SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

IMP

SIMPLE IMPERATIVE PROGRAMS

In this chapter, we'll take a more serious look at how to use Coq to study interesting things outside of itself. Our case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java. Here is a familiar mathematical function written in Imp.

```
Z ::= X;;
Y ::= 1;;
WHILE ! (Z = 0) DO
  Y ::= Y * Z;;
  Z ::= Z - 1
END
```

This chapter looks at how to define the *syntax* and *semantics* of Imp; further chapters in *Programming Language Foundations* (*Software Foundations*, volume 2) develop a theory of *program equivalence* and introduce *Hoare Logic*, a widely used logic for reasoning about imperative programs.

```
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 Coq.Lists.List.
Require Import Coq.omega.Omega.
Import ListNotations.

Require Import Maps.
```

Arithmetic and Boolean Expressions

We'll present Imp in three parts: first a core language of *arithmetic and boolean expressions*, then an extension of these expressions with *variables*, and finally a

language of *commands* including assignment, conditions, sequencing, and loops.

Syntax

```
Module AExp.
```

These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp.
```

In this chapter, we'll mostly elide the translation from the concrete syntax that a programmer would actually write to these abstract syntax trees — the process that, for example, would translate the string `"1+2*3"` to the AST

```
    APlus (ANum 1) (AMult (ANum 2) (ANum 3)).
```

The optional chapter ImpParser develops a simple implementation of a lexical analyzer and parser that can perform this translation. You do *not* need to understand that chapter to understand this one, but if you haven't taken a course where these techniques are covered (e.g., a compilers course) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:

```
a ::= nat
    | a + a
    | a - a
    | a * a

b ::= true
    | false
    | a = a
    | a ≤ a
    | not b
    | b and b
```

Compared to the Coq version above...

- The BNF is more informal — for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written `+` and is an infix symbol) while leaving other aspects of lexical analysis and parsing (like the relative precedence of `+`, `-`, and `*`, the use of parens to explicitly group subexpressions, etc.) unspecified. Some additional information (and human intelligence) would be required to turn this description into a formal definition, for example when implementing a compiler.

  The Coq version consistently omits all this information and concentrates on the abstract syntax only.

- On the other hand, the BNF version is lighter and easier to read. Its informality makes it flexible, a big advantage in situations like discussions at the blackboard, where conveying general ideas is more important than getting every detail nailed down precisely.

  Indeed, there are dozens of BNF-like notations and people switch freely among them, usually without bothering to say which form of BNF they're using because there is no need to: a rough-and-ready informal understanding is all that's important.

It's good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

Evaluation

Evaluating an arithmetic expression produces a number.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) * (aeval a2)
  end.

Example test_aeval1:
  aeval (APlus (ANum 2) (ANum 2)) = 4.
+
```

Similarly, evaluating a boolean expression yields a boolean.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval a1) (aeval a2)
  | BLe a1 a2 ⇒ leb (aeval a1) (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

Optimization

We haven't defined very much yet, but we can already get some mileage out of the definitions. Suppose we define a function that takes an arithmetic expression and slightly simplifies it, changing every occurrence of `0+e` (i.e., `(APlus (ANum 0) e)` into just `e`.

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒
      ANum n
  | APlus (ANum 0) e2 ⇒
      optimize_0plus e2
  | APlus e1 e2 ⇒
      APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒
      AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒
      AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

To make sure our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

```
Example test_optimize_0plus:
  optimize_0plus (APlus (ANum 2)
                        (APlus (ANum 0)
                               (APlus (ANum 0) (ANum 1))))
  = APlus (ANum 2) (ANum 1).
+
```

But if we want to be sure the optimization is correct — i.e., that evaluating an optimized expression gives the same result as the original — we should prove it.

```
Theorem optimize_0plus_sound: ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - (* ANum *) reflexivity.
  - (* APlus *) destruct a1.
    + (* a1 = ANum n *) destruct n.
      * (* n = 0 *) simpl. apply IHa2.
      * (* n <> 0 *) simpl. rewrite IHa2. reflexivity.
    + (* a1 = APlus a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    + (* a1 = AMinus a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    + (* a1 = AMult a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
```

```
  - (* AMinus *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.
  - (* AMult *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.
```

Coq Automation

The amount of repetition in this last proof is a little annoying. And if either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would start to be a real problem.

So far, we've been doing all our proofs using just a small handful of Coq's tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy — Coq's automation is a power tool — but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

Tacticals

Tacticals is Coq's term for tactics that take other tactics as arguments — "higher-order tactics," if you will.

The `try` Tactical

If `T` is a tactic, then `try T` is a tactic that is just like `T` except that, if `T` fails, `try T` *successfully* does nothing at all (instead of failing).

```
Theorem silly1 : ∀ ae, aeval ae = aeval ae.
Proof. try reflexivity. (* this just does reflexivity *) Qed.

Theorem silly2 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  try reflexivity. (* just reflexivity would have failed *)
  apply HP.
(* we can still finish the proof in some other way *)
Qed.
```

There is no real reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the `;` tactical, which we show next.

The `;` Tactical (Simple Form)

In its most common form, the `;` tactical takes two tactics as arguments. The compound tactic `T;T'` first performs `T` and then performs `T'` on *each subgoal* generated by `T`.

