase the following assertions in English (or your favorite natural language).

```
Module ExAssertions.
Definition as1 : Assertion := fun st ⇒ st X = 3.
Definition as2 : Assertion := fun st ⇒ st X ≤ st Y.
Definition as3 : Assertion :=
  fun st ⇒ st X = 3 ∨ st X ≤ st Y.
Definition as4 : Assertion :=
  fun st ⇒ st Z * st Z ≤ st X ∧
             ¬ (((S (st Z)) * (S (st Z))) ≤ st X).
Definition as5 : Assertion := fun st ⇒ True.
Definition as6 : Assertion := fun st ⇒ False.
(* FILL IN HERE *)
End ExAssertions.
```

☐

This way of writing assertions can be a little bit heavy, for two reasons: (1) every single assertion that we ever write is going to begin with `fun st ⇒`; and (2) this state `st` is the only one that we ever use to look up variables in assertions (we will never need to talk about two different memory states at the same time). For discussing examples informally, we'll adopt some simplifying conventions: we'll drop the initial `fun st ⇒`, and we'll write just `X` to mean `st X`. Thus, instead of writing

```
      fun st ⇒ (st Z) * (st Z) ≤ m ∧
                 ¬ ((S (st Z)) * (S (st Z)) ≤ m)
```

we'll write just

```
      Z * Z ≤ m ∧ ~((S Z) * (S Z) ≤ m).
```

This example also illustrates a convention that we'll use throughout the Hoare Logic chapters: in informal assertions, capital letters like `X`, `Y`, and `Z` are Imp variables, while lowercase letters like `x`, `y`, `m`, and `n` are ordinary Coq variables (of type `nat`). This is why, when translating from informal to formal, we replace `X` with `st X` but leave `m` alone.

Given two assertions `P` and `Q`, we say that `P` *implies* `Q`, written `P ->> Q`, if, whenever `P` holds in some state `st`, `Q` also holds.

```
Definition assert_implies (P Q : Assertion) : Prop :=
  ∀ st, P st → Q st.

Notation "P ->> Q" := (assert_implies P Q)
                      (at level 80) : hoare_spec_scope.
Open Scope hoare_spec_scope.
```

(The `hoare_spec_scope` annotation here tells Coq that this notation is not global but is intended to be used in particular contexts. The `Open Scope` tells Coq that this file is one such context.)

We'll also want the "iff" variant of implication between assertions:

```
Notation "P <<->> Q" :=
  (P ->> Q ∧ Q ->> P) (at level 80) : hoare_spec_scope.
```

Hoare Triples

Next, we need a way of making formal claims about the behavior of commands.

In general, the behavior of a command is to transform one state to another, so it is natural to express claims about commands in terms of assertions that are true before and after the command executes:

- "If command `c` is started in a state satisfying assertion `P`, and if `c` eventually terminates in some final state, then this final state will satisfy the assertion `Q`."

Such a claim is called a *Hoare Triple*. The assertion `P` is called the *precondition* of `c`, while `Q` is the *postcondition*.

Formally:

```
Definition hoare_triple
           (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
     c / st \\ st' →
     P st →
     Q st'.
```

Since we'll be working a lot with Hoare triples, it's useful to have a compact notation:

```
       {{P}} c {{Q}}.
```

(The traditional notation is `{P} c {Q}`, but single braces are already used for other things in Coq.)

```
Notation "{{ P }} c {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level)
  : hoare_spec_scope.
```

Exercise: 1 star, optional (triples)

Paraphrase the following Hoare triples in English.

```
   1) {{True}} c {{X = 5}}

   2) {{X = m}} c {{X = m + 5)}}

   3) {{X ≤ Y}} c {{Y ≤ X}}
```

```
4) {{True}} c {{False}}

5) {{X = m}}
   c
   {{Y = real_fact m}}

6) {{X = m}}
   c
   {{(Z * Z) ≤ m ∧ ¬ (((S Z) * (S Z)) ≤ m)}}
```

□

Exercise: 1 star, optional (valid_triples)

Which of the following Hoare triples are *valid* — i.e., the claimed relation between `P`, `c`, and `Q` is true?

```
1) {{True}} X ::= 5 {{X = 5}}

2) {{X = 2}} X ::= X + 1 {{X = 3}}

3) {{True}} X ::= 5; Y ::= 0 {{X = 5}}

4) {{X = 2 ∧ X = 3}} X ::= 5 {{X = 0}}

5) {{True}} SKIP {{False}}

6) {{False}} SKIP {{True}}

7) {{True}} WHILE true DO SKIP END {{False}}

8) {{X = 0}}
     WHILE X = 0 DO X ::= X + 1 END
   {{X = 1}}

9) {{X = 1}}
     WHILE !(X = 0) DO X ::= X + 1 END
   {{X = 100}}
```

□

To get us warmed up for what's coming, here are two simple facts about Hoare triples. (Make sure you understand what they mean.)

```
Theorem hoare_post_true : ∀ (P Q : Assertion) c,
  (∀ st, Q st) →
  {{P}} c {{Q}}.
```

+

```
Theorem hoare_pre_false : ∀ (P Q : Assertion) c,
  (∀ st, ~(P st)) →
  {{P}} c {{Q}}.
```

+

Proof Rules

The goal of Hoare logic is to provide a *compositional* method for proving the validity of specific Hoare triples. That is, we want the structure of a program's correctness proof to mirror the structure of the program itself. To this end, in the sections below, we'll introduce a rule for reasoning about each of the different syntactic forms of commands in Imp — one for assignment, one for sequencing, one for conditionals, etc. — plus a couple of "structural" rules for gluing things together. We will then be able to prove programs correct using these proof rules, without ever unfolding the definition of `hoare_triple`.

