EQUIV

PROGRAM EQUIVALENCE

```
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.Logic.FunctionalExtensionality.
Import ListNotations.
Require Import Maps.
Require Import Imp.
```

Some Advice for Working on Exercises:

- Most of the Coq proofs we ask you to do are similar to proofs that we've provided. Before starting to work on exercises problems, take the time to work through our proofs (both informally and in Coq) and make sure you understand them in detail. This will save you a lot of time.
- The Coq proofs we're doing now are sufficiently complicated that it is more or less impossible to complete them by random experimentation or "following your nose." You need to start with an idea about why the property is true and how the proof is going to go. The best way to do this is to write out at least a sketch of an informal proof on paper — one that intuitively convinces you of the truth of the theorem — before starting to work on the formal one. Alternately, grab a friend and try to convince them that the theorem is true; then try to formalize your explanation.
- Use automation to save work! The proofs in this chapter can get pretty long if you try to write out all the cases explicitly.

Behavioral Equivalence

In an earlier chapter, we investigated the correctness of a very simple program transformation: the `optimize_0plus` function. The programming language we were considering was the first version of the language of arithmetic expressions — with no variables — so in that setting it was very easy to define what it means for a program transformation to be correct: it should always yield a program that evaluates to the same number as the original.

To talk about the correctness of program transformations for the full Imp language, in particular assignment, we need to consider the role of variables and state.

Definitions

For `aexp`s and `bexp`s with variables, the definition we want is clear. We say that two `aexp`s or `bexp`s are *behaviorally equivalent* if they evaluate to the same result in every state.

```
Definition aequiv (a1 a2 : aexp) : Prop :=
  ∀ (st:state),
    aeval st a1 = aeval st a2.

Definition bequiv (b1 b2 : bexp) : Prop :=
  ∀ (st:state),
    beval st b1 = beval st b2.
```

Here are some simple examples of equivalences of arithmetic and boolean expressions.

```
Theorem aequiv_example:
  aequiv (X - X) 0.
+

Theorem bequiv_example:
  bequiv (X - X = 0) true.
+
```

For commands, the situation is a little more subtle. We can't simply say "two commands are behaviorally equivalent if they evaluate to the same ending state whenever they are started in the same initial state," because some commands, when run in some starting states, don't terminate in any final state at all! What we need instead is this: two commands are behaviorally equivalent if, for any given starting state, they either (1) both diverge or (2) both terminate in the same final state. A compact way to express this is "if the first one terminates in a particular state then so does the second, and vice versa."

```
Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state),
    (c1 / st \\ st') ↔ (c2 / st \\ st').
```

Simple Examples

For examples of command equivalence, let's start by looking at some trivial program transformations involving `SKIP`:

```
Theorem skip_left: ∀ c,
  cequiv
     (SKIP;; c)
     c.
Proof.
  (* WORKED IN CLASS *)
  intros c st st'.
  split; intros H.
  - (* -> *)
    inversion H; subst.
    inversion H2. subst.
    assumption.
  - (* <- *)
    apply E_Seq with st.
    apply E_Skip.
    assumption.
Qed.
```

Exercise: 2 stars (skip_right)

Prove that adding a `SKIP` after a command results in an equivalent program

```
Theorem skip_right: ∀ c,
  cequiv
    (c ;; SKIP)
    c.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Similarly, here is a simple transformation that optimizes `IFB` commands:

```
Theorem IFB_true_simple: ∀ c1 c2,
  cequiv
    (IFB BTrue THEN c1 ELSE c2 FI)
    c1.
```

\+

Of course, no human programmer would write a conditional whose guard is literally `BTrue`. The interesting case is when the guard is *equivalent* to true: *Theorem*: If `b` is equivalent to `BTrue`, then `IFB b THEN` c_1 `ELSE` c_2 `FI` is equivalent to c_1.

Proof:

- (→) We must show, for all `st` and `st'`, that if `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'` then c_1 `/ st \\ st'`.

  Proceed by cases on the rules that could possibly have been used to show `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'`, namely `E_IfTrue` and `E_IfFalse`.

- Suppose the final rule rule in the derivation of `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'` was `E_IfTrue`. We then have, by the premises of `E_IfTrue`, that c_1 `/ st \\ st'`. This is exactly what we set out to prove.
- On the other hand, suppose the final rule in the derivation of `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'` was `E_IfFalse`. We then know that `beval st b = false` and c_2 `/ st \\ st'`.

  Recall that `b` is equivalent to `BTrue`, i.e., forall `st`, `beval st b = beval st BTrue`. In particular, this means that `beval st b = true`, since `beval st BTrue = true`. But this is a contradiction, since `E_IfFalse` requires that `beval st b = false`. Thus, the final rule could not have been `E_IfFalse`.

- (`<-`) We must show, for all `st` and `st'`, that if c_1 `/ st \\ st'` then `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'`.

  Since `b` is equivalent to `BTrue`, we know that `beval st b` = `beval st BTrue` = `true`. Together with the assumption that c_1 `/ st \\ st'`, we can apply `E_IfTrue` to derive `IFB b THEN` c_1 `ELSE` c_2 `FI / st \\ st'`. □

Here is the formal version of this proof:

```
Theorem IFB_true: ∀ b c₁ c₂,
     bequiv b BTrue →
     cequiv
       (IFB b THEN c₁ ELSE c₂ FI)
       c₁.
```

+

Exercise: 2 stars, recommended (IFB_false)

```
Theorem IFB_false: ∀ b c₁ c₂,
  bequiv b BFalse →
  cequiv
    (IFB b THEN c₁ ELSE c₂ FI)
    c₂.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (swap_if_branches)

Show that we can swap the branches of an IF if we also negate its guard.

```
Theorem swap_if_branches: ∀ b e₁ e₂,
  cequiv
    (IFB b THEN e₁ ELSE e₂ FI)
    (IFB BNot b THEN e₂ ELSE e₁ FI).
```

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

□

For `WHILE` loops, we can give a similar pair of theorems. A loop whose guard is equivalent to `BFalse` is equivalent to `SKIP`, while a loop whose guard is equivalent to `BTrue` is equivalent to `WHILE BTrue DO SKIP END` (or any other non-terminating program). The first of these facts is easy.

```
Theorem WHILE_false : ∀ b c,
  bequiv b BFalse →
  cequiv
    (WHILE b DO c END)
    SKIP.
```

+

Exercise: 2 stars, advanced, optional (WHILE_false_informal)

Write an informal proof of `WHILE_false`.

