SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

SMALLSTEP

SMALL-STEP OPERATIONAL SEMANTICS

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

The evaluators we have seen so far (for `aexps`, `bexps`, commands, ...) have been formulated in a "big-step" style: they specify how a given expression can be evaluated to its final value (or a command plus a store to a final store) "all in one big step."

This style is simple and natural for many purposes — indeed, Gilles Kahn, who popularized it, called it *natural semantics*. But there are some things it does not do well. In particular, it does not give us a natural way of talking about *concurrent* programming languages, where the semantics of a program — i.e., the essence of how it behaves — is not just which input states get mapped to which output states, but also includes the intermediate states that it passes through along the way, since these states can also be observed by concurrently executing code.

Another shortcoming of the big-step style is more technical, but critical in many situations. Suppose we want to define a variant of Imp where variables could hold *either* numbers *or* lists of numbers. In the syntax of this extended language, it will be possible to write strange expressions like `2 + nil`, and our semantics for arithmetic expressions will then need to say something about how such expressions behave. One possibility is to maintain the convention that every arithmetic expressions evaluates to some number by choosing some way of viewing a list as a number — e.g., by specifying that a list should be interpreted as `0` when it occurs in a context expecting a number. But this is really a bit of a hack.

A much more natural approach is simply to say that the behavior of an expression like `2+nil` is *undefined* — i.e., it doesn't evaluate to any result at all. And we can easily do this: we just have to formulate `aeval` and `beval` as `Inductive` propositions rather than Fixpoints, so that we can make them partial functions instead of total ones.

Now, however, we encounter a serious deficiency. In this language, a command might fail to map a given starting state to any ending state for *two quite different reasons*: either because the execution gets into an infinite loop or because, at some point, the program tries to do an operation that makes no sense, such as adding a number to a list, so that none of the evaluation rules can be applied.

These two outcomes — nontermination vs. getting stuck in an erroneous configuration — should not be confused. In particular, we want to *allow* the first (permitting the possibility of infinite loops is the price we pay for the convenience of programming with general looping constructs like `while`) but *prevent* the second (which is just wrong), for example by adding some form of *typechecking* to the language. Indeed, this will be a major topic for the rest of the course. As a first step, we need a way of presenting the semantics that allows us to distinguish nontermination from erroneous "stuck states."

So, for lots of reasons, we'd often like to have a finer-grained way of defining and reasoning about program behaviors. This is the topic of the present chapter. Our goal is to replace the "big-step" `eval` relation with a "small-step" relation that specifies, for a given program, how the "atomic steps" of computation are performed.

A Toy Language

To save space in the discussion, let's go back to an incredibly simple language containing just constants and addition. (We use single letters — `C` and `P` (for Constant and Plus) — as constructor names, for brevity.) At the end of the chapter, we'll see how to apply the same techniques to the full Imp language.

```
Inductive tm : Type :=
  | C : nat → tm (* Constant *)
  | P : tm → tm → tm. (* Plus *)
```

Here is a standard evaluator for this language, written in the big-step style that we've been using up to this point.

```
Fixpoint evalF (t : tm) : nat :=
  match t with
  | C n ⇒ n
  | P a1 a2 ⇒ evalF a1 + evalF a2
  end.
```

Here is the same evaluator, written in exactly the same style, but formulated as an inductively defined relation. Again, we use the notation `t \\ n` for "`t` evaluates to `n`."

$$\frac{}{\texttt{C n \\\\ n}} \quad \text{(E\_Const)}$$

$$\frac{\begin{array}{c} t_1 \ \backslash\backslash\ n_1 \\ t_2 \ \backslash\backslash\ n_2 \end{array}}{P\ t_1\ t_2 \ \backslash\backslash\ n_1 + n_2} \quad \text{(E\_Plus)}$$

```
Reserved Notation " t '\\' n " (at level 50, left
associativity).

Inductive eval : tm → nat → Prop :=
 +

Module SimpleArith1.
```

Now, here is the corresponding *small-step* evaluation relation.

$$\frac{}{P\ (C\ n_1)\ (C\ n_2) \Longrightarrow C\ (n_1 + n_2)} \quad \text{(ST\_PlusConstConst)}$$

$$\frac{t_1 \Longrightarrow t_1'}{P\ t_1\ t_2 \Longrightarrow P\ t_1'\ t_2} \quad \text{(ST\_Plus1)}$$

$$\frac{t_2 \Longrightarrow t_2'}{P\ (C\ n_1)\ t_2 \Longrightarrow P\ (C\ n_1)\ t_2'} \quad \text{(ST\_Plus2)}$$

```
Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ n1 t2 t2',
      t2 ==> t2' →
      P (C n1) t2 ==> P (C n1) t2'

  where " t '==>' t' " := (step t t').
```

Things to notice:

- We are defining just a single reduction step, in which one `P` node is replaced by its value.
- Each step finds the *leftmost* `P` node that is ready to go (both of its operands are constants) and rewrites it in place. The first rule tells how to rewrite this `P` node itself; the other two rules tell how to find it.
- A term that is just a constant cannot take a step.

Let's pause and check a couple of examples of reasoning with the `step` relation...

If t_1 can take a step to t_1', then $P\ t_1\ t_2$ steps to $P\ t_1'\ t_2$:

```
Example test_step_1 :
      P
        (P (C 0) (C 3))
        (P (C 2) (C 4))
      ==>
      P
        (C (0 + 3))
        (P (C 2) (C 4)).
```

+

Exercise: 1 star (test_step_2)

Right-hand sides of sums can take a step only when the left-hand side is finished: if t_2 can take a step to t_2', then `P (C n) t2` steps to `P (C n) t2'`:

```
Example test_step_2 :
      P
        (C 0)
        (P
          (C 2)
          (P (C 0) (C 3)))
      ==>
      P
        (C 0)
        (P
          (C 2)
          (C (0 + 3))).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

```
End SimpleArith1.
```

Relations

We will be working with several different single-step relations, so it is helpful to generalize a bit and state a few definitions and theorems about relations in general. (The optional chapter `Rel.v` develops some of these ideas in a bit more detail; it may be useful if the treatment here is too dense.)

A *binary relation* on a set `X` is a family of propositions parameterized by two elements of `X` — i.e., a proposition about pairs of elements of `X`.