Assignment

The rule for assignment is the most fundamental of the Hoare logic proof rules. Here's how it works.

Consider this valid Hoare triple:

```
{{ Y = 1 }}  X ::= Y  {{ X = 1 }}
```

In English: if we start out in a state where the value of `Y` is `1` and we assign `Y` to `X`, then we'll finish in a state where `X` is `1`. That is, the property of being equal to `1` gets transferred from `Y` to `X`.

Similarly, in

```
{{ Y + Z = 1 }}  X ::= Y + Z  {{ X = 1 }}
```

the same property (being equal to one) gets transferred to `X` from the expression `Y + Z` on the right-hand side of the assignment.

More generally, if `a` is *any* arithmetic expression, then

```
{{ a = 1 }}  X ::= a {{ X = 1 }}
```

is a valid Hoare triple.

Even more generally, to conclude that an arbitrary assertion `Q` holds after `X ::= a`, we need to assume that `Q` holds before `X ::= a`, but *with all occurrences of* `X` replaced by `a` in `Q`. This leads to the Hoare rule for assignment

```
{{ Q [X |-> a] }} X ::= a {{ Q }}
```

where "`Q [X |-> a]`" is pronounced "`Q` where `a` is substituted for `X`".

For example, these are valid applications of the assignment rule:

```
      {{ (X ≤ 5) [X |-> X + 1]
         i.e., X + 1 ≤ 5 }}
      X ::= X + 1
      {{ X ≤ 5 }}

      {{ (X = 3) [X |-> 3]
         i.e., 3 = 3}}
      X ::= 3
      {{ X = 3 }}

      {{ (0 ≤ X ∧ X ≤ 5) [X |-> 3]
         i.e., (0 ≤ 3 ∧ 3 ≤ 5)}}
      X ::= 3
      {{ 0 ≤ X ∧ X ≤ 5 }}
```

To formalize the rule, we must first formalize the idea of "substituting an expression for an Imp variable in an assertion", which we refer to as assertion substitution, or `assn_sub`. That is, given a proposition `P`, a variable `X`, and an arithmetic expression `a`, we want to derive another proposition `P'` that is just the same as `P` except that `P'` should mention `a` wherever `P` mentions `X`.

Since `P` is an arbitrary Coq assertion, we can't directly "edit" its text. However, we can achieve the same effect by evaluating `P` in an updated state:

```
Definition assn_sub X a P : Assertion :=
  fun (st : state) ⇒
    P (st & { X --> aeval st a }).

Notation "P [ X |-> a ]" := (assn_sub X a P) (at level 10).
```

That is, `P [X |-> a]` stands for an assertion — let's call it `P'` — that is just like `P` except that, wherever `P` looks up the variable `X` in the current state, `P'` instead uses the value of the expression `a`.

To see how this works, let's calculate what happens with a couple of examples. First, suppose `P'` is `(X ≤ 5) [X |-> 3]` — that is, more formally, `P'` is the Coq expression

```
    fun st ⇒
      (fun st' ⇒ st' X ≤ 5)
      (st & { X --> aeval st 3 }),
```

which simplifies to

```
    fun st ⇒
      (fun st' ⇒ st' X ≤ 5)
      (st & { X --> 3 })
```

and further simplifies to

```
    fun st ⇒
      ((st & { X --> 3 }) X) ≤ 5
```

and finally to

```
    fun st ⇒
      3 ≤ 5.
```

That is, `P'` is the assertion that `3` is less than or equal to `5` (as expected).

For a more interesting example, suppose `P'` is `(X ≤ 5) [X |-> X+1]`. Formally, `P'` is the Coq expression

```
    fun st ⇒
      (fun st' ⇒ st' X ≤ 5)
      (st & { X --> aeval st (X+1) }),
```

which simplifies to

```
    fun st ⇒
      (st & { X --> aeval st (X+1) }) X ≤ 5
```

and further simplifies to

```
    fun st ⇒
      (aeval st (X+1)) ≤ 5.
```

That is, `P'` is the assertion that `X+1` is at most `5`.

Now, using the concept of substitution, we can give the precise proof rule for assignment:

$$\frac{}{\{\{Q\ [X \mapsto a]\}\}\ X ::= a\ \{\{Q\}\}}\quad \text{(hoare\_asgn)}$$

We can prove formally that this rule is indeed valid.

```
Theorem hoare_asgn : ∀ Q X a,
  {{Q [X |-> a]}} (X ::= a) {{Q}}.
+
```

Here's a first formal proof using this rule.

```
Example assn_sub_example :
  {{(fun st ⇒ st X < 5) [X |-> X+1]}}
  (X ::= X+1)
  {{fun st ⇒ st X < 5}}.
Proof.
  (* WORKED IN CLASS *)
  apply hoare_asgn. Qed.
```

Of course, what would be even more helpful is to prove this simpler triple:

```
    {{X < 4}} (X ::= X+1) {{X < 5}}
```

We will see how to do so in the next section.