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

□

To prove the second fact, we need an auxiliary lemma stating that `WHILE` loops whose guards are equivalent to `BTrue` never terminate.

Lemma: If `b` is equivalent to `BTrue`, then it cannot be the case that `(WHILE b DO c END) / st \\ st'`.

Proof: Suppose that `(WHILE b DO c END) / st \\ st'`. We show, by induction on a derivation of `(WHILE b DO c END) / st \\ st'`, that this assumption leads to a contradiction.

- Suppose `(WHILE b DO c END) / st \\ st'` is proved using rule `E_WhileFalse`. Then by assumption `beval st b = false`. But this contradicts the assumption that `b` is equivalent to `BTrue`.
- Suppose `(WHILE b DO c END) / st \\ st'` is proved using rule `E_WhileTrue`. Then we are given the induction hypothesis that `(WHILE b DO c END) / st \\ st'` is contradictory, which is exactly what we are trying to prove!
- Since these are the only rules that could have been used to prove `(WHILE b DO c END) / st \\ st'`, the other cases of the induction are immediately contradictory. □

```
Lemma WHILE_true_nonterm : ∀ b c st st',
  bequiv b BTrue →
  ~( (WHILE b DO c END) / st \\ st' ).
Proof.
  (* WORKED IN CLASS *)
  intros b c st st' Hb.
  intros H.
  remember (WHILE b DO c END) as cw eqn:Heqcw.
  induction H;
```

```
    (* Most rules don't apply; we rule them out by inversion: *)
    inversion Heqcw; subst; clear Heqcw.
    (* The two interesting cases are the ones for WHILE loops: *)
    - (* E_WhileFalse *) (* contradictory -- b is always true! *)
      unfold bequiv in Hb.
      (* rewrite is able to instantiate the quantifier in st *)
      rewrite Hb in H. inversion H.
    - (* E_WhileTrue *) (* immediate from the IH *)
      apply IHceval2. reflexivity. Qed.
```

Exercise: 2 stars, optional (WHILE_true_nonterm_informal)

Explain what the lemma `WHILE_true_nonterm` means in English.

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

☐

Exercise: 2 stars, recommended (WHILE_true)

Prove the following theorem. *Hint*: You'll want to use `WHILE_true_nonterm` here.

```
Theorem WHILE_true: ∀ b c,
  bequiv b true →
  cequiv
    (WHILE b DO c END)
    (WHILE true DO SKIP END).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

A more interesting fact about `WHILE` commands is that any number of copies of the body can be "unrolled" without changing meaning. Loop unrolling is a common transformation in real compilers.

```
Theorem loop_unrolling: ∀ b c,
  cequiv
    (WHILE b DO c END)
    (IFB b THEN (c ;; WHILE b DO c END) ELSE SKIP FI).
Proof.
  (* WORKED IN CLASS *)
```

+

Exercise: 2 stars, optional (seq_assoc)

```
Theorem seq_assoc : ∀ c1 c2 c3,
  cequiv ((c1;;c2);;c3) (c1;;(c2;;c3)).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Proving program properties involving assignments is one place where the Functional Extensionality axiom often comes in handy.

```
Theorem identity_assignment : ∀ (X:string),
  cequiv
    (X ::= X)
```

```
    SKIP.
Proof.
   intros. split; intro H.
     - (* -> *)
       inversion H; subst. simpl.
       replace (st & { X --> st X }) with st.
       + constructor.
       + apply functional_extensionality. intro.
         rewrite t_update_same; reflexivity.
     - (* <- *)
       replace st' with (st' & { X --> aeval st' X }).
       + inversion H. subst. apply E_Ass. reflexivity.
       + apply functional_extensionality. intro.
         rewrite t_update_same. reflexivity.
Qed.
```

Exercise: 2 stars, recommended (assign_aequiv)

```
Theorem assign_aequiv : ∀ (X:string) e,
  aequiv X e →
  cequiv SKIP (X ::= e).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (equiv_classes)

Given the following programs, group together those that are equivalent in Imp. Your answer should be given as a list of lists, where each sub-list represents a group of equivalent programs. For example, if you think programs (a) through (h) are all equivalent to each other, but not to (i), your answer should look like this:

```
[ [prog_a;prog_b;prog_c;prog_d;prog_e;prog_f;prog_g;prog_h] ;
  [prog_i] ]
```

Write down your answer below in the definition of `equiv_classes`.

```
Definition prog_a : com :=
  WHILE ! (X ≤ 0) DO
    X ::= X + 1
  END.

Definition prog_b : com :=
  IFB X = 0 THEN
    X ::= X + 1;;
    Y ::= 1
  ELSE
    Y ::= 0
  FI;;
  X ::= X - Y;;
  Y ::= 0.

Definition prog_c : com :=
  SKIP.
```

```
Definition prog_d : com :=
  WHILE ! (X = 0) DO
    X ::= (X * Y) + 1
  END.