```
Definition relation (X: Type) := X → X → Prop.
```

Our main examples of such relations in this chapter will be the single-step reduction relation, `==>`, and its multi-step variant, `==>*` (defined below), but there are many other examples — e.g., the "equals," "less than," "less than or equal to," and "is the square of" relations on numbers, and the "prefix of" relation on lists and strings.

One simple property of the `==>` relation is that, like the big-step evaluation relation for Imp, it is *deterministic*.

Theorem: For each `t`, there is at most one `t'` such that `t` steps to `t'` (`t ==> t'` is provable). This is the same as saying that `==>` is deterministic.

Proof sketch: We show that if `x` steps to both y_1 and y_2, then y_1 and y_2 are equal, by induction on a derivation of `step` `x` y_1. There are several cases to consider, depending on the last rule used in this derivation and the last rule in the given derivation of `step` `x` y_2.

- If both are `ST_PlusConstConst`, the result is immediate.
- The cases when both derivations end with `ST_Plus1` or `ST_Plus2` follow by the induction hypothesis.
- It cannot happen that one is `ST_PlusConstConst` and the other is `ST_Plus1` or `ST_Plus2`, since this would imply that `x` has the form `P` t_1 t_2 where both t_1 and t_2 are constants (by `ST_PlusConstConst`) *and* one of t_1 or t_2 has the form `P _`.
- Similarly, it cannot happen that one is `ST_Plus1` and the other is `ST_Plus2`, since this would imply that `x` has the form `P` t_1 t_2 where t_1 has both the form `P` t_{11} t_{12} and the form `C n`. □

Formally:

```
Definition deterministic {X: Type} (R: relation X) :=
  ∀ x y₁ y₂ : X, R x y₁ → R x y₂ → y₁ = y₂.

Module SimpleArith2.
Import SimpleArith1.

Theorem step_deterministic:
  deterministic step.
 +

End SimpleArith2.
```

There is some annoying repetition in this proof. Each use of `inversion` Hy_2 results in three subcases, only one of which is relevant (the one that matches the current case in the induction on Hy_1). The other two subcases need to be dismissed by finding the contradiction among the hypotheses and doing inversion on it.

The following custom tactic, called `solve_by_inverts`, can be helpful in such cases. It will solve the goal if it can be solved by inverting some hypothesis; otherwise, it fails.

```
Ltac solve_by_inverts n :=
  match goal with | H : ?T |- _ ⇒
  match type of T with Prop ⇒
    solve [
      inversion H;
      match n with S (S (?n')) ⇒ subst; solve_by_inverts (S n')
end ]
  end end.
```

The details of how this works are not important for now, but it illustrates the power of Coq's `Ltac` language for programmatically defining special-purpose tactics. It looks through the current proof state for a hypothesis `H` (the first `match`) of type `Prop` (the second `match`) such that performing inversion on `H` (followed by a recursive invocation of the same tactic, if its argument `n` is greater than one) completely solves the current goal. If no such hypothesis exists, it fails.

We will usually want to call `solve_by_inverts` with argument 1 (especially as larger arguments can lead to very slow proof checking), so we define `solve_by_invert` as a shorthand for this case.

```
Ltac solve_by_invert :=
  solve_by_inverts 1.
```

Let's see how a proof of the previous theorem can be simplified using this tactic...

```
Module SimpleArith3.
Import SimpleArith1.

Theorem step_deterministic_alt: deterministic step.
Proof.
  intros x y₁ y₂ Hy₁ Hy₂.
  generalize dependent y₂.
  induction Hy₁; intros y₂ Hy₂;
    inversion Hy₂; subst; try solve_by_invert.
  - (* ST_PlusConstConst *) reflexivity.
  - (* ST_Plus1 *)
    apply IHHy1 in H₂. rewrite H₂. reflexivity.
  - (* ST_Plus2 *)
    apply IHHy1 in H₂. rewrite H₂. reflexivity.
Qed.

End SimpleArith3.
```

Values

Next, it will be useful to slightly reformulate the definition of single-step reduction by stating it in terms of "values."

It is useful to think of the ==> relation as defining an *abstract machine*:

- At any moment, the *state* of the machine is a term.
- A *step* of the machine is an atomic unit of computation — here, a single "add" operation.
- The *halting states* of the machine are ones where there is no more computation to be done.

We can then execute a term `t` as follows:

- Take `t` as the starting state of the machine.

- Repeatedly use the ==> relation to find a sequence of machine states, starting with `t`, where each state steps to the next.
- When no more reduction is possible, "read out" the final state of the machine as the result of execution.

Intuitively, it is clear that the final states of the machine are always terms of the form `C n` for some `n`. We call such terms *values*.