Exercise: 2 stars (hoare_asgn_examples)

Translate these informal Hoare triples...

```
    1) {{ (X ≤ 10) [X |-> 2 * X] }}
       X ::= 2 * X
       {{ X ≤ 10 }}

    2) {{ (0 ≤ X ∧ X ≤ 5) [X |-> 3] }}
       X ::= 3
       {{ 0 ≤ X ∧ X ≤ 5 }}
```

...into formal statements (use the names assn_sub_ex$_1$ and assn_sub_ex$_2$) and use hoare_asgn to prove them.

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars, recommended (hoare_asgn_wrong)

The assignment rule looks backward to almost everyone the first time they see it. If it still seems puzzling, it may help to think a little about alternative "forward" rules. Here is a seemingly natural one:

$$\frac{}{\{\{\ \texttt{True}\ \}\}\ \texttt{X ::= a}\ \{\{\ \texttt{X = a}\ \}\}}\quad \text{(hoare\_asgn\_wrong)}$$

Give a counterexample showing that this rule is incorrect and argue informally that it is really a counterexample. (Hint: The rule universally quantifies over the arithmetic expression a, and your counterexample needs to exhibit an a for which the rule doesn't work.)

```
(* FILL IN HERE *)
```

□

```
Local Close Scope aexp_scope.
```

Exercise: 3 stars, advanced (hoare_asgn_fwd)

However, by using a *parameter* m (a Coq number) to remember the original value of X we can define a Hoare rule for assignment that does, intuitively, "work forwards" rather than backwards.

$$\frac{}{\begin{array}{c}\{\{\texttt{fun st} \Rightarrow \texttt{P st} \wedge \texttt{st X = m}\}\}\\ \texttt{X ::= a}\\ \{\{\texttt{fun st} \Rightarrow \texttt{P st'} \wedge \texttt{st X = aeval st' a}\ \}\}\\ \texttt{(where st' = st \& \{ X --> m \})}\end{array}}\quad \text{(hoare\_asgn\_fwd)}$$

Note that we use the original value of X to reconstruct the state st' before the assignment took place. Prove that this rule is correct. (Also note that this rule is more complicated than hoare_asgn.)

```
Theorem hoare_asgn_fwd :
  ∀ m a P,
  {{fun st ⇒ P st ∧ st X = m}}
    X ::= a
  {{fun st ⇒ P (st & { X --> m })
             ∧ st X = aeval (st & { X --> m }) a }}.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, advanced, optional (hoare_asgn_fwd_exists)

Another way to define a forward rule for assignment is to existentially quantify over the previous value of the assigned variable. Prove that it is correct.

```
    ------------------------------------------------ (hoare_asgn_fwd_exists)
                  {{fun st ⇒ P st}}
                        X ::= a
    {{fun st ⇒ ∃ m, P (st & { X --> m }) ∧
     st X = aeval (st & { X --> m }) a }}
```

```
Theorem hoare_asgn_fwd_exists :
  ∀ a P,
  {{fun st ⇒ P st}}
    X ::= a
  {{fun st ⇒ ∃ m, P (st & { X --> m }) ∧
                 st X = aeval (st & { X --> m }) a }}.
Proof.
  intros a P.
  (* FILL IN HERE *) Admitted.
```

☐

Consequence

Sometimes the preconditions and postconditions we get from the Hoare rules won't quite be the ones we want in the particular situation at hand — they may be logically equivalent but have a different syntactic form that fails to unify with the goal we are trying to prove, or they actually may be logically weaker (for preconditions) or stronger (for postconditions) than what we need.

For instance, while

```
{{(X = 3) [X |-> 3]}} X ::= 3 {{X = 3}},
```

follows directly from the assignment rule,

```
{{True}} X ::= 3 {{X = 3}}
```

does not. This triple is valid, but it is not an instance of `hoare_asgn` because `True` and `(X = 3) [X |-> 3]` are not syntactically equal assertions. However, they are logically *equivalent*, so if one triple is valid, then the other must certainly be as well. We can capture this observation with the following rule:

```
    {{P'}} c {{Q}}
      P <<->> P'
    --------------    (hoare_consequence_pre_equiv)
```

```
{{P}} c {{Q}}
```

Taking this line of thought a bit further, we can see that strengthening the precondition or weakening the postcondition of a valid triple always produces another valid triple. This observation is captured by two *Rules of Consequence*.

```
      {{P'}} c {{Q}}
         P ->> P'
      ---------------          (hoare_consequence_pre)
       {{P}} c {{Q}}

      {{P}} c {{Q'}}
         Q' ->> Q
      ---------------          (hoare_consequence_post)
       {{P}} c {{Q}}
```

Here are the formal versions:

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P ->> P' →
  {{P}} c {{Q}}.
+

Theorem hoare_consequence_post : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' ->> Q →
  {{P}} c {{Q}}.
+
```

For example, we can use the first consequence rule like this:

```
                {{ True }} ->>
                {{ 1 = 1 }}
  X ::= 1
                {{ X = 1 }}
```

Or, formally...

```
Example hoare_asgn_example1 :
  {{fun st ⇒ True}} (X ::= 1) {{fun st ⇒ st X = 1}}.
Proof.
  (* WORKED IN CLASS *)
  apply hoare_consequence_pre
    with (P' := (fun st ⇒ st X = 1) [X |-> 1]).
  apply hoare_asgn.
  intros st H. unfold assn_sub, t_update. simpl. reflexivity.
Qed.
```

We can also use it to prove the example mentioned earlier.

```
                {{ X < 4 }} ->>
                {{ (X < 5)[X |-> X+1] }}
  X ::= X + 1
                {{ X < 5 }}
```