Definition prog_e : com :=
  Y ::= 0.

Definition prog_f : com :=
  Y ::= X + 1;;
  WHILE ! (X = Y) DO
    Y ::= X + 1
  END.

Definition prog_g : com :=
  WHILE true DO
    SKIP
  END.

Definition prog_h : com :=
  WHILE ! (X = X) DO
    X ::= X + 1
  END.

Definition prog_i : com :=
  WHILE ! (X = Y) DO
    X ::= Y + 1
  END.

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

☐

Properties of Behavioral Equivalence

We next consider some fundamental properties of program equivalence.

Behavioral Equivalence Is an Equivalence

First, we verify that the equivalences on `aexps`, `bexps`, and `coms` really are *equivalences* — i.e., that they are reflexive, symmetric, and transitive. The proofs are all easy.

```
Lemma refl_aequiv : ∀ (a : aexp), aequiv a a.
```

+

```
Lemma sym_aequiv : ∀ (a1 a2 : aexp),
  aequiv a1 a2 → aequiv a2 a1.
```

+

```
Lemma trans_aequiv : ∀ (a1 a2 a3 : aexp),
  aequiv a1 a2 → aequiv a2 a3 → aequiv a1 a3.
```

+

```
Lemma refl_bequiv : ∀ (b : bexp), bequiv b b.
```

+

```
Lemma sym_bequiv : ∀ (b1 b2 : bexp),
  bequiv b1 b2 → bequiv b2 b1.
```

+

```
Lemma trans_bequiv : ∀ (b1 b2 b3 : bexp),
  bequiv b1 b2 → bequiv b2 b3 → bequiv b1 b3.
```

+

```
Lemma refl_cequiv : ∀ (c : com), cequiv c c.
```

+

```
Lemma sym_cequiv : ∀ (c1 c2 : com),
  cequiv c1 c2 → cequiv c2 c1.
```

+

```
Lemma iff_trans : ∀ (P1 P2 P3 : Prop),
  (P1 ↔ P2) → (P2 ↔ P3) → (P1 ↔ P3).
```

+

```
Lemma trans_cequiv : ∀ (c1 c2 c3 : com),
  cequiv c1 c2 → cequiv c2 c3 → cequiv c1 c3.
```

+

Behavioral Equivalence Is a Congruence

Less obviously, behavioral equivalence is also a *congruence*. That is, the equivalence of two subprograms implies the equivalence of the larger programs in which they are embedded:

$$\frac{\texttt{aequiv}\ a_1\ a_1'}{\texttt{cequiv}\ (i ::= a_1)\ (i ::= a_1')}$$

$$\frac{\texttt{cequiv}\ c_1\ c_1' \qquad \texttt{cequiv}\ c_2\ c_2'}{\texttt{cequiv}\ (c_1;;c_2)\ (c_1';;c_2')}$$

...and so on for the other forms of commands.

(Note that we are using the inference rule notation here not as part of a definition, but simply to write down some valid implications in a readable format. We prove these implications below.)

We will see a concrete example of why these congruence properties are important in the following section (in the proof of `fold_constants_com_sound`), but the main idea is that they allow us to replace a small part of a large program with an equivalent small part and know that the whole large programs are equivalent *without* doing an explicit proof about the non-varying parts — i.e., the "proof burden" of a small change to a large program is proportional to the size of the change, not the program.

```
Theorem CAss_congruence : ∀ i a₁ a₁',
  aequiv a₁ a₁' →
  cequiv (CAss i a₁) (CAss i a₁').
```

+

The congruence property for loops is a little more interesting, since it requires induction.

Theorem: Equivalence is a congruence for `WHILE` — that is, if b_1 is equivalent to b_1' and c_1 is equivalent to c_1', then `WHILE` b_1 `DO` c_1 `END` is equivalent to `WHILE` b_1' `DO` c_1' `END`.

Proof: Suppose b_1 is equivalent to b_1' and c_1 is equivalent to c_1'. We must show, for every `st` and `st'`, that `WHILE` b_1 `DO` c_1 `END / st \\ st'` iff `WHILE` b_1' `DO` c_1' `END / st \\ st'`. We consider the two directions separately.

- (→) We show that `WHILE` b_1 `DO` c_1 `END / st \\ st'` implies `WHILE` b_1' `DO` c_1' `END / st \\ st'`, by induction on a derivation of `WHILE` b_1 `DO` c_1 `END / st \\ st'`. The only nontrivial cases are when the final rule in the derivation is `E_WhileFalse` or `E_WhileTrue`.
  - `E_WhileFalse`: In this case, the form of the rule gives us `beval st` b_1 `= false` and `st = st'`. But then, since b_1 and b_1' are equivalent, we have `beval st` b_1' `= false`, and `E-WhileFalse` applies, giving us `WHILE` b_1' `DO` c_1' `END / st \\ st'`, as required.
  - `E_WhileTrue`: The form of the rule now gives us `beval st` b_1 `= true`, with c_1 `/ st \\ st'0` and `WHILE` b_1 `DO` c_1 `END / st'0 \\ st'` for some state `st'0`, with the induction hypothesis `WHILE` b_1' `DO` c_1' `END / st'0 \\ st'`.