```
Inductive value : tm → Prop :=
  | v_const : ∀ n, value (C n).
```

Having introduced the idea of values, we can use it in the definition of the ==> relation to write `ST_Plus2` rule in a slightly more elegant way:

$$\frac{}{\texttt{P (C } n_1\texttt{) (C } n_2\texttt{) ==> C (}n_1 + n_2\texttt{)}} \quad \text{(ST\_PlusConstConst)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{P } t_1\ t_2 \texttt{ ==> P } t_1'\ t_2} \quad \text{(ST\_Plus1)}$$

$$\frac{\begin{array}{c}\texttt{value } v_1 \\ t_2 \texttt{ ==> } t_2'\end{array}}{\texttt{P } v_1\ t_2 \texttt{ ==> P } v_1\ t_2'} \quad \text{(ST\_Plus2)}$$

Again, the variable names here carry important information: by convention, v_1 ranges only over values, while t_1 and t_2 range over arbitrary terms. (Given this convention, the explicit `value` hypothesis is arguably redundant. We'll keep it for now, to maintain a close correspondence between the informal and Coq versions of the rules, but later on we'll drop it in informal rules for brevity.)

Here are the formal rules:

```
Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
          P (C n1) (C n2)
      ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
        t1 ==> t1' →
        P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ v1 t2 t2',
        value v1 → (* <----- n.b. *)
        t2 ==> t2' →
        P v1 t2 ==> P v1 t2'

  where " t '==>' t' " := (step t t').
```

Exercise: 3 stars, recommended (redo_determinism)

As a sanity check on this change, let's re-verify determinism.

Proof sketch: We must show that if x steps to both y_1 and y_2, then y_1 and y_2 are equal. Consider the final rules used in the derivations of step x y_1 and step x y_2.

- If both are ST_PlusConstConst, the result is immediate.
- It cannot happen that one is ST_PlusConstConst and the other is ST_Plus1 or ST_Plus2, since this would imply that x has the form P t_1 t_2 where both t_1 and t_2 are constants (by ST_PlusConstConst) *and* one of t_1 or t_2 has the form P _.
- Similarly, it cannot happen that one is ST_Plus1 and the other is ST_Plus2, since this would imply that x has the form P t_1 t_2 where t_1 both has the form P t_{11} t_{12} and is a value (hence has the form C n).
- The cases when both derivations end with ST_Plus1 or ST_Plus2 follow by the induction hypothesis. □

Most of this proof is the same as the one above. But to get maximum benefit from the exercise you should try to write your formal version from scratch and just use the earlier one if you get stuck.

```
Theorem step_deterministic :
  deterministic step.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Strong Progress and Normal Forms

The definition of single-step reduction for our toy language is fairly simple, but for a larger language it would be easy to forget one of the rules and accidentally create a situation where some term cannot take a step even though it has not been completely reduced to a value. The following theorem shows that we did not, in fact, make such a mistake here.

Theorem (*Strong Progress*): If t is a term, then either t is a value or else there exists a term t' such that t ==> t'.

Proof: By induction on t.

- Suppose t = C n. Then t is a value.
- Suppose t = P t_1 t_2, where (by the IH) t_1 either is a value or can step to some t_1', and where t_2 is either a value or can step to some t_2'. We must show P t_1 t_2 is either a value or steps to some t'.

- If `t1` and `t2` are both values, then `t` can take a step, by `ST_PlusConstConst`.
- If `t1` is a value and `t2` can take a step, then so can `t`, by `ST_Plus2`.
- If `t1` can take a step, then so can `t`, by `ST_Plus1`. □

Or, formally:

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> t').
+
```

This important property is called *strong progress*, because every term either is a value or can "make progress" by stepping to some other term. (The qualifier "strong" distinguishes it from a more refined version that we'll see in later chapters, called just *progress*.)

The idea of "making progress" can be extended to tell us something interesting about values: in this language, values are exactly the terms that *cannot* make progress in this sense.

To state this observation formally, let's begin by giving a name to terms that cannot make progress. We'll call them *normal forms*.

```
Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ¬ ∃ t', R t t'.
```

Note that this definition specifies what it is to be a normal form for an *arbitrary* relation `R` over an arbitrary set `X`, not just for the particular single-step reduction relation over terms that we are interested in at the moment. We'll re-use the same terminology for talking about other relations later in the course.

We can use this terminology to generalize the observation we made in the strong progress theorem: in this language, normal forms and values are actually the same thing.

```
Lemma value_is_nf : ∀ v,
  value v → normal_form step v.
+
```

```
Lemma nf_is_value : ∀ t,
  normal_form step t → value t.
+
```

```
Corollary nf_same_as_value : ∀ t,
  normal_form step t ↔ value t.
+
```

Why is this interesting?

Because `value` is a syntactic concept — it is defined by looking at the form of a term — while `normal_form` is a semantic one — it is defined by looking at how the term steps. It is not obvious that these concepts should coincide!

Indeed, we could easily have written the definitions so that they would *not* coincide.

Exercise: 3 stars, optional (value_not_same_as_normal_form1)

We might, for example, mistakenly define `value` so that it includes some terms that are not finished reducing. (Even if you don't work this exercise and the following ones in Coq, make sure you can think of an example of such a term.)

```
Module Temp1.

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n)
| v_funny : ∀ t1 n2, (* <---- *)
              value (P t1 (C n2)).

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ v1 t2 t2',
      value v1 →
      t2 ==> t2' →
      P v1 t2 ==> P v1 t2'

  where " t '==>' t' " := (step t t').

Lemma value_not_same_as_normal_form :
  ∃ v, value v ∧ ¬ normal_form step v.
Proof.
  (* FILL IN HERE *) Admitted.
End Temp1.
```
☐

Exercise: 2 stars, optional (value_not_same_as_normal_form2)

Alternatively, we might mistakenly define `step` so that it permits something designated as a value to reduce further.

```
Module Temp2.

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n).

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_Funny : ∀ n, (* <---- *)
      C n ==> P (C n) (C 0)
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n₁) (C n₂) ==> C (n₁ + n₂)
  | ST_Plus1 : ∀ t₁ t₁' t₂,
      t₁ ==> t₁' →
      P t₁ t₂ ==> P t₁' t₂
  | ST_Plus2 : ∀ v₁ t₂ t₂',
      value v₁ →
      t₂ ==> t₂' →
      P v₁ t₂ ==> P v₁ t₂'

  where " t '==>' t' " := (step t t').