Or, formally ...

```
Example assn_sub_example2 :
  {{(fun st ⇒ st X < 4)}}
  (X ::= X+1)
  {{fun st ⇒ st X < 5}}.
Proof.
  (* WORKED IN CLASS *)
  apply hoare_consequence_pre
    with (P' := (fun st ⇒ st X < 5) [X |-> X+1]).
  apply hoare_asgn.
  intros st H. unfold assn_sub, t_update. simpl. omega.
Qed.
```

Finally, for convenience in proofs, here is a combined rule of consequence that allows us to vary both the precondition and the postcondition in one go.

```
          {{P'}} c {{Q'}}
             P ->> P'
             Q' ->> Q
         -----------------     (hoare_consequence)
           {{P}} c {{Q}}
```

```
Theorem hoare_consequence : ∀ (P P' Q Q' : Assertion) c,
  {{P'}} c {{Q'}} →
  P ->> P' →
  Q' ->> Q →
  {{P}} c {{Q}}.
```

+

Digression: The `eapply` Tactic

This is a good moment to take another look at the `eapply` tactic, which we introduced briefly in the Auto chapter of *Logical Foundations*.

We had to write "`with (P' := ...)`" explicitly in the proof of `hoare_asgn_example1` and `hoare_consequence` above, to make sure that all of the metavariables in the premises to the `hoare_consequence_pre` rule would be set to specific values. (Since `P'` doesn't appear in the conclusion of `hoare_consequence_pre`, the process of unifying the conclusion with the current goal doesn't constrain `P'` to a specific assertion.)

This is annoying, both because the assertion is a bit long and also because, in `hoare_asgn_example1`, the very next thing we are going to do — applying the `hoare_asgn` rule — will tell us exactly what it should be! We can use `eapply` instead of `apply` to tell Coq, essentially, "Be patient: The missing part is going to be filled in later in the proof."

```
Example hoare_asgn_example1' :
  {{fun st ⇒ True}}
  (X ::= 1)
  {{fun st ⇒ st X = 1}}.
Proof.
  eapply hoare_consequence_pre.
```

```
    apply hoare_asgn.
    intros st H. reflexivity. Qed.
```

In general, `eapply H` tactic works just like `apply H` except that, instead of failing if unifying the goal with the conclusion of `H` does not determine how to instantiate all of the variables appearing in the premises of `H`, `eapply H` will replace these variables with *existential variables* (written `?nnn`), which function as placeholders for expressions that will be determined (by further unification) later in the proof.

In order for `Qed` to succeed, all existential variables need to be determined by the end of the proof. Otherwise Coq will (rightly) refuse to accept the proof. Remember that the Coq tactics build proof objects, and proof objects containing existential variables are not complete.

```
Lemma silly1 : ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∀ x y : nat, P x y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. eapply HQ. apply HP.
```

Coq gives a warning after `apply HP`. ("All the remaining goals are on the shelf," means that we've finished all our top-level proof obligations but along the way we've put some aside to be done later, and we have not finished those.) Trying to close the proof with `Qed` gives an error.

```
Abort.
```

An additional constraint is that existential variables cannot be instantiated with terms containing ordinary variables that did not exist at the time the existential variable was created. (The reason for this technical restriction is that allowing such instantiation would lead to inconsistency of Coq's logic.)

```
Lemma silly2 :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. eapply HQ. destruct HP as [y HP'].
```

Doing `apply HP'` above fails with the following error:

```
    Error: Impossible to unify "?175" with "y".
```

In this case there is an easy fix: doing `destruct HP` *before* doing `eapply HQ`.

```
Abort.

Lemma silly2_fixed :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
```

```
Proof.
  intros P Q HP HQ. destruct HP as [y HP'].
  eapply HQ. apply HP'.
Qed.
```

The `apply HP'` in the last step unifies the existential variable in the goal with the variable `y`.

Note that the `assumption` tactic doesn't work in this case, since it cannot handle existential variables. However, Coq also provides an `eassumption` tactic that solves the goal if one of the premises matches the goal up to instantiations of existential variables. We can use it instead of `apply HP'` if we like.

```
Lemma silly2_eassumption : ∀ (P : nat → nat → Prop) (Q : nat →
Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. destruct HP as [y HP']. eapply HQ.
eassumption.
Qed.
```

Exercise: 2 stars (hoare_asgn_examples_2)

Translate these informal Hoare triples...

```
{{ X + 1 ≤ 5 }}  X ::= X + 1  {{ X ≤ 5 }}
{{ 0 ≤ 3 ∧ 3 ≤ 5 }}  X ::= 3  {{ 0 ≤ X ∧ X ≤ 5 }}
```

...into formal statements (name them `assn_sub_ex`$_1$`'` and `assn_sub_ex`$_2$`'`) and use `hoare_asgn` and `hoare_consequence_pre` to prove them.

```
(* FILL IN HERE *)
```

□

Skip

Since `SKIP` doesn't change the state, it preserves any assertion `P`:

$$\frac{}{\{\{\ P\ \}\}\ \texttt{SKIP}\ \{\{\ P\ \}\}}\quad \text{(hoare\_skip)}$$

```
Theorem hoare_skip : ∀ P,
     {{P}} SKIP {{P}}.
```