    Since c_1 and c_1' are equivalent, we know that c_1' `/ st \\ st'0`. And since b_1 and b_1' are equivalent, we have `beval st` b_1' `= true`. Now `E-WhileTrue` applies, giving us `WHILE` b_1' `DO` c_1' `END / st \\ st'`, as required.
- (<-) Similar. □

```
Theorem CWhile_congruence : ∀ b₁ b₁' c₁ c₁',
  bequiv b₁ b₁' → cequiv c₁ c₁' →
  cequiv (WHILE b₁ DO c₁ END) (WHILE b₁' DO c₁' END).
Proof.
  (* WORKED IN CLASS *)
```

```
    unfold bequiv,cequiv.
    intros b1 b1' c1 c1' Hb1e Hc1e st st'.
    split; intros Hce.
    - (* -> *)
      remember (WHILE b1 DO c1 END) as cwhile
        eqn:Heqcwhile.
      induction Hce; inversion Heqcwhile; subst.
      + (* E_WhileFalse *)
        apply E_WhileFalse. rewrite <- Hb1e. apply H.
      + (* E_WhileTrue *)
        apply E_WhileTrue with (st' := st').
        * (* show loop runs *) rewrite <- Hb1e. apply H.
        * (* body execution *)
          apply (Hc1e st st'). apply Hce1.
        * (* subsequent loop execution *)
          apply IHHce2. reflexivity.
    - (* <- *)
      remember (WHILE b1' DO c1' END) as c'while
        eqn:Heqc'while.
      induction Hce; inversion Heqc'while; subst.
      + (* E_WhileFalse *)
        apply E_WhileFalse. rewrite → Hb1e. apply H.
      + (* E_WhileTrue *)
        apply E_WhileTrue with (st' := st').
        * (* show loop runs *) rewrite → Hb1e. apply H.
        * (* body execution *)
          apply (Hc1e st st'). apply Hce1.
        * (* subsequent loop execution *)
          apply IHHce2. reflexivity. Qed.
```

Exercise: 3 stars, optional (CSeq_congruence)

```
Theorem CSeq_congruence : ∀ c1 c1' c2 c2',
  cequiv c1 c1' → cequiv c2 c2' →
  cequiv (c1;;c2) (c1';;c2').
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (CIf_congruence)

```
Theorem CIf_congruence : ∀ b b' c1 c1' c2 c2',
  bequiv b b' → cequiv c1 c1' → cequiv c2 c2' →
  cequiv (IFB b THEN c1 ELSE c2 FI)
         (IFB b' THEN c1' ELSE c2' FI).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

For example, here are two equivalent programs and a proof of their equivalence...

```
Example congruence_example:
  cequiv
    (* Program 1: *)
    (X ::= 0;;
```

```
       IFB X = 0
       THEN
         Y ::= 0
       ELSE
         Y ::= 42
       FI)
      (* Program 2: *)
      (X ::= 0;;
       IFB X = 0
       THEN
         Y ::= X - X (* <--- changed here *)
       ELSE
         Y ::= 42
       FI).
Proof.
  apply CSeq_congruence.
    apply refl_cequiv.
    apply CIf_congruence.
      apply refl_bequiv.
      apply CAss_congruence. unfold aequiv. simpl.
        symmetry. apply minus_diag.
      apply refl_cequiv.
Qed.
```

Exercise: 3 stars, advanced, optional (not_congr)

We've shown that the `cequiv` relation is both an equivalence and a congruence on commands. Can you think of a relation on commands that is an equivalence but *not* a congruence?

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

□

Program Transformations

A *program transformation* is a function that takes a program as input and produces some variant of the program as output. Compiler optimizations such as constant folding are a canonical example, but there are many others.

A program transformation is *sound* if it preserves the behavior of the original program.

```
Definition atrans_sound (atrans : aexp → aexp) : Prop :=
  ∀ (a : aexp),
    aequiv a (atrans a).

Definition btrans_sound (btrans : bexp → bexp) : Prop :=
  ∀ (b : bexp),
    bequiv b (btrans b).

Definition ctrans_sound (ctrans : com → com) : Prop :=
  ∀ (c : com),
    cequiv c (ctrans c).
```

The Constant-Folding Transformation

An expression is *constant* when it contains no variable references.

Constant folding is an optimization that finds constant expressions and replaces them by their values.

```
Fixpoint fold_constants_aexp (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | AId i ⇒ AId i
  | APlus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
    | (a1', a2') ⇒ APlus a1' a2'
    end
  | AMinus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
    | (a1', a2') ⇒ AMinus a1' a2'
    end
  | AMult a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 * n2)
    | (a1', a2') ⇒ AMult a1' a2'
    end
  end.

(* needed for parsing the examples below *)
Local Open Scope aexp_scope.
Local Open Scope bexp_scope.

Example fold_aexp_ex1 :
    fold_constants_aexp ((1 + 2) * X) = (3 * X).
+
```

Note that this version of constant folding doesn't eliminate trivial additions, etc. — we are focusing attention on a single optimization for the sake of simplicity. It is not hard to incorporate other ways of simplifying expressions; the definitions and proofs just get longer.

```
Example fold_aexp_ex2 :
  fold_constants_aexp (X - ((0 * 6) + Y)) = (X - (0 + Y)).
+
```

Not only can we lift `fold_constants_aexp` to `bexp`s (in the `BEq` and `BLe` cases); we can also look for constant *boolean* expressions and evaluate them in-place.

```
Fixpoint fold_constants_bexp (b : bexp) : bexp :=
  match b with
  | BTrue ⇒ BTrue
  | BFalse ⇒ BFalse
```

```
    | BEq a1 a2 ⇒
      match (fold_constants_aexp a1, fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if beq_nat n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒
          BEq a1' a2'
      end
    | BLe a1 a2 ⇒
      match (fold_constants_aexp a1, fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if leb n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒
          BLe a1' a2'
      end
    | BNot b1 ⇒
      match (fold_constants_bexp b1) with
      | BTrue ⇒ BFalse
      | BFalse ⇒ BTrue
      | b1' ⇒ BNot b1'
      end
    | BAnd b1 b2 ⇒
      match (fold_constants_bexp b1, fold_constants_bexp b2) with
      | (BTrue, BTrue) ⇒ BTrue
      | (BTrue, BFalse) ⇒ BFalse
      | (BFalse, BTrue) ⇒ BFalse
      | (BFalse, BFalse) ⇒ BFalse
      | (b1', b2') ⇒ BAnd b1' b2'
      end
  end.

Example fold_bexp_ex1 :
  fold_constants_bexp (true && ! (false && true)) = true.
+

Example fold_bexp_ex2 :
  fold_constants_bexp ((X = Y) && (0 = (2 - (1 + 1)))) =
  ((X = Y) && true).
+
```