Lemma value_not_same_as_normal_form :
  ∃ v, value v ∧ ¬ normal_form step v.
 +
End Temp2.
```
□

Exercise: 3 stars, optional (value_not_same_as_normal_form3)

Finally, we might define `value` and `step` so that there is some term that is not a value but that cannot take a step in the `step` relation. Such terms are said to be *stuck*. In this case this is caused by a mistake in the semantics, but we will also see situations where, even in a correct language definition, it makes sense to allow some terms to be stuck.

```
  Module Temp3.

  Inductive value : tm → Prop :=
    | v_const : ∀ n, value (C n).

  Reserved Notation " t '==>' t' " (at level 40).

  Inductive step : tm → tm → Prop :=
    | ST_PlusConstConst : ∀ n₁ n₂,
        P (C n₁) (C n₂) ==> C (n₁ + n₂)
    | ST_Plus1 : ∀ t₁ t₁' t₂,
        t₁ ==> t₁' →
        P t₁ t₂ ==> P t₁' t₂

    where " t '==>' t' " := (step t t').
```

(Note that `ST_Plus2` is missing.)

```
  Lemma value_not_same_as_normal_form :
    ∃ t, ¬ value t ∧ normal_form step t.
  Proof.
    (* FILL IN HERE *) Admitted.

  End Temp3.
```
□

Additional Exercises

```
  Module Temp4.
```

Here is another very simple language whose terms, instead of being just addition expressions and numbers, are just the booleans true and false and a conditional expression...

```
Inductive tm : Type :=
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.

Inductive value : tm → Prop :=
  | v_true : value ttrue
  | v_false : value tfalse.

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      tif t1 t2 t3 ==> tif t1' t2 t3

  where " t '==>' t' " := (step t t').
```

Exercise: 1 star (smallstep_bools)

Which of the following propositions are provable? (This is just a thought exercise, but for an extra challenge feel free to prove your answers in Coq.)

```
Definition bool_step_prop1 :=
  tfalse ==> tfalse.

(* FILL IN HERE *)

Definition bool_step_prop2 :=
     tif
       ttrue
       (tif ttrue ttrue ttrue)
       (tif tfalse tfalse tfalse)
  ==>
     ttrue.

(* FILL IN HERE *)

Definition bool_step_prop3 :=
     tif
       (tif ttrue ttrue ttrue)
       (tif ttrue ttrue ttrue)
       tfalse
   ==>
     tif
       ttrue
```

```
          (tif ttrue ttrue ttrue)
          tfalse.

  (* FILL IN HERE *)
```
□

Exercise: 3 stars, optional (progress_bool)

Just as we proved a progress theorem for plus expressions, we can do so for boolean expressions, as well.

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> t').
Proof.
  (* FILL IN HERE *) Admitted.
```
□

Exercise: 2 stars, optional (step_deterministic)

```
Theorem step_deterministic :
  deterministic step.
Proof.
  (* FILL IN HERE *) Admitted.
```
□

```
Module Temp5.
```

Exercise: 2 stars (smallstep_bool_shortcut)

Suppose we want to add a "short circuit" to the step relation for boolean expressions, so that it can recognize when the `then` and `else` branches of a conditional are the same value (either `ttrue` or `tfalse`) and reduce the whole conditional to this value in a single step, even if the guard has not yet been reduced to a value. For example, we would like this proposition to be provable:

```
         tif
            (tif ttrue ttrue ttrue)
            tfalse
            tfalse
     ==>
         tfalse.
```

Write an extra clause for the step relation that achieves this effect and prove `bool_step_prop4`.

```
Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t₁ ==> t₁' →
      tif t₁ t₂ t₃ ==> tif t₁' t₂ t₃
  (* FILL IN HERE *)

  where " t '==>' t' " := (step t t').

Definition bool_step_prop4 :=
         tif
            (tif ttrue ttrue ttrue)
            tfalse
            tfalse
     ==>
         tfalse.

Example bool_step_prop4_holds :
  bool_step_prop4.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, optional (properties_of_altered_step)

It can be shown that the determinism and strong progress theorems for the step relation in the lecture notes also hold for the definition of step given above. After we add the clause `ST_ShortCircuit`...

- Is the `step` relation still deterministic? Write yes or no and briefly (1 sentence) explain your answer.

  Optional: prove your answer correct in Coq.

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

- Does a strong progress theorem hold? Write yes or no and briefly (1 sentence) explain your answer.

  Optional: prove your answer correct in Coq.

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

- In general, is there any way we could cause strong progress to fail if we took away one or more constructors from the original step relation? Write yes or no and briefly (1 sentence) explain your answer.

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

☐

```
    End Temp5.
    End Temp4.
```

Multi-Step Reduction

We've been working so far with the *single-step reduction* relation ==>, which formalizes the individual steps of an abstract machine for executing programs.

We can use the same machine to reduce programs to completion — to find out what final result they yield. This can be formalized as follows:

- First, we define a *multi-step reduction relation* ==>*, which relates terms t and t' if t can reach t' by any number (including zero) of single reduction steps.
- Then we define a "result" of a term t as a normal form that t can reach by multi-step reduction.

Since we'll want to reuse the idea of multi-step reduction many times, let's take a little extra trouble and define it generically.

Given a relation R, we define a relation multi R, called the *multi-step closure of R* as follows.