+

Sequencing

More interestingly, if the command c_1 takes any state where `P` holds to a state where `Q` holds, and if c_2 takes any state where `Q` holds to one where `R` holds, then doing c_1 followed by c_2 will take any state where `P` holds to one where `R` holds:

$$\{\{\ P\ \}\}\ c_1\ \{\{\ Q\ \}\}$$

$$\frac{\{\!\{\ Q\ \}\!\}\ c_2\ \{\!\{\ R\ \}\!\}}{\{\!\{\ P\ \}\!\}\ c_1;;c_2\ \{\!\{\ R\ \}\!\}}\quad \text{(hoare\_seq)}$$

```
Theorem hoare_seq : ∀ P Q R c₁ c₂,
     {{Q}} c₂ {{R}} →
     {{P}} c₁ {{Q}} →
     {{P}} c₁;;c₂ {{R}}.
```

+

Note that, in the formal rule `hoare_seq`, the premises are given in backwards order (c_2 before c_1). This matches the natural flow of information in many of the situations where we'll use the rule, since the natural way to construct a Hoare-logic proof is to begin at the end of the program (with the final postcondition) and push postconditions backwards through commands until we reach the beginning.

Informally, a nice way of displaying a proof using the sequencing rule is as a "decorated program" where the intermediate assertion Q is written between c_1 and c_2:

```
      {{ a = n }}
    X ::= a;;
      {{ X = n }}    <---- decoration for Q
    SKIP
      {{ X = n }}
```

Here's an example of a program involving both assignment and sequencing.

```
Example hoare_asgn_example3 : ∀ a n,
  {{fun st ⇒ aeval st a = n}}
  (X ::= a;; SKIP)
  {{fun st ⇒ st X = n}}.
Proof.
  intros a n. eapply hoare_seq.
  - (* right part of seq *)
    apply hoare_skip.
  - (* left part of seq *)
    eapply hoare_consequence_pre. apply hoare_asgn.
    intros st H. subst. reflexivity.
Qed.
```

We typically use `hoare_seq` in conjunction with `hoare_consequence_pre` and the `eapply` tactic, as in this example.

Exercise: 2 stars, recommended (hoare_asgn_example4)

Translate this "decorated program" into a formal proof:

```
                   {{ True }} ->>
                   {{ 1 = 1 }}
    X ::= 1;;
```

```
                                    {{ X = 1 }} ->>
                                    {{ X = 1 ∧ 2 = 2 }}
      Y ::= 2
                                    {{ X = 1 ∧ Y = 2 }}
```

(Note the use of "->>" decorations, each marking a use of `hoare_consequence_pre`.)

```
Example hoare_asgn_example4 :
  {{fun st ⇒ True}} (X ::= 1;; Y ::= 2)
  {{fun st ⇒ st X = 1 ∧ st Y = 2}}.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (swap_exercise)

Write an Imp program `c` that swaps the values of `X` and `Y` and show that it satisfies the following specification:

```
      {{X ≤ Y}} c {{Y ≤ X}}
```

Your proof should not need to use `unfold hoare_triple`.

```
Definition swap_program : com
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Theorem swap_exercise :
  {{fun st ⇒ st X ≤ st Y}}
  swap_program
  {{fun st ⇒ st Y ≤ st X}}.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (hoarestate1)

Explain why the following proposition can't be proven:

```
      ∀ (a : aexp) (n : nat),
         {{fun st ⇒ aeval st a = n}}
           (X ::= 3;; Y ::= a)
         {{fun st ⇒ st Y = n}}.
```

```
(* FILL IN HERE *)
```

□

Conditionals

What sort of rule do we want for reasoning about conditional commands?

Certainly, if the same assertion `Q` holds after executing either of the branches, then it holds after the whole conditional. So we might be tempted to write:

$$\{\{P\}\}\ c_1\ \{\{Q\}\}$$
$$\{\{P\}\}\ c_2\ \{\{Q\}\}$$

```
{{P}} IFB b THEN c₁ ELSE c₂ {{Q}}
```

However, this is rather weak. For example, using this rule, we cannot show

```
{{ True }}
IFB X = 0
THEN Y ::= 2
ELSE Y ::= X + 1
FI
{{ X ≤ Y }}
```

since the rule tells us nothing about the state in which the assignments take place in the "then" and "else" branches.

Fortunately, we can say something more precise. In the "then" branch, we know that the boolean expression `b` evaluates to `true`, and in the "else" branch, we know it evaluates to `false`. Making this information available in the premises of the rule gives us more information to work with when reasoning about the behavior of c_1 and c_2 (i.e., the reasons why they establish the postcondition `Q`).

$$\frac{\{\{P \land b\}\}\ c_1\ \{\{Q\}\} \qquad \{\{P \land \sim b\}\}\ c_2\ \{\{Q\}\}}{\{\{P\}\}\ \texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}\ \{\{Q\}\}} \quad \text{(hoare\_if)}$$

To interpret this rule formally, we need to do a little work. Strictly speaking, the assertion we've written, `P ∧ b`, is the conjunction of an assertion and a boolean expression — i.e., it doesn't typecheck. To fix this, we need a way of formally "lifting" any bexp `b` to an assertion. We'll write `bassn b` for the assertion "the boolean expression `b` evaluates to `true` (in the given state)."

```
Definition bassn b : Assertion :=
  fun st ⇒ (beval st b = true).
```

A couple of useful facts about `bassn`:

```
Lemma bexp_eval_true : ∀ b st,
  beval st b = true → (bassn b) st.
+

Lemma bexp_eval_false : ∀ b st,
  beval st b = false → ¬ ((bassn b) st).
+
```