For example, consider the following trivial lemma:

```
Lemma foo : ∀ n, leb 0 n = true.
Proof.
  intros.
  destruct n.
    (* Leaves two subgoals, which are discharged identically...  *)
    - (* n=0 *) simpl. reflexivity.
    - (* n=Sn' *) simpl. reflexivity.
Qed.
```

We can simplify this proof using the `;` tactical:

```
Lemma foo' : ∀ n, leb 0 n = true.
Proof.
  intros.
  (* destruct the current goal *)
  destruct n;
  (* then simpl each resulting subgoal *)
  simpl;
  (* and do reflexivity on each resulting subgoal *)
  reflexivity.
Qed.
```

Using `try` and `;` together, we can get rid of the repetition in the proof that was bothering us a little while ago.

```
Theorem optimize_0plus_sound': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    (* Most cases follow directly by the IH... *)
    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
    (* ... but the remaining cases -- ANum and APlus --
       are different: *)
  - (* ANum *) reflexivity.
  - (* APlus *)
    destruct a1;
      (* Again, most cases follow directly by the IH: *)
      try (simpl; simpl in IHa1; rewrite IHa1;
           rewrite IHa2; reflexivity).
    (* The interesting case, on which the try...
       does nothing, is when e1 = ANum n. In this
       case, we have to destruct n (to see whether
       the optimization applies) and rewrite with the
       induction hypothesis. *)
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; reflexivity. Qed.
```

Coq experts often use this "`...; try...` " idiom after a tactic like `induction` to take care of many similar cases all at once. Naturally, this practice has an analog in informal proofs. For example, here is an informal proof of the optimization theorem that matches the structure of the formal one:

Theorem: For all arithmetic expressions a,

```
aeval (optimize_0plus a) = aeval a.
```

Proof: By induction on a. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose a = ANum n for some n. We must show

  ```
  aeval (optimize_0plus (ANum n)) = aeval (ANum n).
  ```

  This is immediate from the definition of optimize_0plus.

- Suppose a = APlus a_1 a_2 for some a_1 and a_2. We must show

  ```
  aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2).
  ```

  Consider the possible forms of a_1. For most of them, optimize_0plus simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as a_1; in these cases, the result follows directly from the IH.

  The interesting case is when a_1 = ANum n for some n. If n = 0, then

  ```
  optimize_0plus (APlus a1 a2) = optimize_0plus a2
  ```

  and the IH for a_2 is exactly what we need. On the other hand, if n = S n' for some n', then again optimize_0plus simply calls itself recursively, and the result follows from the IH. □

However, this proof can still be improved: the first case (for a = ANum n) is very trivial — even more trivial than the cases that we said simply followed from the IH — yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, "Most cases are either immediate or direct from the IH. The only interesting case is the one for APlus..." We can make the same improvement in our formal proof too. Here's how it looks:

```
Theorem optimize_0plus_sound'': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    (* Most cases follow directly by the IH *)
    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);
    (* ... or are immediate by definition *)
    try reflexivity.
  (* The interesting case is when a = APlus a1 a2. *)
  - (* APlus *)
    destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
                      rewrite IHa2; reflexivity).
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; reflexivity. Qed.
```

The ; Tactical (General Form)

The `;` tactical also has a more general form than the simple `T;T'` we've seen above. If `T`, T_1, ..., `Tn` are tactics, then

```
T; [T1 | T2 | ... | Tn]
```

is a tactic that first performs `T` and then performs T_1 on the first subgoal generated by `T`, performs T_2 on the second subgoal, etc.

So `T;T'` is just special notation for the case when all of the `Ti`'s are the same tactic; i.e., `T;T'` is shorthand for:

```
T; [T' | T' | ... | T']
```

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing that `10` is in a long list using repeat.

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

The tactic `repeat T` never fails: if the tactic `T` doesn't apply to the original goal, then repeat still succeeds without changing the original goal (i.e., it repeats zero times).

```
Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (left; reflexivity).
  repeat (right; try (left; reflexivity)).
Qed.
```

The tactic `repeat T` also does not have any upper bound on the number of times it applies `T`. If `T` is a tactic that always succeeds, then repeat `T` will loop forever (e.g., `repeat simpl` loops, since `simpl` always succeeds). While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of `repeat` and other tactics is to guide Coq in constructing proofs; if the construction process diverges, this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

Exercise: 3 stars (optimize_0plus_b_sound)

anyone wrote it with subst?

Since the `optimize_0plus` transformation doesn't change the value of `aexps`, we should be able to apply it to all the `aexps` that appear in a `bexp` without changing the `bexp`'s value. Write a function that performs this transformation on `bexps` and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

```
Fixpoint optimize_0plus_b (b : bexp) : bexp
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
```

```
Admitted.

Theorem optimize_0plus_b_sound : ∀ b,
  beval (optimize_0plus_b b) = beval b.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, optional (optimizer)

Design exercise: The optimization implemented by our `optimize_0plus` function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small — add just a single, simple optimization and prove it correct — and build up to something more interesting incrementally.)

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

□

Defining New Tactic Notations

Coq also provides several ways of "programming" tactic scripts.

- The `Tactic Notation` idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.
- For more sophisticated programming, Coq offers a built-in language called `Ltac` with primitives that can examine and modify the proof state. The details are a bit too complicated to get into here (and it is generally agreed that `Ltac` is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq, and there are many examples of `Ltac` definitions in the Coq standard library that you can use as examples.
- There is also an OCaml API, which can be used to build tactics that access Coq's internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

The `Tactic Notation` mechanism is the easiest to come to grips with, and it offers plenty of power for many purposes. Here's an example.