```
Inductive multi {X:Type} (R: relation X) : relation X :=
  | multi_refl : ∀ (x : X), multi R x x
  | multi_step : ∀ (x y z : X),
                    R x y →
                    multi R y z →
                    multi R x z.
```

(In the Rel chapter of *Logical Foundations* and the Coq standard library, this relation is called clos_refl_trans_1n. We give it a shorter name here for the sake of readability.)

The effect of this definition is that multi R relates two elements x and y if

- x = y, or
- R x y, or
- there is some nonempty sequence z_1, z_2, ..., zn such that

  R x z_1

  R z_1 z_2

  ...

  R zn y.

Thus, if R describes a single-step of computation, then z_1...zn is the sequence of intermediate steps of computation between x and y.

We write ==>* for the multi step relation on terms.

```
Notation " t '==>*' t' " := (multi step t t') (at level 40).
```

The relation multi R has several crucial properties.

First, it is obviously *reflexive* (that is, ∀x, multi R x x). In the case of the ==>* (i.e., multi step) relation, the intuition is that a term can execute to itself by taking zero steps of execution.

Second, it contains `R` — that is, single-step executions are a particular case of multi-step executions. (It is this fact that justifies the word "closure" in the term "multi-step closure of `R`.")

```
Theorem multi_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → (multi R) x y.
```
+

Third, `multi R` is *transitive*.

```
Theorem multi_trans :
  ∀ (X:Type) (R: relation X) (x y z : X),
      multi R x y →
      multi R y z →
      multi R x z.
```
+

In particular, for the `multi step` relation on terms, if t_1==>*t_2 and t_2==>*t_3, then t_1==>*t_3.

Examples

Here's a specific instance of the `multi step` relation:

```
Lemma test_multistep_1:
      P
        (P (C 0) (C 3))
        (P (C 2) (C 4))
   ==>*
      C ((0 + 3) + (2 + 4)).
```
+

Here's an alternate proof of the same fact that uses `eapply` to avoid explicitly constructing all the intermediate terms.

```
Lemma test_multistep_1':
      P
        (P (C 0) (C 3))
        (P (C 2) (C 4))
  ==>*
      C ((0 + 3) + (2 + 4)).
Proof.
  eapply multi_step. apply ST_Plus1. apply ST_PlusConstConst.
  eapply multi_step. apply ST_Plus2. apply v_const.
  apply ST_PlusConstConst.
  eapply multi_step. apply ST_PlusConstConst.
  apply multi_refl. Qed.
```

Exercise: 1 star, optional (test_multistep_2)

```
Lemma test_multistep_2:
  C 3 ==>* C 3.
Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Exercise: 1 star, optional (test_multistep_3)

```
Lemma test_multistep_3:
      P (C 0) (C 3)
   ==>*
      P (C 0) (C 3).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (test_multistep_4)

```
Lemma test_multistep_4:
      P
        (C 0)
        (P
          (C 2)
          (P (C 0) (C 3)))
  ==>*
      P
        (C 0)
        (C (2 + (0 + 3))).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Normal Forms Again

If `t` reduces to `t'` in zero or more steps and `t'` is a normal form, we say that "`t'` is a normal form of `t`."

```
Definition step_normal_form := normal_form step.

Definition normal_form_of (t t' : tm) :=
  (t ==>* t' ∧ step_normal_form t').
```

We have already seen that, for our language, single-step reduction is deterministic — i.e., a given term can take a single step in at most one way. It follows from this that, if `t` can reach a normal form, then this normal form is unique. In other words, we can actually pronounce `normal_form t t'` as "`t'` is *the* normal form of `t`."

Exercise: 3 stars, optional (normal_forms_unique)

```
Theorem normal_forms_unique:
  deterministic normal_form_of.
Proof.
  (* We recommend using this initial setup as-is! *)
  unfold deterministic. unfold normal_form_of.
  intros x y1 y2 P1 P2.
  inversion P1 as [P11 P12]; clear P1.
  inversion P2 as [P21 P22]; clear P2.
  generalize dependent y2.
  (* FILL IN HERE *) Admitted.
```

☐

Indeed, something stronger is true for this language (though not for all languages): the reduction of *any* term `t` will eventually reach a normal form — i.e., `normal_form_of` is a *total* function. Formally, we say the `step` relation is *normalizing*.

```
Definition normalizing {X:Type} (R:relation X) :=
  ∀ t, ∃ t',
    (multi R) t t' ∧ normal_form R t'.
```

To prove that `step` is normalizing, we need a couple of lemmas.

First, we observe that, if `t` reduces to `t'` in many steps, then the same sequence of reduction steps within `t` is also possible when `t` appears as the left-hand child of a `P` node, and similarly when `t` appears as the right-hand child of a `P` node whose left-hand child is a value.

```
Lemma multistep_congr_1 : ∀ t₁ t₁' t₂,
     t₁ ==>* t₁' →
     P t₁ t₂ ==>* P t₁' t₂.
 +
```

Exercise: 2 stars (multistep_congr_2)

```
Lemma multistep_congr_2 : ∀ t₁ t₂ t₂',
     value t₁ →
     t₂ ==>* t₂' →
     P t₁ t₂ ==>* P t₁ t₂'.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

With these lemmas in hand, the main proof is a straightforward induction.

Theorem: The `step` function is normalizing — i.e., for every `t` there exists some `t'` such that `t` steps to `t'` and `t'` is a normal form.

Proof sketch: By induction on terms. There are two cases to consider:

- `t = C n` for some `n`. Here `t` doesn't take a step, and we have `t' = t`. We can derive the left-hand side by reflexivity and the right-hand side by observing (a) that values are normal forms (by `nf_same_as_value`) and (b) that `t` is a value (by `v_const`).

- `t` = `P` t_1 t_2 for some t_1 and t_2. By the IH, t_1 and t_2 have normal forms t_1' and t_2'. Recall that normal forms are values (by `nf_same_as_value`); we know that t_1' = `C` n_1 and t_2' = `C` n_2, for some n_1 and n_2. We can combine the `==>*` derivations for t_1 and t_2 using `multi_congr_1` and `multi_congr_2` to prove that `P` t_1 t_2 reduces in many steps to `C` (n_1 + n_2).

  It is clear that our choice of `t'` = `C` (n_1 + n_2) is a value, which is in turn a normal form. □