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

```
Theorem hoare_if : ∀ P Q b c₁ c₂,
  {{fun st ⇒ P st ∧ bassn b st}} c₁ {{Q}} →
  {{fun st ⇒ P st ∧ ~(bassn b st)}} c₂ {{Q}} →
  {{P}} (IFB b THEN c₁ ELSE c₂ FI) {{Q}}.
+
```

Example

Here is a formal proof that the program we used to motivate the rule satisfies the specification we gave.

```
Example if_example :
    {{fun st ⇒ True}}
  IFB X = 0
    THEN Y ::= 2
    ELSE Y ::= X + 1
  FI
    {{fun st ⇒ st X ≤ st Y}}.
+
```

Exercise: 2 stars (if_minus_plus)

Prove the following hoare triple using `hoare_if`. Do not use `unfold hoare_triple`.

```
Theorem if_minus_plus :
  {{fun st ⇒ True}}
  IFB X ≤ Y
    THEN Z ::= Y - X
    ELSE Y ::= X + Z
  FI
  {{fun st ⇒ st Y = st X + st Z}}.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: One-sided conditionals

Exercise: 4 stars (if1_hoare)

In this exercise we consider extending Imp with "one-sided conditionals" of the form `IF`$_1$ `b THEN c FI`. Here `b` is a boolean expression, and `c` is a command. If `b` evaluates to `true`, then command `c` is evaluated. If `b` evaluates to `false`, then `IF`$_1$ `b THEN c FI` does nothing.

We recommend that you complete this exercise before attempting the ones that follow, as it should help solidify your understanding of the material.

The first step is to extend the syntax of commands and introduce the usual notations. (We've done this for you. We use a separate module to prevent polluting the global name space.)

```
Module If1.

Inductive com : Type :=
  | CSkip : com
  | CAss : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CIf1 : bexp → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAss X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'IF1' b 'THEN' c 'FI'" :=
  (CIf1 b c) (at level 80, right associativity).
```

Next we need to extend the evaluation relation to accommodate `IF1` branches. This is for you to do... What rule(s) need to be added to `ceval` to evaluate one-sided conditionals?

```
Reserved Notation "c1 '/' st '\\' st'" (at level 40, st at level
39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st : state, SKIP / st \\ st
  | E_Ass : ∀ (st : state) (a1 : aexp) (n : nat) (X : string),
            aeval st a1 = n → (X ::= a1) / st \\ st & { X --> n }
  | E_Seq : ∀ (c1 c2 : com) (st st' st'' : state),
            c1 / st \\ st' → c2 / st' \\ st'' → (c1 ;; c2) / st
\\ st''
  | E_IfTrue : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
               beval st b1 = true →
               c1 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st
\\ st'
  | E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
                beval st b1 = false →
                c2 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st
\\ st'
  | E_WhileFalse : ∀ (b1 : bexp) (st : state) (c1 : com),
                   beval st b1 = false → (WHILE b1 DO c1 END) / st
\\ st
  | E_WhileTrue : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
                  beval st b1 = true →
                  c1 / st \\ st' →
                  (WHILE b1 DO c1 END) / st' \\ st'' →
                  (WHILE b1 DO c1 END) / st \\ st''
(* FILL IN HERE *)

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

Now we repeat (verbatim) the definition and notation of Hoare triples.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) :
Prop :=
  ∀ st st',
       c / st \\ st' →
       P st →
       Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
                                  (at level 90, c at next level)
                                  : hoare_spec_scope.
```

Finally, we (i.e., you) need to state and prove a theorem, `hoare_if₁`, that expresses an appropriate Hoare logic proof rule for one-sided conditionals. Try to come up with a rule that is both sound and as precise as possible.

```
(* FILL IN HERE *)
```

For full credit, prove formally `hoare_if1_good` that your rule is precise enough to show the following valid Hoare triple:

```
{{ X + Y = Z }}
IF₁ !(Y = 0) THEN
  X ::= X + Y
FI
{{ X = Z }}
```

Hint: Your proof of this triple may need to use the other proof rules also. Because we're working in a separate module, you'll need to copy here the rules you find necessary.

```
Lemma hoare_if1_good :
  {{ fun st ⇒ st X + st Y = st Z }}
  IF₁ !(Y = 0) THEN
    X ::= X + Y
  FI
  {{ fun st ⇒ st X = st Z }}.
Proof. (* FILL IN HERE *) Admitted.

End If₁.
```

☐

Loops

Finally, we need a rule for reasoning about while loops.

Suppose we have a loop

```
      WHILE b DO c END
```

and we want to find a pre-condition `P` and a post-condition `Q` such that

```
      {{P}} WHILE b DO c END {{Q}}
```

is a valid triple.

First of all, let's think about the case where b is false at the beginning — i.e., let's assume that the loop body never executes at all. In this case, the loop behaves like SKIP, so we might be tempted to write:

```
      {{P}} WHILE b DO c END {{P}}.
```

But, as we remarked above for the conditional, we know a little more at the end — not just P, but also the fact that b is false in the current state. So we can enrich the postcondition a little:

```
      {{P}} WHILE b DO c END {{P ∧ ¬b}}
```

What about the case where the loop body *does* get executed? In order to ensure that P holds when the loop finally exits, we certainly need to make sure that the command c guarantees that P holds whenever c is finished. Moreover, since P holds at the beginning of the first execution of c, and since each execution of c re-establishes P when it finishes, we can always assume that P holds at the beginning of c. This leads us to the following rule:

$$\frac{\{\{P\}\}\ c\ \{\{P\}\}}{\{\{P\}\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{\{P \land \sim b\}\}}$$

This is almost the rule we want, but again it can be improved a little: at the beginning of the loop body, we know not only that P holds, but also that the guard b is true in the current state.

This gives us a little more information to use in reasoning about c (showing that it establishes the invariant by the time it finishes). This gives us the final version of the rule:

$$\frac{\{\{P \land b\}\}\ c\ \{\{P\}\}}{\{\{P\}\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{\{P \land \sim b\}\}} \quad \text{(hoare\_while)}$$

The proposition P is called an *invariant* of the loop.