```
Tactic Notation "simpl_and_try" tactic(c) :=
  simpl;
  try c.
```

This defines a new tactical called `simpl_and_try` that takes one tactic `c` as an argument and is defined to be equivalent to the tactic `simpl; try c`. Now writing "`simpl_and_try reflexivity.`" in a proof will be the same as writing "`simpl; try reflexivity.`"

The `omega` Tactic

The `omega` tactic implements a decision procedure for a subset of first-order logic called *Presburger arithmetic*. It is based on the Omega algorithm invented by William Pugh [Pugh 1991].

If the goal is a universally quantified formula made out of

- numeric constants, addition (`+` and `S`), subtraction (`-` and `pred`), and multiplication by constants (this is what makes it Presburger arithmetic),
- equality (`=` and `≠`) and ordering (`≤`), and
- the logical connectives `∧`, `∨`, `¬`, and `→`,

then invoking `omega` will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, `omega` will also fail.)

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

(Note the `Require Import Coq.omega.Omega.` at the top of the file.)

A Few More Handy Tactics

Finally, here are some miscellaneous tactics that you may find convenient.

- `clear H`: Delete hypothesis `H` from the context.
- `subst x`: Find an assumption `x = e` or `e = x` in the context, replace `x` with `e` throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away *all* assumptions of the form `x = e` or `e = x`.
- `rename... into...`: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named `x`, then `rename x into y` will change all occurrences of `x` to `y`.
- `assumption`: Try to find a hypothesis `H` in the context that exactly matches the goal; if one is found, behave like `apply H`.
- `contradiction`: Try to find a hypothesis `H` in the current context that is logically equivalent to `False`. If one is found, solve the goal.
- `constructor`: Try to find a constructor `c` (from some `Inductive` definition in the current environment) that can be applied to solve the current goal. If one is found, behave like `apply c`.

We'll see examples as we go along.

Evaluation as a Relation

We have presented `aeval` and `beval` as functions defined by `Fixpoints`. Another way to think about evaluation — one that we will see is often more flexible — is as a *relation* between expressions and their values. This leads naturally to `Inductive` definitions like the following one for arithmetic expressions...

```
Module aevalR_first_try.
```

think of it as inference rules!!!

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum : ∀ (n: nat),
      aevalR (ANum n) n
  | E_APlus : ∀ (e1 e2: aexp) (n1 n2: nat),
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus: ∀ (e1 e2: aexp) (n1 n2: nat),
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult : ∀ (e1 e2: aexp) (n1 n2: nat),
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMult e1 e2) (n1 * n2).
```

It will be convenient to have an infix notation for `aevalR`. We'll write `e \\ n` to mean that arithmetic expression `e` evaluates to value `n`.

```
Notation "e '\\' n"
         := (aevalR e n)
            (at level 50, left associativity)
         : type_scope.

End aevalR_first_try.
```

In fact, Coq provides a way to use this notation in the definition of `aevalR` itself. This reduces confusion by avoiding situations where we're working on a proof involving statements in the form `e \\ n` but we have to refer back to a definition written using the form `aevalR e n`.

We do this by first "reserving" the notation, then giving the definition together with a declaration of what the notation means.

explain @Eric

```
Reserved Notation "e '\\' n" (at level 50, left associativity).

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum : ∀ (n:nat),
      (ANum n) \\ n
  | E_APlus : ∀ (e1 e2: aexp) (n1 n2 : nat),
      (e1 \\ n1) → (e2 \\ n2) → (APlus e1 e2) \\ (n1 + n2)
  | E_AMinus : ∀ (e1 e2: aexp) (n1 n2 : nat),
      (e1 \\ n1) → (e2 \\ n2) → (AMinus e1 e2) \\ (n1 - n2)
  | E_AMult : ∀ (e1 e2: aexp) (n1 n2 : nat),
```

```
(e1 \\ n1) → (e2 \\ n2) → (AMult e1 e2) \\ (n1 * n2)
```

```
where "e '\\' n" := (aevalR e n) : type_scope.
```

Inference Rule Notation

In informal discussions, it is convenient to write the rules for `aevalR` and similar relations in the more readable graphical form of *inference rules*, where the premises above the line justify the conclusion below the line (we have already seen them in the IndProp chapter).

For example, the constructor `E_APlus`...

```
| E_APlus : ∀ (e1 e2: aexp) (n1 n2: nat),
    aevalR e1 n1 →
    aevalR e2 n2 →
    aevalR (APlus e1 e2) (n1 + n2)
```

...would be written like this as an inference rule:

$$\frac{\texttt{e1 \textbackslash\textbackslash n1} \qquad \texttt{e2 \textbackslash\textbackslash n2}}{\texttt{APlus e1 e2 \textbackslash\textbackslash n1+n2}}\ \text{(E\_APlus)}$$

Formally, there is nothing deep about inference rules: they are just implications. You can read the rule name on the right as the name of the constructor and read each of the linebreaks between the premises above the line (as well as the line itself) as →. All the variables mentioned in the rule (e_1, n_1, etc.) are implicitly bound by universal quantifiers at the beginning. (Such variables are often called *metavariables* to distinguish them from the variables of the language we are defining. At the moment, our arithmetic expressions don't include variables, but we'll soon be adding them.) The whole collection of rules is understood as being wrapped in an `Inductive` declaration. In informal prose, this is either elided or else indicated by saying something like "Let `aevalR` be the smallest relation closed under the following rules...".