To fold constants in a command, we apply the appropriate folding functions on all embedded expressions.

```
Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | SKIP ⇒
      SKIP
  | i ::= a ⇒
      CAss i (fold_constants_aexp a)
  | c1 ;; c2 ⇒
      (fold_constants_com c1) ;; (fold_constants_com c2)
  | IFB b THEN c1 ELSE c2 FI ⇒
      match fold_constants_bexp b with
```

```
          | BTrue ⇒ fold_constants_com c₁
          | BFalse ⇒ fold_constants_com c₂
          | b' ⇒ IFB b' THEN fold_constants_com c₁
                        ELSE fold_constants_com c₂ FI
         end
      | WHILE b DO c END ⇒
         match fold_constants_bexp b with
          | BTrue ⇒ WHILE BTrue DO SKIP END
          | BFalse ⇒ SKIP
          | b' ⇒ WHILE b' DO (fold_constants_com c) END
         end
     end.
```

```
Example fold_com_ex₁ :
  fold_constants_com
    (* Original program: *)
    (X ::= 4 + 5;;
     Y ::= X - 3;;
     IFB (X - Y) = (2 + 4) THEN
       SKIP
     ELSE
       Y ::= 0
     FI;;
     IFB 0 ≤ (4 - (2 + 1))
     THEN
       Y ::= 0
     ELSE
       SKIP
     FI;;
     WHILE Y = 0 DO
       X ::= X + 1
     END)
  = (* After constant folding: *)
    (X ::= 9;;
     Y ::= X - 3;;
     IFB (X - Y) = 6 THEN
       SKIP
     ELSE
       Y ::= 0
     FI;;
     Y ::= 0;;
     WHILE Y = 0 DO
       X ::= X + 1
     END).
```

+

Soundness of Constant Folding

Now we need to show that what we've done is correct.

Here's the proof for arithmetic expressions:

```
Theorem fold_constants_aexp_sound :
  atrans_sound fold_constants_aexp.
```

+

Exercise: 3 stars, optional (fold_bexp_Eq_informal)

Here is an informal proof of the `BEq` case of the soundness argument for boolean expression constant folding. Read it carefully and compare it to the formal proof that follows. Then fill in the `BLe` case of the formal proof (without looking at the `BEq` case, if possible).

Theorem: The constant folding function for booleans, `fold_constants_bexp`, is sound.

Proof: We must show that `b` is equivalent to `fold_constants_bexp`, for all boolean expressions `b`. Proceed by induction on `b`. We show just the case where `b` has the form `BEq` a_1 a_2.

In this case, we must show

```
  beval st (BEq a1 a2)
= beval st (fold_constants_bexp (BEq a1 a2)).
```

There are two cases to consider:

- First, suppose `fold_constants_aexp` a_1 = `ANum` n_1 and `fold_constants_aexp` a_2 = `ANum` n_2 for some n_1 and n_2.

  In this case, we have

  ```
    fold_constants_bexp (BEq a1 a2)
  = if beq_nat n1 n2 then BTrue else BFalse
  ```

  and

  ```
    beval st (BEq a1 a2)
  = beq_nat (aeval st a1) (aeval st a2).
  ```

  By the soundness of constant folding for arithmetic expressions (Lemma `fold_constants_aexp_sound`), we know

  ```
    aeval st a1
  = aeval st (fold_constants_aexp a1)
  = aeval st (ANum n1)
  = n1
  ```

  and

  ```
    aeval st a2
  = aeval st (fold_constants_aexp a2)
  = aeval st (ANum n2)
  = n2,
  ```

  so

```
    beval st (BEq a₁ a₂)
  = beq_nat (aeval a₁) (aeval a₂)
  = beq_nat n₁ n₂.
```

Also, it is easy to see (by considering the cases $n_1 = n_2$ and $n_1 \neq n_2$ separately) that

```
    beval st (if beq_nat n₁ n₂ then BTrue else BFalse)
  = if beq_nat n₁ n₂ then beval st BTrue else beval st BFalse
  = if beq_nat n₁ n₂ then true else false
  = beq_nat n₁ n₂.
```

So

```
    beval st (BEq a₁ a₂)
  = beq_nat n₁ n₂.
  = beval st (if beq_nat n₁ n₂ then BTrue else BFalse),
```

as required.

- Otherwise, one of `fold_constants_aexp` a_1 and `fold_constants_aexp` a_2 is not a constant. In this case, we must show

```
    beval st (BEq a₁ a₂)
  = beval st (BEq (fold_constants_aexp a₁)
                  (fold_constants_aexp a₂)),
```

which, by the definition of `beval`, is the same as showing

```
    beq_nat (aeval st a₁) (aeval st a₂)
  = beq_nat (aeval st (fold_constants_aexp a₁))
            (aeval st (fold_constants_aexp a₂)).
```

But the soundness of constant folding for arithmetic expressions (`fold_constants_aexp_sound`) gives us

```
  aeval st a₁ = aeval st (fold_constants_aexp a₁)
  aeval st a₂ = aeval st (fold_constants_aexp a₂),
```

completing the case. □

```
Theorem fold_constants_bexp_sound:
  btrans_sound fold_constants_bexp.
Proof.
  unfold btrans_sound. intros b. unfold bequiv. intros st.
  induction b;
    (* BTrue and BFalse are immediate *)
    try reflexivity.
  - (* BEq *)
    rename a into a₁. rename a₀ into a₂. simpl.
```