```
Theorem hoare_while : ∀ P b c,
  {{fun st ⇒ P st ∧ bassn b st}} c {{P}} →
  {{P}} WHILE b DO c END {{fun st ⇒ P st ∧ ¬ (bassn b st)}}.
+
```

One subtlety in the terminology is that calling some assertion P a "loop invariant" doesn't just mean that it is preserved by the body of the loop in question (i.e., {{P}} c {{P}}, where c is the loop body), but rather that P *together with the fact that the loop's guard is true* is a sufficient precondition for c to ensure P as a postcondition.

This is a slightly (but importantly) weaker requirement. For example, if `P` is the assertion `X = 0`, then `P` *is* an invariant of the loop

```
WHILE X = 2 DO X := 1 END
```

although it is clearly *not* preserved by the body of the loop.

```
Example while_example :
    {{fun st ⇒ st X ≤ 3}}
  WHILE X ≤ 2
  DO X ::= X + 1 END
    {{fun st ⇒ st X = 3}}.
+
```

We can use the WHILE rule to prove the following Hoare triple...

```
Theorem always_loop_hoare : ∀ P Q,
  {{P}} WHILE true DO SKIP END {{Q}}.
+
```

Of course, this result is not surprising if we remember that the definition of `hoare_triple` asserts that the postcondition must hold *only* when the command terminates. If the command doesn't terminate, we can prove anything we like about the post-condition.

Hoare rules that only talk about what happens when commands terminate (without proving that they do) are often said to describe a logic of "partial" correctness. It is also possible to give Hoare rules for "total" correctness, which build in the fact that the commands terminate. However, in this course we will only talk about partial correctness.

Exercise: `REPEAT`

Exercise: 4 stars, advanced (hoare_repeat)

In this exercise, we'll add a new command to our language of commands: `REPEAT` c `UNTIL` a `END`. You will write the evaluation rule for `repeat` and add a new Hoare rule to the language for programs involving it. (You may recall that the evaluation rule is given in an example in the Auto chapter. Try to figure it out yourself here rather than peeking.)

```
Module RepeatExercise.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CRepeat : com → bexp → com.
```

`REPEAT` behaves like `WHILE`, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*.

Because of this, the body will always execute at least once.

```
Notation "'SKIP'" :=
  CSkip.
Notation "c₁ ;; c₂" :=
  (CSeq c₁ c₂) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e₁ 'THEN' e₂ 'ELSE' e₃ 'FI'" :=
  (CIf e₁ e₂ e₃) (at level 80, right associativity).
Notation "'REPEAT' e₁ 'UNTIL' b₂ 'END'" :=
  (CRepeat e₁ b₂) (at level 80, right associativity).
```

Add new rules for `REPEAT` to `ceval` below. You can use the rules for `WHILE` as a guide, but remember that the body of a `REPEAT` should always execute at least once, and that the loop ends when the guard becomes true.

```
Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      ceval st SKIP st
  | E_Ass : ∀ st a₁ n X,
      aeval st a₁ = n →
      ceval st (X ::= a₁) (st & { X --> n })
  | E_Seq : ∀ c₁ c₂ st st' st'',
      ceval st c₁ st' →
      ceval st' c₂ st'' →
      ceval st (c₁ ;; c₂) st''
  | E_IfTrue : ∀ st st' b₁ c₁ c₂,
      beval st b₁ = true →
      ceval st c₁ st' →
      ceval st (IFB b₁ THEN c₁ ELSE c₂ FI) st'
  | E_IfFalse : ∀ st st' b₁ c₁ c₂,
      beval st b₁ = false →
      ceval st c₂ st' →
      ceval st (IFB b₁ THEN c₁ ELSE c₂ FI) st'
  | E_WhileFalse : ∀ b₁ st c₁,
      beval st b₁ = false →
      ceval st (WHILE b₁ DO c₁ END) st
  | E_WhileTrue : ∀ st st' st'' b₁ c₁,
      beval st b₁ = true →
      ceval st c₁ st' →
      ceval st' (WHILE b₁ DO c₁ END) st'' →
      ceval st (WHILE b₁ DO c₁ END) st''
(* FILL IN HERE *)
.
```

A couple of definitions from above, copied here so they use the new `ceval`.

```
Notation "c₁ '/' st '\\' st'" := (ceval st c₁ st')
                                  (at level 40, st at level 39).

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion)
                        : Prop :=
  ∀ st st', (c / st \\ st') → P st → Q st'.

Notation "{{ P }} c {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level).
```

To make sure you've got the evaluation rules for `REPEAT` right, prove that `ex1_repeat` evaluates correctly.

```
Definition ex1_repeat :=
  REPEAT
    X ::= 1;;
    Y ::= Y + 1
  UNTIL X = 1 END.