For example, `\\` is the smallest relation closed under these rules:

$$\frac{}{\texttt{ANum n \textbackslash\textbackslash n}}\ \text{(E\_ANum)}$$

$$\frac{\texttt{e1 \textbackslash\textbackslash n1} \qquad \texttt{e2 \textbackslash\textbackslash n2}}{\texttt{APlus e1 e2 \textbackslash\textbackslash n1+n2}}\ \text{(E\_APlus)}$$

$$\frac{\texttt{e1 \textbackslash\textbackslash n1} \qquad \texttt{e2 \textbackslash\textbackslash n2}}{\texttt{AMinus e1 e2 \textbackslash\textbackslash n1-n2}}\ \text{(E\_AMinus)}$$

$$\frac{e_1 \;\backslash\backslash\; n_1 \qquad e_2 \;\backslash\backslash\; n_2}{\texttt{AMult}\ e_1\ e_2 \;\backslash\backslash\; n_1 * n_2} \quad \text{(E\_AMult)}$$

Equivalence of the Definitions

It is straightforward to prove that the relational and functional definitions of evaluation agree:

```
Theorem aeval_iff_aevalR : ∀ a n,
  (a \\ n) ↔ aeval a = n.
+
```

We can make the proof quite a bit shorter by making more use of tacticals.

```
Theorem aeval_iff_aevalR' : ∀ a n,
  (a \\ n) ↔ aeval a = n.
Proof.
  (* WORKED IN CLASS *)
  split.
  - (* -> *)
    intros H; induction H; subst; reflexivity.
  - (* <- *)
    generalize dependent n.
    induction a; simpl; intros; subst; constructor;
       try apply IHa1; try apply IHa2; reflexivity.
Qed.
```

anyone wrote it with assumption? answered in the .v file!

Exercise: 3 stars (bevalR)

Write a relation `bevalR` in the same style as `aevalR`, and prove that it is equivalent to `beval`.

```
Inductive bevalR: bexp → bool → Prop :=
(* FILL IN HERE *)
.

Lemma beval_iff_bevalR : ∀ b bv,
  bevalR b bv ↔ beval b = bv.
Proof.
  (* FILL IN HERE *) Admitted.
□

End AExp.
```

Computational vs. Relational Definitions

For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste: either way works.

However, there are circumstances where relational definitions of evaluation work much better than functional ones.

```
Module aevalR_division.
```

For example, suppose that we wanted to extend the arithmetic operations by considering also a division operation:

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp
  | ADiv : aexp → aexp → aexp. (* <--- new *)
```

Extending the definition of `aeval` to handle this new operation would not be straightforward (what should we return as the result of `ADiv (ANum 5) (ANum 0)`?). But extending `aevalR` is straightforward.

```
Reserved Notation "e '\\' n"
                  (at level 50, left associativity).

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum : ∀ (n:nat),
      (ANum n) \\ n
  | E_APlus : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (APlus a1 a2) \\ (n1 + n2)
  | E_AMinus : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (AMinus a1 a2) \\ (n1 - n2)
  | E_AMult : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (AMult a1 a2) \\ (n1 * n2)
  | E_ADiv : ∀ (a1 a2: aexp) (n1 n2 n3: nat),
      (a1 \\ n1) → (a2 \\ n2) → (n2 > 0) →
      (mult n2 n3 = n1) → (ADiv a1 a2) \\ n3

where "a '\\' n" := (aevalR a n) : type_scope.

End aevalR_division.

Module aevalR_extended.
```

Suppose, instead, that we want to extend the arithmetic operations by a nondeterministic number generator `any` that, when evaluated, may yield any number. (Note that this is not the same as making a *probabilistic* choice among all possible numbers — we're not specifying any particular distribution of results, but just saying what results are *possible*.)

```
Reserved Notation "e '\\' n" (at level 50, left associativity).

Inductive aexp : Type :=
  | AAny : aexp (* <--- NEW *)
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

Again, extending `aeval` would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers, but extending `aevalR` is no problem...

```
Inductive aevalR : aexp → nat → Prop :=
  | E_Any : ∀ (n:nat),
      AAny \\ n (* <--- new *)
  | E_ANum : ∀ (n:nat),
      (ANum n) \\ n
  | E_APlus : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (APlus a1 a2) \\ (n1 + n2)
  | E_AMinus : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (AMinus a1 a2) \\ (n1 - n2)
  | E_AMult : ∀ (a1 a2: aexp) (n1 n2 : nat),
      (a1 \\ n1) → (a2 \\ n2) → (AMult a1 a2) \\ (n1 * n2)

where "a '\\' n" := (aevalR a n) : type_scope.

End aevalR_extended.
```

At this point you maybe wondering: which style should I use by default? The examples above show that relational definitions are fundamentally more powerful than functional ones. For situations like these, where the thing being defined is not easy to express as a function, or indeed where it is *not* a function, there is no choice. But what about when both styles are workable?

One point in favor of relational definitions is that they can be more elegant and easier to understand.

Another is that Coq automatically generates nice inversion and induction principles from `Inductive` definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are by definition deterministic and defined on all arguments; for a relation we have to show these properties explicitly if we need them.
- With functions we can also take advantage of Coq's computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly "extracted" to executable code in OCaml or Haskell.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large Coq developments it is common to see a definition given in *both* functional and relational styles, plus a lemma stating that the two coincide, allowing further proofs to switch from one point of view to the other at will.

Expressions With Variables

Let's turn our attention back to defining Imp. The next thing we need to do is to enrich our arithmetic and boolean expressions with variables. To keep things simple, we'll assume that all variables are global and that they only hold numbers.

States

Since we'll want to look variables up to find out their current values, we'll reuse maps from the Maps chapter, with `string`s as the type of variables in Imp.