(Doing induction when there are a lot of constructors makes specifying variable names a chore, but Coq doesn't always choose nice variable names. We can rename entries in the context with the `rename` tactic: `rename a into a1` will change `a` to `a1` in the current goal and context.)

```
      remember (fold_constants_aexp a1) as a1' eqn:Heqa1'.
      remember (fold_constants_aexp a2) as a2' eqn:Heqa2'.
      replace (aeval st a1) with (aeval st a1') by
         (subst a1'; rewrite <- fold_constants_aexp_sound;
reflexivity).
      replace (aeval st a2) with (aeval st a2') by
         (subst a2'; rewrite <- fold_constants_aexp_sound;
reflexivity).
      destruct a1'; destruct a2'; try reflexivity.

      (* The only interesting case is when both a1 and a2
         become constants after folding *)
        simpl. destruct (beq_nat n n0); reflexivity.
    - (* BLe *)
      (* FILL IN HERE *) admit.
    - (* BNot *)
      simpl. remember (fold_constants_bexp b) as b' eqn:Heqb'.
      rewrite IHb.
      destruct b'; reflexivity.
    - (* BAnd *)
      simpl.
      remember (fold_constants_bexp b1) as b1' eqn:Heqb1'.
      remember (fold_constants_bexp b2) as b2' eqn:Heqb2'.
      rewrite IHb1. rewrite IHb2.
      destruct b1'; destruct b2'; reflexivity.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (fold_constants_com_sound)

Complete the `WHILE` case of the following proof.

```
Theorem fold_constants_com_sound :
  ctrans_sound fold_constants_com.
Proof.
  unfold ctrans_sound. intros c.
  induction c; simpl.
  - (* SKIP *) apply refl_cequiv.
  - (* ::= *) apply CAss_congruence.
              apply fold_constants_aexp_sound.
  - (* ;; *) apply CSeq_congruence; assumption.
  - (* IFB *)
    assert (bequiv b (fold_constants_bexp b)). {
      apply fold_constants_bexp_sound. }
    destruct (fold_constants_bexp b) eqn:Heqb;
      try (apply CIf_congruence; assumption).
      (* (If the optimization doesn't eliminate the if, then the
          result is easy to prove from the IH and
```

```
          fold_constants_bexp_sound.) *)
    + (* b always true *)
      apply trans_cequiv with c1; try assumption.
      apply IFB_true; assumption.
    + (* b always false *)
      apply trans_cequiv with c2; try assumption.
      apply IFB_false; assumption.
  - (* WHILE *)
    (* FILL IN HERE *) Admitted.
```

☐

Soundness of (0 + n) Elimination, Redux

Exercise: 4 stars, advanced, optional (optimize_0plus)

Recall the definition `optimize_0plus` from the Imp chapter of *Logical Foundations*:

```
Fixpoint optimize_0plus (e:aexp) : aexp :=
  match e 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.
```

Note that this function is defined over the old `aexp`s, without states.

Write a new version of this function that accounts for variables, plus analogous ones for `bexp`s and commands:

```
optimize_0plus_aexp
optimize_0plus_bexp
optimize_0plus_com
```

Prove that these three functions are sound, as we did for `fold_constants_*`. Make sure you use the congruence lemmas in the proof of `optimize_0plus_com` — otherwise it will be *long*!

Then define an optimizer on commands that first folds constants (using `fold_constants_com`) and then eliminates `0 + n` terms (using `optimize_0plus_com`).

- Give a meaningful example of this optimizer's output.

- Prove that the optimizer is sound. (This part should be *very* easy.)

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

□

Proving Inequivalence

Suppose that c_1 is a command of the form `X ::= a1;; Y ::= a2` and c_2 is the command `X ::= a1;; Y ::= a2'`, where `a2'` is formed by substituting a_1 for all occurrences of `X` in a_2. For example, c_1 and c_2 might be:

```
c1  =  (X ::= 42 + 53;;
        Y ::= Y + X)
c2  =  (X ::= 42 + 53;;
        Y ::= Y + (42 + 53))
```

Clearly, this *particular* c_1 and c_2 are equivalent. Is this true in general?

We will see in a moment that it is not, but it is worthwhile to pause, now, and see if you can find a counter-example on your own.

More formally, here is the function that substitutes an arithmetic expression for each occurrence of a given variable in another expression:

```
Fixpoint subst_aexp (i : string) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒
      ANum n
  | AId i' ⇒
      if beq_string i i' then u else AId i'
  | APlus a1 a2 ⇒
      APlus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMinus a1 a2 ⇒
      AMinus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMult a1 a2 ⇒
      AMult (subst_aexp i u a1) (subst_aexp i u a2)
  end.

Example subst_aexp_ex :
  subst_aexp X (42 + 53) (Y + X)
  = (Y + (42 + 53)).
```

+

And here is the property we are interested in, expressing the claim that commands c_1 and c_2 as described above are always equivalent.