```
Theorem step_normalizing :
  normalizing step.
```
+

Equivalence of Big-Step and Small-Step

Having defined the operational semantics of our tiny programming language in two different ways (big-step and small-step), it makes sense to ask whether these definitions actually define the same thing! They do, though it takes a little work to show it. The details are left as an exercise.

Exercise: 3 stars (eval__multistep)

```
Theorem eval__multistep : ∀ t n,
  t \\ n → t ==>* C n.
```

The key ideas in the proof can be seen in the following picture:

```
P t1 t2 ==>               (by ST_Plus1)
P t1' t2 ==>              (by ST_Plus1)
P t1'' t2 ==>             (by ST_Plus1)
...
P (C n1) t2 ==>           (by ST_Plus2)
P (C n1) t2' ==>          (by ST_Plus2)
P (C n1) t2'' ==>         (by ST_Plus2)
...
P (C n1) (C n2) ==>       (by ST_PlusConstConst)
C (n1 + n2)
```

That is, the multistep reduction of a term of the form `P` t_1 t_2 proceeds in three phases:

- First, we use `ST_Plus1` some number of times to reduce t_1 to a normal form, which must (by `nf_same_as_value`) be a term of the form `C` n_1 for some n_1.
- Next, we use `ST_Plus2` some number of times to reduce t_2 to a normal form, which must again be a term of the form `C` n_2 for some n_2.
- Finally, we use `ST_PlusConstConst` one time to reduce `P (C` n_1`) (C` n_2`)` to `C (`$n_1 + n_2$`)`.

To formalize this intuition, you'll need to use the congruence lemmas from above (you might want to review them now, so that you'll be able to recognize when they are useful), plus some basic properties of `==>*`: that it is reflexive, transitive, and includes `==>`.

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

Exercise: 3 stars, advanced (eval__multistep_inf)

Write a detailed informal version of the proof of `eval__multistep`.

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

□

For the other direction, we need one lemma, which establishes a relation between single-step reduction and big-step evaluation.

Exercise: 3 stars (step__eval)

```
Lemma step__eval : ∀ t t' n,
     t ==> t' →
     t' \\ n →
     t \\ n.
Proof.
  intros t t' n Hs. generalize dependent n.
  (* FILL IN HERE *) Admitted.
```

□

The fact that small-step reduction implies big-step evaluation is now straightforward to prove, once it is stated correctly.

The proof proceeds by induction on the multi-step reduction sequence that is buried in the hypothesis `normal_form_of t t'`.

Make sure you understand the statement before you start to work on the proof.

Exercise: 3 stars (multistep__eval)

```
Theorem multistep__eval : ∀ t t',
  normal_form_of t t' → ∃ n, t' = C n ∧ t \\ n.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Additional Exercises

Exercise: 3 stars, optional (interp_tm)

Remember that we also defined big-step evaluation of terms as a function `evalF`. Prove that it is equivalent to the existing semantics. (Hint: we just proved that `eval` and `multistep` are equivalent, so logically it doesn't matter which you choose. One will be easier than the other, though!)

```
Theorem evalF_eval : ∀ t n,
  evalF t = n ↔ t \\ n.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars (combined_properties)

We've considered arithmetic and conditional expressions separately. This exercise explores how the two interact.

```
Module Combined.

Inductive tm : Type :=
  | C : nat → tm
  | P : tm → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.

Inductive value : tm → Prop :=
  | v_const : ∀ n, value (C n)
  | v_true : value ttrue
  | v_false : value tfalse.

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ v1 t2 t2',
      value v1 →
      t2 ==> t2' →
      P v1 t2 ==> P v1 t2'
  | ST_IfTrue : ∀ t1 t2,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      tif t1 t2 t3 ==> tif t1' t2 t3

  where " t '==>' t' " := (step t t').
```

Earlier, we separately proved for both plus- and if-expressions...

- that the step relation was deterministic, and
- a strong progress lemma, stating that every term is either a value or can take a step.

Formally prove or disprove these two properties for the combined language. (That is, state a theorem saying that the property holds or does not hold, and prove your theorem.)

```
(* FILL IN HERE *)

End Combined.
```
☐

Small-Step Imp

Now for a more serious example: a small-step version of the Imp operational semantics.

The small-step reduction relations for arithmetic and boolean expressions are straightforward extensions of the tiny language we've been working up to now. To make them easier to read, we introduce the symbolic notations ==>a and ==>b for the arithmetic and boolean step relations.

```
Inductive aval : aexp → Prop :=
  | av_num : ∀ n, aval (ANum n).
```

We are not actually going to bother to define boolean values, since they aren't needed in the definition of ==>b below (why?), though they might be if our language were a bit larger (why?).

```
Reserved Notation " t '/' st '==>a' t' "
                  (at level 40, st at level 39).