A *machine state* (or just *state*) represents the current values of *all* variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only going to mention a finite number of them. The state captures all of the information stored in memory. For Imp programs, because each variable stores a natural number, we can represent the state as a mapping from strings to `nat`, and will use `0` as default value in the store. For more complex programming languages, the state might have more structure.

```
Definition state := total_map nat.
```

Syntax

We can add variables to the arithmetic expressions we had before by simply adding one more constructor:

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : string → aexp (* <----- NEW *)
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

Defining a few variable names as notational shorthands will make examples easier to read:

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
```

(This convention for naming program variables (`X`, `Y`, `Z`) clashes a bit with our earlier use of uppercase letters for types. Since we're not using polymorphism heavily in the chapters developed to Imp, this overloading should not cause confusion.)

The definition of `bexp`s is unchanged (except that it now refers to the new `aexp`s):

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp.
```

Notations

To make Imp programs easier to read and write, we introduce some notations and implicit coercions.

You do not need to understand what these declarations do in detail to follow this chapter. Briefly, though, the `Coercion` declaration in Coq stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for `AId` allows us to use plain strings when an `aexp` is expected; the string will implicitly be wrapped with `AId`.

The notations below are declared in specific *notation scopes*, in order to avoid conflicts with other interpretations of the same symbols. Again, it is not necessary to understand the details.

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.
Definition bool_to_bexp (b: bool) : bexp :=
  if b then BTrue else BFalse.
Coercion bool_to_bexp : bool >-> bexp.

Bind Scope aexp_scope with aexp.
Infix "+" := APlus : aexp_scope.
Infix "-" := AMinus : aexp_scope.
Infix "*" := AMult : aexp_scope.
Bind Scope bexp_scope with bexp.
Infix "≤" := BLe : bexp_scope.
Infix "=" := BEq : bexp_scope.
Infix "&&" := BAnd : bexp_scope.
Notation "'!' b" := (BNot b) (at level 60) : bexp_scope.
```

We can now write `3 + (X * 2)` instead of `APlus 3 (AMult X 2)`, and `true && !(X ≤ 4)` instead of `BAnd true (BNot (BLe X 4))`.

Evaluation

The arith and boolean evaluators are extended to handle variables in the obvious way, taking a state as an extra argument:

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x (* <----- NEW *)
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2 ⇒ (aeval st a1) - (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval st a1) (aeval st a2)
```

```
  | BLe a1 a2 ⇒ leb (aeval st a1) (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.
```

We specialize our notation for total maps to the specific case of states, i.e. using `{ --> 0 }` as empty state.

```
Notation "{ a --> x }" :=
  (t_update { --> 0 } a x) (at level 0).
Notation "{ a --> x ; b --> y }" :=
  (t_update ({ a --> x }) b y) (at level 0).
Notation "{ a --> x ; b --> y ; c --> z }" :=
  (t_update ({ a --> x ; b --> y }) c z) (at level 0).
Notation "{ a --> x ; b --> y ; c --> z ; d --> t }" :=
    (t_update ({ a --> x ; b --> y ; c --> z }) d t) (at level 0).
Notation "{ a --> x ; b --> y ; c --> z ; d --> t ; e --> u }" :=
  (t_update ({ a --> x ; b --> y ; c --> z ; d --> t }) e u) (at
level 0).
Notation "{ a --> x ; b --> y ; c --> z ; d --> t ; e --> u ; f --> v
}" :=
  (t_update ({ a --> x ; b --> y ; c --> z ; d --> t ; e --> u }) f
v) (at level 0).

Example aexp1 :
  aeval { X --> 5 } (3 + (X * 2))
  = 13.
+

Example bexp1 :
  beval { X --> 5 } (true && !(X ≤ 4))
  = true.
+
```

Commands

Now we are ready define the syntax and behavior of Imp *commands* (sometimes called *statements*).

Syntax

Informally, commands `c` are described by the following BNF grammar. (We choose this slightly awkward concrete syntax for the sake of being able to define Imp syntax using Coq's Notation mechanism. In particular, we use `IFB` to avoid conflicting with the `if` notation from the standard library.)

```
c ::= SKIP | x ::= a | c ;; c | IFB b THEN c ELSE c FI
    | WHILE b DO c END
```

For example, here's factorial in Imp:

```
  Z ::= X;;
  Y ::= 1;;
  WHILE ! (Z = 0) DO
    Y ::= Y * Z;;
    Z ::= Z - 1
  END
```

When this command terminates, the variable `Y` will contain the factorial of the initial value of `X`.

Here is the formal definition of the abstract syntax of commands:

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

As for expressions, we can use a few `Notation` declarations to make reading and writing Imp programs more convenient.

```
Bind Scope com_scope with com.
Notation "'SKIP'" :=
   CSkip : com_scope.
Notation "x '::=' a" :=
  (CAss x a) (at level 60) : com_scope.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity) : com_scope.
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity) : com_scope.
Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity) : com_scope.
```

The following declaration is needed to be able to use the notations in match patterns.

```
Open Scope com_scope.
```

For example, here is the factorial function again, written as a formal definition to Coq:

```
Definition fact_in_coq : com :=
  Z ::= X;;
  Y ::= 1;;
  WHILE ! (Z = 0) DO
    Y ::= Y * Z;;
    Z ::= Z - 1
  END.
```

More Examples

Assignment:

```
Definition plus2 : com :=
  X ::= X + 2.
```

```
Definition XtimesYinZ : com :=
  Z ::= X * Y.

Definition subtract_slowly_body : com :=
  Z ::= Z - 1 ;;
  X ::= X - 1.
```

Loops

```
Definition subtract_slowly : com :=
  WHILE ! (X = 0) DO
    subtract_slowly_body
  END.

Definition subtract_3_from_5_slowly : com :=
  X ::= 3 ;;
  Z ::= 5 ;;
  subtract_slowly.
```

An infinite loop:

```
Definition loop : com :=
  WHILE true DO
    SKIP
  END.
```

Evaluating Commands

Next we need to define what it means to evaluate an Imp command. The fact that `WHILE` loops don't necessarily terminate makes defining an evaluation function tricky...

Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands, omitting the `WHILE` case.

```
Fixpoint ceval_fun_no_while (st : state) (c : com)
                          : state :=
  match c with
    | SKIP ⇒
        st
    | x ::= a1 ⇒
        st & { x --> (aeval st a1) }
    | c1 ;; c2 ⇒
        let st' := ceval_fun_no_while st c1 in
        ceval_fun_no_while st' c2
    | IFB b THEN c1 ELSE c2 FI ⇒
        if (beval st b)
          then ceval_fun_no_while st c1
```

```
            else ceval_fun_no_while st c2
      | WHILE b DO c END ⇒
          st (* bogus *)
  end.
```

In a traditional functional programming language like OCaml or Haskell we could add the `WHILE` case as follows:

```
Fixpoint ceval_fun (st : state) (c : com) : state :=
  match c with
    ...
    | WHILE b DO c END =>
        if (beval st b)
          then ceval_fun st (c; WHILE b DO c END)
          else st
  end.
```

Coq doesn't accept such a definition ("Error: Cannot guess decreasing argument of fix") because the function we want to define is not guaranteed to terminate. Indeed, it *doesn't* always terminate: for example, the full version of the `ceval_fun` function applied to the `loop` program above would never terminate. Since Coq is not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an (invalid!) program showing what would go wrong if Coq allowed non-terminating recursive functions:

```
Fixpoint loop_false (n : nat) : False := loop_false n.
```

That is, propositions like `False` would become provable (`loop_false 0` would be a proof of `False`), which would be a disaster for Coq's logical consistency.

Thus, because it doesn't terminate on all inputs, of `ceval_fun` cannot be written in Coq — at least not without additional tricks and workarounds (see chapter ImpCEvalFun if you're curious about what those might be).

Evaluation as a Relation

Here's a better way: define `ceval` as a *relation* rather than a *function* — i.e., define it in `Prop` instead of `Type`, as we did for `aevalR` above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a lot more flexibility in the definition. For example, if we add nondeterministic features like `any` to the language, we want the definition of evaluation to be nondeterministic — i.e., not only will it not be total, it will not even be a function!

We'll use the notation `c / st \\ st'` for the `ceval` relation: `c / st \\ st'` means that executing program `c` in a starting state `st` results in an ending state `st'`. This can be pronounced "`c` takes state `st` to `st'`".

Operational Semantics

Here is an informal definition of evaluation, presented as inference rules for readability:

$$\frac{}{\texttt{SKIP / st \\ st}}\ \text{(E\_Skip)}$$

$$\frac{\texttt{aeval st } a_1 \texttt{ = n}}{\texttt{x := } a_1 \texttt{ / st \\ st \& \{ x --> n \}}}\ \text{(E\_Ass)}$$

$$\frac{c_1 \texttt{ / st \\ st'} \qquad c_2 \texttt{ / st' \\ st''}}{c_1 \texttt{;;} c_2 \texttt{ / st \\ st''}}\ \text{(E\_Seq)}$$

$$\frac{\texttt{beval st } b_1 \texttt{ = true} \qquad c_1 \texttt{ / st \\ st'}}{\texttt{IF } b_1 \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ FI / st \\ st'}}\ \text{(E\_IfTrue)}$$

$$\frac{\texttt{beval st } b_1 \texttt{ = false} \qquad c_2 \texttt{ / st \\ st'}}{\texttt{IF } b_1 \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ FI / st \\ st'}}\ \text{(E\_IfFalse)}$$

$$\frac{\texttt{beval st b = false}}{\texttt{WHILE b DO c END / st \\ st}}\ \text{(E\_WhileFalse)}$$

$$\frac{\texttt{beval st b = true} \qquad \texttt{c / st \\ st'} \qquad \texttt{WHILE b DO c END / st' \\ st''}}{\texttt{WHILE b DO c END / st \\ st''}}\ \text{(E\_WhileTrue)}$$

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

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

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      SKIP / st \\ st
  | E_Ass : ∀ st a1 n x,
      aeval st a1 = n →
      (x ::= a1) / st \\ st & { x --> n }
  | E_Seq : ∀ c1 c2 st st' st'',
      c1 / st \\ st' →
      c2 / st' \\ st'' →
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      c1 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      c2 / st \\ st' →
```

```
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      c / st \\ st' →
      (WHILE b DO c END) / st' \\ st'' →
      (WHILE b DO c END) / st \\ st''

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

The cost of defining evaluation as a relation instead of a function is that we now need to construct *proofs* that some program evaluates to some result state, rather than just letting Coq's computation mechanism do it for us.

```
Example ceval_example1:
    (X ::= 2;;
     IFB X ≤ 1
       THEN Y ::= 3
       ELSE Z ::= 4
     FI)
   / { --> 0 } \\ { X --> 2 ; Z --> 4 }.
Proof.
  (* We must supply the intermediate state *)
  apply E_Seq with { X --> 2 }.
  - (* assignment command *)
    apply E_Ass. reflexivity.
  - (* if command *)
    apply E_IfFalse.
      reflexivity.
      apply E_Ass. reflexivity. Qed.
```