```
Definition subst_equiv_property := ∀ i1 i2 a1 a2,
  cequiv (i1 ::= a1;; i2 ::= a2)
         (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).
```

Sadly, the property does *not* always hold — i.e., it is not the case that, for all i_1, i_2, a_1, and a_2,

```
cequiv (i₁ ::= a₁;; i₂ ::= a₂)
       (i₁ ::= a₁;; i₂ ::= subst_aexp i₁ a₁ a₂).
```

To see this, suppose (for a contradiction) that for all i_1, i_2, a_1, and a_2, we have

```
cequiv (i₁ ::= a₁;; i₂ ::= a₂)
       (i₁ ::= a₁;; i₂ ::= subst_aexp i₁ a₁ a₂).
```

Consider the following program:

```
X ::= X + 1;; Y ::= X
```

Note that

```
(X ::= X + 1;; Y ::= X)
/ { --> 0 } \\ st₁,
```

where st_1 = { X --> 1; Y --> 1 }.

By assumption, we know that

```
cequiv (X ::= X + 1;;
        Y ::= X)
       (X ::= X + 1;;
        Y ::= X + 1)
```

so, by the definition of `cequiv`, we have

```
(X ::= X + 1;; Y ::= X + 1
/ { --> 0 } \\ st₁.
```

But we can also derive

```
(X ::= X + 1;; Y ::= X + 1)
/ { --> 0 } \\ st₂,
```

where st_2 = { X --> 1; Y --> 2 }. But $st_1 \neq st_2$, which is a contradiction, since `ceval` is deterministic! □

```
Theorem subst_inequiv :
  ¬ subst_equiv_property.
```

+

Exercise: 4 stars, optional (better_subst_equiv)

The equivalence we had in mind above was not complete nonsense — it was actually almost right. To make it correct, we just need to exclude the case where the variable X occurs in the right-hand-side of the first assignment statement.

```
Inductive var_not_used_in_aexp (X:string) : aexp → Prop :=
  | VNUNum: ∀ n, var_not_used_in_aexp X (ANum n)
  | VNUId: ∀ Y, X ≠ Y → var_not_used_in_aexp X (AId Y)
  | VNUPlus: ∀ a1 a2,
      var_not_used_in_aexp X a1 →
      var_not_used_in_aexp X a2 →
      var_not_used_in_aexp X (APlus a1 a2)
  | VNUMinus: ∀ a1 a2,
      var_not_used_in_aexp X a1 →
      var_not_used_in_aexp X a2 →
      var_not_used_in_aexp X (AMinus a1 a2)
  | VNUMult: ∀ a1 a2,
      var_not_used_in_aexp X a1 →
      var_not_used_in_aexp X a2 →
      var_not_used_in_aexp X (AMult a1 a2).

Lemma aeval_weakening : ∀ i st a ni,
  var_not_used_in_aexp i a →
  aeval (st & { i --> ni }) a = aeval st a.
Proof.
  (* FILL IN HERE *) Admitted.
```

Using `var_not_used_in_aexp`, formalize and prove a correct verson of `subst_equiv_property`.

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

□

Exercise: 3 stars (inequiv_exercise)

Prove that an infinite loop is not equivalent to `SKIP`

```
Theorem inequiv_exercise:
  ¬ cequiv (WHILE true DO SKIP END) SKIP.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Extended Exercise: Nondeterministic Imp

As we have seen (in theorem `ceval_deterministic` in the `Imp` chapter), Imp's evaluation relation is deterministic. However, *non*-determinism is an important part of the definition of many real programming languages. For example, in many imperative languages (such as C and its relatives), the order in which function arguments are evaluated is unspecified. The program fragment

```
x = 0;;
f(++x, x)
```

might call `f` with arguments `(1, 0)` or `(1, 1)`, depending how the compiler chooses to order things. This can be a little confusing for programmers, but it gives the compiler writer useful freedom.

In this exercise, we will extend Imp with a simple nondeterministic command and study how this change affects program equivalence. The new command has the syntax `HAVOC X`, where `X` is an identifier. The effect of executing `HAVOC X` is to assign an *arbitrary* number to the variable `X`, nondeterministically. For example, after executing the program:

```
HAVOC Y;;
Z ::= Y * 2
```

the value of `Y` can be any number, while the value of `Z` is twice that of `Y` (so `Z` is always even). Note that we are not saying anything about the *probabilities* of the outcomes — just that there are (infinitely) many different outcomes that can possibly happen after executing this nondeterministic code.

In a sense, a variable on which we do `HAVOC` roughly corresponds to an unitialized variable in a low-level language like C. After the `HAVOC`, the variable holds a fixed but arbitrary number. Most sources of nondeterminism in language definitions are there precisely because programmers don't care which choice is made (and so it is good to leave it open to the compiler to choose whichever will run faster).

We call this new language *Himp* (``Imp extended with `HAVOC`'').

```
Module Himp.
```

To formalize Himp, we first add a clause to the definition of commands.

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

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' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'HAVOC' l" := (CHavoc l) (at level 60).
```

Exercise: 2 stars (himp_ceval)

Now, we must extend the operational semantics. We have provided a template for the `ceval` relation below, specifying the big-step semantics. What rule(s) must be added to the definition of `ceval` to formalize the behavior of the `HAVOC` command?

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

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

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

As a sanity check, the following claims should be provable for your definition:

```
Example havoc_example1 : (HAVOC X) / { --> 0 } \\ { X --> 0 }.
Proof.
(* FILL IN HERE *) Admitted.

Example havoc_example2 :
  (SKIP;; HAVOC Z) / { --> 0 } \\ { Z --> 42 }.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Finally, we repeat the definition of command equivalence from above:

```
Definition cequiv (c₁ c₂ : com) : Prop := ∀ st st' : state,
  c₁ / st \\ st' ↔ c₂ / st \\ st'.
```

Let's apply this definition to prove some nondeterministic programs equivalent / inequivalent.

Exercise: 3 stars (havoc_swap)

Are the following two programs equivalent?

```
Definition pXY :=
  HAVOC X;; HAVOC Y.

Definition pYX :=
  HAVOC Y;; HAVOC X.
```

If you think they are equivalent, prove it. If you think they are not, prove that.

```
Theorem pXY_cequiv_pYX :
  cequiv pXY pYX ∨ ¬cequiv pXY pYX.
Proof. (* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, optional (havoc_copy)

Are the following two programs equivalent?