Inductive astep : state → aexp → aexp → Prop :=
  | AS_Id : ∀ st i,
      AId i / st ==>a ANum (st i)
  | AS_Plus : ∀ st n1 n2,
      APlus (ANum n1) (ANum n2) / st ==>a ANum (n1 + n2)
  | AS_Plus1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (APlus a1 a2) / st ==>a (APlus a1' a2)
  | AS_Plus2 : ∀ st v1 a2 a2',
      aval v1 →
      a2 / st ==>a a2' →
      (APlus v1 a2) / st ==>a (APlus v1 a2')
  | AS_Minus : ∀ st n1 n2,
      (AMinus (ANum n1) (ANum n2)) / st ==>a (ANum (minus n1 n2))
  | AS_Minus1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (AMinus a1 a2) / st ==>a (AMinus a1' a2)
  | AS_Minus2 : ∀ st v1 a2 a2',
      aval v1 →
      a2 / st ==>a a2' →
      (AMinus v1 a2) / st ==>a (AMinus v1 a2')
  | AS_Mult : ∀ st n1 n2,
      (AMult (ANum n1) (ANum n2)) / st ==>a (ANum (mult n1 n2))
  | AS_Mult1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (AMult a1 a2) / st ==>a (AMult a1' a2)
  | AS_Mult2 : ∀ st v1 a2 a2',
      aval v1 →
      a₂ / st ==>a a₂' →
      (AMult v₁ a₂) / st ==>a (AMult v₁ a₂')

    where " t '/' st '==>a' t' " := (astep st t t').

Reserved Notation " t '/' st '==>b' t' "
                  (at level 40, st at level 39).

Inductive bstep : state → bexp → bexp → Prop :=
| BS_Eq : ∀ st n₁ n₂,
    (BEq (ANum n₁) (ANum n₂)) / st ==>b
    (if (beq_nat n₁ n₂) then BTrue else BFalse)
| BS_Eq₁ : ∀ st a₁ a₁' a₂,
    a₁ / st ==>a a₁' →
    (BEq a₁ a₂) / st ==>b (BEq a₁' a₂)
| BS_Eq₂ : ∀ st v₁ a₂ a₂',
    aval v₁ →
    a₂ / st ==>a a₂' →
    (BEq v₁ a₂) / st ==>b (BEq v₁ a₂')
| BS_LtEq : ∀ st n₁ n₂,
    (BLe (ANum n₁) (ANum n₂)) / st ==>b
             (if (leb n₁ n₂) then BTrue else BFalse)
| BS_LtEq1 : ∀ st a₁ a₁' a₂,
    a₁ / st ==>a a₁' →
    (BLe a₁ a₂) / st ==>b (BLe a₁' a₂)
| BS_LtEq2 : ∀ st v₁ a₂ a₂',
    aval v₁ →
    a₂ / st ==>a a₂' →
    (BLe v₁ a₂) / st ==>b (BLe v₁ a₂')
| BS_NotTrue : ∀ st,
    (BNot BTrue) / st ==>b BFalse
| BS_NotFalse : ∀ st,
    (BNot BFalse) / st ==>b BTrue
| BS_NotStep : ∀ st b₁ b₁',
    b₁ / st ==>b b₁' →
    (BNot b₁) / st ==>b (BNot b₁')
| BS_AndTrueTrue : ∀ st,
    (BAnd BTrue BTrue) / st ==>b BTrue
| BS_AndTrueFalse : ∀ st,
    (BAnd BTrue BFalse) / st ==>b BFalse
| BS_AndFalse : ∀ st b₂,
    (BAnd BFalse b₂) / st ==>b BFalse
| BS_AndTrueStep : ∀ st b₂ b₂',
    b₂ / st ==>b b₂' →
    (BAnd BTrue b₂) / st ==>b (BAnd BTrue b₂')
| BS_AndStep : ∀ st b₁ b₁' b₂,
    b₁ / st ==>b b₁' →
    (BAnd b₁ b₂) / st ==>b (BAnd b₁' b₂)
```

```
where " t '/' st '==>b' t' " := (bstep st t t').
```

The semantics of commands is the interesting part. We need two small tricks to make it work:

- We use `SKIP` as a "command value" — i.e., a command that has reached a normal form.
    - An assignment command reduces to `SKIP` (and an updated state).
    - The sequencing command waits until its left-hand subcommand has reduced to `SKIP`, then throws it away so that reduction can continue with the right-hand subcommand.
- We reduce a `WHILE` command by transforming it into a conditional followed by the same `WHILE`.

(There are other ways of achieving the effect of the latter trick, but they all share the feature that the original `WHILE` command needs to be saved somewhere while a single copy of the loop body is being reduced.)

```
Reserved Notation " t '/' st '==>' t' '/' st' "
                  (at level 40, st at level 39, t' at level 39).

Inductive cstep : (com * state) → (com * state) → Prop :=
  | CS_AssStep : ∀ st i a a',
      a / st ==>a a' →
      (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : ∀ st i n,
      (i ::= (ANum n)) / st ==> SKIP / (st & { i --> n })
  | CS_SeqStep : ∀ st c₁ c₁' st' c₂,
      c₁ / st ==> c₁' / st' →
      (c₁ ;; c₂) / st ==> (c₁' ;; c₂) / st'
  | CS_SeqFinish : ∀ st c₂,
      (SKIP ;; c₂) / st ==> c₂ / st
  | CS_IfTrue : ∀ st c₁ c₂,
      IFB BTrue THEN c₁ ELSE c₂ FI / st ==> c₁ / st
  | CS_IfFalse : ∀ st c₁ c₂,
      IFB BFalse THEN c₁ ELSE c₂ FI / st ==> c₂ / st
  | CS_IfStep : ∀ st b b' c₁ c₂,
      b / st ==>b b' →
          IFB b THEN c₁ ELSE c₂ FI / st
      ==> (IFB b' THEN c₁ ELSE c₂ FI) / st
  | CS_While : ∀ st b c₁,
          (WHILE b DO c₁ END) / st
      ==> (IFB b THEN (c₁;; (WHILE b DO c₁ END)) ELSE SKIP FI) /
st

  where " t '/' st '==>' t' '/' st' " := (cstep (t,st)
(t',st')).
```

Concurrent Imp

Finally, to show the power of this definitional style, let's enrich Imp with a new form of command that runs two subcommands in parallel and terminates when both have terminated. To reflect the unpredictability of scheduling, the actions of the subcommands may be interleaved in any order, but they share the same memory and can communicate by reading and writing the same variables.