Exercise: 2 stars (ceval_example2)

```
Example ceval_example2:
  (X ::= 0;; Y ::= 1;; Z ::= 2) / { --> 0 } \\
  { X --> 0 ; Y --> 1 ; Z --> 2 }.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, optional (pup_to_n)

Write an Imp program that sums the numbers from 1 to X (inclusive: 1 + 2 + ... + X) in the variable Y. Prove that this program executes as intended for X = 2 (this is trickier than you might expect).

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

Theorem pup_to_2_ceval :
  pup_to_n / { X --> 2 }
     \\ { X --> 2 ; Y --> 0 ; Y --> 2 ; X --> 1 ; Y --> 3 ; X --> 0 }.
```

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

□

Determinism of Evaluation

Changing from a computational to a relational definition of evaluation is a good move because it frees us from the artificial requirement that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation really a partial function? Or is it possible that, beginning from the same state `st`, we could evaluate some command `c` in different ways to reach two different output states `st'` and `st''`?

may need to do this for your evaluation function!!

In fact, this cannot happen: `ceval` *is* a partial function:

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
```

+

Reasoning About Imp Programs

We'll get deeper into systematic techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can do quite a bit just working with the bare definitions. This section explores some examples.

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  plus2 / st \\ st' →
  st' X = (n + 2).
Proof.
  intros st n st' HX Heval.
```

Inverting `Heval` essentially forces Coq to expand one step of the `ceval` computation — in this case revealing that `st'` must be `st` extended with the new value of `X`, since `plus2` is an assignment

```
    inversion Heval. subst. clear Heval. simpl.
    apply t_update_eq. Qed.
```

Exercise: 3 stars, recommended (XtimesYinZ_spec)

State and prove a specification of `XtimesYinZ`.

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

□

Exercise: 3 stars, recommended (loop_never_stops)

```
Theorem loop_never_stops : ∀ st st',
  ~(loop / st \\ st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE true DO SKIP END) as loopdef
           eqn:Heqloopdef.
```

Proceed by induction on the assumed derivation showing that `loopdef` terminates. Most of the cases are immediately contradictory (and so can be solved in one step with `inversion`).

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

□

Exercise: 3 stars (no_whiles_eqv)

Consider the following function:

```
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | SKIP ⇒
      true
  | _ ::= _ ⇒
      true
  | c1 ;; c2 ⇒
      andb (no_whiles c1) (no_whiles c2)
  | IFB _ THEN ct ELSE cf FI ⇒
      andb (no_whiles ct) (no_whiles cf)
  | WHILE _ DO _ END ⇒
      false
  end.
```

This predicate yields `true` just on programs that have no while loops. Using `Inductive`, write a property `no_whilesR` such that `no_whilesR c` is provable exactly when `c` is a program with no while loops. Then prove its equivalence with `no_whiles`.

```
Inductive no_whilesR: com → Prop :=
 (* FILL IN HERE *)
.

Theorem no_whiles_eqv:
   ∀ c, no_whiles c = true ↔ no_whilesR c.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars (no_whiles_terminating)

Imp programs that don't involve while loops always terminate. State and prove a theorem `no_whiles_terminating` that says this. Use either `no_whiles` or `no_whilesR`, as you prefer.

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

□

Additional Exercises

Exercise: 3 stars (stack_compiler)

Old HP Calculators, programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a *stack*. For instance, the expression

```
(2*3)+(3*(4-2))
```

would be written as

```
2 3 * 3 4 2 - * +
```

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

```
[ ]            |    2 3 * 3 4 2 - * +
[2]            |    3 * 3 4 2 - * +
[3, 2]         |    * 3 4 2 - * +
[6]            |    3 4 2 - * +
[3, 6]         |    4 2 - * +
[4, 3, 6]      |    2 - * +
[2, 4, 3, 6]   |    - * +
[2, 3, 6]      |    * +
[6, 6]         |    +
[12]           |
```

The goal of this exercise is to write a small compiler that translates `aexp`s into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- `SPush n`: Push the number `n` on the stack.
- `SLoad x`: Load the identifier `x` from the store and push it on the stack
- `SPlus`: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- `SMinus`: Similar, but subtract.
- `SMult`: Similar, but multiply.

```
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : string → sinstr
| SPlus : sinstr
| SMinus : sinstr
| SMult : sinstr.
```

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that the specification leaves unspecified what to do when encountering an `SPlus`, `SMinus`, or `SMult` instruction if the stack contains less than two elements. In a sense, it is immaterial what we do, since our compiler will never emit such a malformed program.

```
Fixpoint s_execute (st : state) (stack : list nat)
                   (prog : list sinstr)
                 : list nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example s_execute1 :
     s_execute { --> 0 } []
       [SPush 5; SPush 3; SPush 1; SMinus]
   = [2; 5].
(* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: s_execute1 *)

Example s_execute2 :
     s_execute { X --> 3 } [3;4]
       [SPush 4; SLoad X; SMult; SPlus]
   = [15; 4].
(* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: s_execute2 *)
```

Next, write a function that compiles an `aexp` into a stack machine program. The effect of running the program should be the same as pushing the value of the expression on the stack.