Theorem ex1_repeat_works :
  ex1_repeat / { --> 0 } \\ { X --> 1 ; Y --> 1 }.
Proof.
  (* FILL IN HERE *) Admitted.
```

Now state and prove a theorem, `hoare_repeat`, that expresses an appropriate proof rule for `repeat` commands. Use `hoare_while` as a model, and try to make your rule as precise as possible.

```
(* FILL IN HERE *)
```

For full credit, make sure (informally) that your rule can be used to prove the following valid Hoare triple:

```
{{ X > 0 }}
REPEAT
  Y ::= X;;
  X ::= X - 1
UNTIL X = 0 END
{{ X = 0 ∧ Y > 0 }}

End RepeatExercise.
```

□

Summary

So far, we've introduced Hoare Logic as a tool for reasoning about Imp programs. The rules of Hoare Logic are:

$$\frac{}{\{\{Q\ [X\ \mapsto\ a]\}\}\ X::=a\ \{\{Q\}\}}\ \text{(hoare\_asgn)}$$

$$\{\!\{\ P\ \}\!\}\ \texttt{SKIP}\ \{\!\{\ P\ \}\!\} \qquad \text{(hoare\_skip)}$$

$$\frac{\{\!\{\ P\ \}\!\}\ c_1\ \{\!\{\ Q\ \}\!\} \qquad \{\!\{\ Q\ \}\!\}\ c_2\ \{\!\{\ R\ \}\!\}}{\{\!\{\ P\ \}\!\}\ c_1;;c_2\ \{\!\{\ R\ \}\!\}} \qquad \text{(hoare\_seq)}$$

$$\frac{\{\!\{P \land\ b\}\!\}\ c_1\ \{\!\{Q\}\!\} \qquad \{\!\{P \land \sim b\}\!\}\ c_2\ \{\!\{Q\}\!\}}{\{\!\{P\}\!\}\ \texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}\ \{\!\{Q\}\!\}} \qquad \text{(hoare\_if)}$$

$$\frac{\{\!\{P \land b\}\!\}\ c\ \{\!\{P\}\!\}}{\{\!\{P\}\!\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{\!\{P \land \sim b\}\!\}} \qquad \text{(hoare\_while)}$$

$$\frac{\{\!\{P'\}\!\}\ c\ \{\!\{Q'\}\!\} \qquad P \twoheadrightarrow P' \qquad Q' \twoheadrightarrow Q}{\{\!\{P\}\!\}\ c\ \{\!\{Q\}\!\}} \qquad \text{(hoare\_consequence)}$$

In the next chapter, we'll see how these rules are used to prove that programs satisfy specifications of their behavior.

Additional Exercises

Exercise: 3 stars (hoare_havoc)

In this exercise, we will derive proof rules for a HAVOC command, which is similar to the nondeterministic any expression from the the Imp chapter.

First, we enclose this work in a separate module, and recall the syntax and big-step semantics of Himp commands.

```
Module Himp.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : string → com.

Notation "'SKIP'" :=
  CSkip.
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e₁ e₂ e₃) (at level 80, right associativity).
Notation "'HAVOC' X" := (CHavoc X) (at level 60).

Reserved Notation "c₁ '/' st '\\' st'" (at level 40, st at level
39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st : state, SKIP / st \\ st
  | E_Ass : ∀ (st : state) (a₁ : aexp) (n : nat) (X : string),
            aeval st a₁ = n → (X ::= a₁) / st \\ st & { X --> n }
  | E_Seq : ∀ (c₁ c₂ : com) (st st' st'' : state),
            c₁ / st \\ st' → c₂ / st' \\ st'' → (c₁ ;; c₂) / st
\\ st''
  | E_IfTrue : ∀ (st st' : state) (b₁ : bexp) (c₁ c₂ : com),
               beval st b₁ = true →
               c₁ / st \\ st' → (IFB b₁ THEN c₁ ELSE c₂ FI) / st
\\ st'
  | E_IfFalse : ∀ (st st' : state) (b₁ : bexp) (c₁ c₂ : com),
                beval st b₁ = false →
                c₂ / st \\ st' → (IFB b₁ THEN c₁ ELSE c₂ FI) / st
\\ st'
  | E_WhileFalse : ∀ (b₁ : bexp) (st : state) (c₁ : com),
                   beval st b₁ = false → (WHILE b₁ DO c₁ END) / st
\\ st
  | E_WhileTrue : ∀ (st st' st'' : state) (b₁ : bexp) (c₁ : com),
                  beval st b₁ = true →
                  c₁ / st \\ st' →
                  (WHILE b₁ DO c₁ END) / st' \\ st'' →
                  (WHILE b₁ DO c₁ END) / st \\ st''
  | E_Havoc : ∀ (st : state) (X : string) (n : nat),
              (HAVOC X) / st \\ st & { X --> n }

  where "c₁ '/' st '\\' st'" := (ceval c₁ st st').
```

The definition of Hoare triples is exactly as before.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) :
Prop :=
  ∀ st st', c / st \\ st' → P st → Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
                                  (at level 90, c at next level)
                                  : hoare_spec_scope.
```

Complete the Hoare rule for `HAVOC` commands below by defining `havoc_pre` and prove that the resulting rule is correct.

```
Definition havoc_pre (X : string) (Q : Assertion) : Assertion
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Theorem hoare_havoc : ∀ (Q : Assertion) (X : string),
  {{ havoc_pre X Q }} HAVOC X {{ Q }}.
```

```
Proof.
  (* FILL IN HERE *) Admitted.

End Himp.
```

□