```
Module CImp.

Inductive com : Type :=
  | CSkip : com
  | CAss : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  (* New: *)
  | CPar : com → com → com.

Notation "'SKIP'" :=
  CSkip.
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' b 'THEN' c1 'ELSE' c2 'FI'" :=
  (CIf b c1 c2) (at level 80, right associativity).
Notation "'PAR' c1 'WITH' c2 'END'" :=
  (CPar c1 c2) (at level 80, right associativity).

Inductive cstep : (com * state) → (com * state) → Prop :=
    (* Old part *)
  | CS_AssStep : ∀ st i a a',
      a / st ==>a a' →
      (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : ∀ st i n,
      (i ::= (ANum n)) / st ==> SKIP / st & { i --> n }
  | CS_SeqStep : ∀ st c1 c1' st' c2,
      c1 / st ==> c1' / st' →
      (c1 ;; c2) / st ==> (c1' ;; c2) / st'
  | CS_SeqFinish : ∀ st c2,
      (SKIP ;; c2) / st ==> c2 / st
  | CS_IfTrue : ∀ st c1 c2,
      (IFB BTrue THEN c1 ELSE c2 FI) / st ==> c1 / st
  | CS_IfFalse : ∀ st c1 c2,
      (IFB BFalse THEN c1 ELSE c2 FI) / st ==> c2 / st
  | CS_IfStep : ∀ st b b' c1 c2,
      b /st ==>b b' →
          (IFB b THEN c1 ELSE c2 FI) / st
      ==> (IFB b' THEN c₁ ELSE c₂ FI) / st
  | CS_While : ∀ st b c₁,
          (WHILE b DO c₁ END) / st
      ==> (IFB b THEN (c₁;; (WHILE b DO c₁ END)) ELSE SKIP FI) /
st
    (* New part: *)
  | CS_Par1 : ∀ st c₁ c₁' c₂ st',
      c₁ / st ==> c₁' / st' →
      (PAR c₁ WITH c₂ END) / st ==> (PAR c₁' WITH c₂ END) / st'
  | CS_Par2 : ∀ st c₁ c₂ c₂' st',
      c₂ / st ==> c₂' / st' →
      (PAR c₁ WITH c₂ END) / st ==> (PAR c₁ WITH c₂' END) / st'
  | CS_ParDone : ∀ st,
      (PAR SKIP WITH SKIP END) / st ==> SKIP / st
  where " t '/' st '==>' t' '/' st' " := (cstep (t,st)
(t',st')).

Definition cmultistep := multi cstep.

Notation " t '/' st '==>*' t' '/' st' " :=
   (multi cstep (t,st) (t',st'))
   (at level 40, st at level 39, t' at level 39).
```

Among the many interesting properties of this language is the fact that the following program can terminate with the variable `X` set to any value.

```
Definition par_loop : com :=
  PAR
    Y ::= 1
  WITH
    WHILE Y = 0 DO
      X ::= X + 1
    END
  END.
```

In particular, it can terminate with `X` set to `0`:

```
Example par_loop_example_0:
  ∃ st',
       par_loop / { --> 0 } ==>* SKIP / st'
    ∧ st' X = 0.
```
+

It can also terminate with `X` set to `2`:

```
Example par_loop_example_2:
  ∃ st',
       par_loop / { --> 0 } ==>* SKIP / st'
    ∧ st' X = 2.
```
+

More generally...

Exercise: 3 stars, optional (par_body_n__Sn)

```
Lemma par_body_n__Sn : ∀ n st,
  st X = n ∧ st Y = 0 →
  par_loop / st ==>* par_loop / st & { X --> S n}.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, optional (par_body_n)

```
Lemma par_body_n : ∀ n st,
  st X = 0 ∧ st Y = 0 →
  ∃ st',
    par_loop / st ==>* par_loop / st' ∧ st' X = n ∧ st' Y = 0.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

... the above loop can exit with X having any value whatsoever.

```
Theorem par_loop_any_X:
  ∀ n, ∃ st',
    par_loop / { --> 0 } ==>* SKIP / st'
    ∧ st' X = n.
+

End CImp.
```

A Small-Step Stack Machine

Our last example is a small-step semantics for the stack machine example from the Imp chapter of *Logical Foundations*.

```
Definition stack := list nat.
Definition prog := list sinstr.

Inductive stack_step : state → prog * stack → prog * stack →
Prop :=
  | SS_Push : ∀ st stk n p',
    stack_step st (SPush n :: p', stk) (p', n :: stk)
  | SS_Load : ∀ st stk i p',
    stack_step st (SLoad i :: p', stk) (p', st i :: stk)
  | SS_Plus : ∀ st stk n m p',
    stack_step st (SPlus :: p', n::m::stk) (p', (m+n)::stk)
  | SS_Minus : ∀ st stk n m p',
    stack_step st (SMinus :: p', n::m::stk) (p', (m-n)::stk)
  | SS_Mult : ∀ st stk n m p',
    stack_step st (SMult :: p', n::m::stk) (p', (m*n)::stk).

Theorem stack_step_deterministic : ∀ st,
  deterministic (stack_step st).
+

Definition stack_multistep st := multi (stack_step st).
```

Exercise: 3 stars, advanced (compiler_is_correct)

Remember the definition of `compile` for `aexp` given in the Imp chapter of *Logical Foundations*. We want now to prove `compile` correct with respect to the stack machine.

State what it means for the compiler to be correct according to the stack machine small step semantics and then prove it.

```
Definition compiler_is_correct_statement : Prop
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Theorem compiler_is_correct : compiler_is_correct_statement.
Proof.
(* FILL IN HERE *) Admitted.
```
□