```
Fixpoint s_compile (e : aexp) : list sinstr
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

After you've defined `s_compile`, prove the following to test that it works.

```
Example s_compile1 :
  s_compile (X - (2 * Y))
  = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, advanced (stack_compiler_correct)

Now we'll prove the correctness of the compiler implemented in the previous exercise. Remember that the specification left unspecified what to do when encountering an `SPlus`, `SMinus`, or `SMult` instruction if the stack contains less than two elements. (In order to make your correctness proof easier you might find it helpful to go back and change your implementation!)

Prove the following theorem. You will need to start by stating a more general lemma to get a usable induction hypothesis; the main theorem will then be a simple corollary of this lemma.

```
Theorem s_compile_correct : ∀ (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
```

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

Exercise: 3 stars, optional (short_circuit)

Most modern programming languages use a "short-circuit" evaluation rule for boolean `and`: to evaluate `BAnd` b_1 b_2, first evaluate b_1. If it evaluates to `false`, then the entire `BAnd` expression evaluates to `false` immediately, without evaluating b_2. Otherwise, b_2 is evaluated to determine the result of the `BAnd` expression.

Write an alternate version of `beval` that performs short-circuit evaluation of `BAnd` in this manner, and prove that it is equivalent to `beval`.

```
(* FILL IN HERE *)
```
□

```
Module BreakImp.
```

Exercise: 4 stars, advanced (break_imp)

Imperative languages like C and Java often include a `break` or similar statement for interrupting the execution of loops. In this exercise we consider how to add `break` to Imp. First, we need to enrich the language of commands with an additional case.

```
Inductive com : Type :=
  | CSkip : com
  | CBreak : com (* <-- new *)
  | CAss : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "'BREAK'" :=
  CBreak.
Notation "x '::=' a" :=
  (CAss 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' c₁ 'THEN' c₂ 'ELSE' c₃ 'FI'" :=
  (CIf c₁ c₂ c₃) (at level 80, right associativity).
```

Next, we need to define the behavior of `BREAK`. Informally, whenever `BREAK` is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. (If there aren't any enclosing loops, then the whole program simply terminates.) The final state should be the same as the one in which the `BREAK` statement was executed.

One important point is what to do when there are multiple loops enclosing a given BREAK. In those cases, BREAK should only terminate the *innermost* loop. Thus, after executing the following...

```
X ::= 0;;
Y ::= 1;;
WHILE 0 ≠ Y DO
  WHILE TRUE DO
    BREAK
  END;;
  X ::= 1;;
  Y ::= Y - 1
END
```

... the value of X should be 1, and not 0.

One way of expressing this behavior is to add another parameter to the evaluation relation that specifies whether evaluation of a command executes a BREAK statement:

```
Inductive result : Type :=
  | SContinue : result
  | SBreak : result.

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

Intuitively, `c / st \\ s / st'` means that, if `c` is started in state `st`, then it terminates in state `st'` and either signals that the innermost surrounding loop (or the whole program) should exit immediately (`s = SBreak`) or that execution should continue normally (`s = SContinue`).

The definition of the "`c / st \\ s / st'`" relation is very similar to the one we gave above for the regular evaluation relation (`c / st \\ st'`) — we just need to handle the termination signals appropriately:

- If the command is SKIP, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is BREAK, the state stays unchanged but we signal a SBreak.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form IFB b THEN c_1 ELSE c_2 FI, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.
- If the command is a sequence c_1 ;; c_2, we first execute c_1. If this yields a SBreak, we skip the execution of c_2 and propagate the SBreak signal to the surrounding context; the resulting state is the same as the one obtained by

executing c_1 alone. Otherwise, we execute c_2 on the state obtained after executing c_1, and propagate the signal generated there.

- Finally, for a loop of the form `WHILE b DO c END`, the semantics is almost the same as before. The only difference is that, when `b` evaluates to true, we execute `c` and check the signal that it raises. If that signal is `SContinue`, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since `BREAK` only terminates the innermost loop, `WHILE` signals `SContinue`.

Based on the above description, complete the definition of the `ceval` relation.

```
Inductive ceval : com → state → result → state → Prop :=
  | E_Skip : ∀ st,
      CSkip / st \\ SContinue / st
  (* FILL IN HERE *)

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

Now prove the following properties of your definition of `ceval`:

```
Theorem break_ignore : ∀ c st st' s,
     (BREAK;; c) / st \\ s / st' →
     st = st'.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem while_continue : ∀ b c st st' s,
  (WHILE b DO c END) / st \\ s / st' →
  s = SContinue.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem while_stops_on_break : ∀ b c st st',
  beval st b = true →
  c / st \\ SBreak / st' →
  (WHILE b DO c END) / st \\ SContinue / st'.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, advanced, optional (while_break_true)

```
Theorem while_break_true : ∀ b c st st',
  (WHILE b DO c END) / st \\ SContinue / st' →
  beval st' b = true →
  ∃ st'', c / st'' \\ SBreak / st'.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, advanced, optional (ceval_deterministic)

```
Theorem ceval_deterministic: ∀ (c:com) st st1 st2 s1 s2,
     c / st \\ s1 / st1 →
     c / st \\ s2 / st2 →
     st1 = st2 ∧ s1 = s2.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

```
End BreakImp.
```

Exercise: 4 stars, optional (add_for_loop)

Add C-style `for` loops to the language of commands, update the `ceval` definition to define the semantics of `for` loops, and add cases for `for` loops as needed so that all the proofs in this file are accepted by Coq.

A `for` loop should be parameterized by (a) a statement executed initially, (b) a test that is run on each iteration of the loop to determine whether the loop should continue, (c) a statement executed at the end of each loop iteration, and (d) a statement that makes up the body of the loop. (You don't need to worry about making up a concrete Notation for `for` loops, but feel free to play with this too if you like.)

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

□