```
Definition ptwice :=
  HAVOC X;; HAVOC Y.

Definition pcopy :=
  HAVOC X;; Y ::= X.
```

If you think they are equivalent, then prove it. If you think they are not, then prove that. (Hint: You may find the `assert` tactic useful.)

```
Theorem ptwice_cequiv_pcopy :
  cequiv ptwice pcopy ∨ ¬cequiv ptwice pcopy.
Proof. (* FILL IN HERE *) Admitted.
```

☐

The definition of program equivalence we are using here has some subtle consequences on programs that may loop forever. What `cequiv` says is that the set of possible *terminating* outcomes of two equivalent programs is the same. However, in a language with nondeterminism, like Himp, some programs always terminate, some programs always diverge, and some programs can nondeterministically terminate in some runs and diverge in others. The final part of the following exercise illustrates this phenomenon.

Exercise: 4 stars, advanced (p1_p2_term)

Consider the following commands:

```
Definition p₁ : com :=
  WHILE ! (X = 0) DO
    HAVOC Y;;
```

```
    X ::= X + 1
  END.

Definition p2 : com :=
  WHILE ! (X = 0) DO
    SKIP
  END.
```

Intuitively, p_1 and p_2 have the same termination behavior: either they loop forever, or they terminate in the same state they started in. We can capture the termination behavior of p_1 and p_2 individually with these lemmas:

```
Lemma p1_may_diverge : ∀ st st', st X ≠ 0 →
  ¬ p1 / st \\ st'.
Proof. (* FILL IN HERE *) Admitted.

Lemma p2_may_diverge : ∀ st st', st X ≠ 0 →
  ¬ p2 / st \\ st'.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, advanced (p1_p2_equiv)

Use these two lemmas to prove that p_1 and p_2 are actually equivalent.

```
Theorem p1_p2_equiv : cequiv p1 p2.
Proof. (* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, advanced (p3_p4_inequiv)

Prove that the following programs are *not* equivalent. (Hint: What should the value of `Z` be when p_3 terminates? What about p_4?)

```
Definition p3 : com :=
  Z ::= 1;;
  WHILE ! (X = 0) DO
    HAVOC X;;
    HAVOC Z
  END.

Definition p4 : com :=
  X ::= 0;;
  Z ::= 1.

Theorem p3_p4_inequiv : ¬ cequiv p3 p4.
Proof. (* FILL IN HERE *) Admitted.
```

☐

Exercise: 5 stars, advanced, optional (p5_p6_equiv)

Prove that the following commands are equivalent. (Hint: As mentioned above, our definition of `cequiv` for Himp only takes into account the sets of possible terminating configurations: two programs are equivalent if and only if when given a same starting state `st`, the set of possible terminating states is the same for both programs. If p_5 terminates, what should the final state be? Conversely, is it always possible to make p_5 terminate?)

```
Definition p5 : com :=
  WHILE ! (X = 1) DO
    HAVOC X
  END.

Definition p6 : com :=
  X ::= 1.

Theorem p5_p6_equiv : cequiv p5 p6.
Proof. (* FILL IN HERE *) Admitted.
```

☐

```
End Himp.
```

Additional Exercises

Exercise: 4 stars, optional (for_while_equiv)

This exercise extends the optional `add_for_loop` exercise from the Imp chapter, where you were asked to extend the language of commands with C-style `for` loops. Prove that the command:

```
for (c1 ; b ; c2) {
    c3
}
```

is equivalent to:

```
c1 ;
WHILE b DO
  c3 ;
  c2
END
```

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

☐

Exercise: 3 stars, optional (swap_noninterfering_assignments)

(Hint: You'll need `functional_extensionality` for this one.)

```
Theorem swap_noninterfering_assignments: ∀ l1 l2 a1 a2,
  l1 ≠ l2 →
  var_not_used_in_aexp l1 a2 →
  var_not_used_in_aexp l2 a1 →
```

```
    cequiv
      (l₁ ::= a₁;; l₂ ::= a₂)
      (l₂ ::= a₂;; l₁ ::= a₁).
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, advanced, optional (capprox)

In this exercise we define an asymmetric variant of program equivalence we call *program approximation*. We say that a program c_1 *approximates* a program c_2 when, for each of the initial states for which c_1 terminates, c_2 also terminates and produces the same final state. Formally, program approximation is defined as follows:

```
Definition capprox (c₁ c₂ : com) : Prop := ∀ (st st' : state),
  c₁ / st \\ st' → c₂ / st \\ st'.
```

For example, the program c_1 = `WHILE !(X = 1) DO X ::= X - 1 END` approximates c_2 = `X ::= 1`, but c_2 does not approximate c_1 since c_1 does not terminate when `X = 0` but c_2 does. If two programs approximate each other in both directions, then they are equivalent.

Find two programs c_3 and c_4 such that neither approximates the other.

```
Definition c₃ : com
(* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
Definition c₄ : com
(* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.

Theorem c3_c4_different : ¬ capprox c₃ c₄ ∧ ¬ capprox c₄ c₃.
Proof. (* FILL IN HERE *) Admitted.
```

Find a program `cmin` that approximates every other program.

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

Theorem cmin_minimal : ∀ c, capprox cmin c.
Proof. (* FILL IN HERE *) Admitted.
```

Finally, find a non-trivial property which is preserved by program approximation (when going from left to right).

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

Theorem zprop_preserving : ∀ c c',
  zprop c → capprox c c' → zprop c'.
Proof. (* FILL IN HERE *) Admitted.
```

☐