CIS 500 — Software Foundations

Midterm II

(Standard and advanced versions together)

November 7, 2013

Answer key

1. (16 points)

Multiple choice. Mark *all* correct answers—there may be zero or more than one. The definition of Hoare triples is given on page 15, for reference.

(a) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
X ::= 3;;
Y ::= X + Y
{{ X=3 /\ Y=5 }}
```

`P` is . . .

- `X = 3 /\ Y = 2` ← *this one*
- `X = 2 /\ Y = 3`
- `X = 2 /\ Y = 2` ← *this one*
- `X = 3 /\ Y = 3`

(b) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
X ::= X + Y;;
Y ::= X - Y
{{ X=m /\ Y=n }}
```

`P` is . . .

- `True`
- `X = n /\ Y = m`
- `X = n /\ X + Y = m` ← *this one*
- `False` ← *this one*

(c) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
WHILE X <= Y DO
  X = X + 1;;
  Y = Y + 1
END
{{ False }}
```

`P` is ...

- `X = 0 /\ Y = 1` ← *this one*
- `2*X <= 2*Y` ← *this one*
- `X+1 <= Y+2`
- `Y = X+3` ← *this one*

(d) Which instances of assertion `Q` make the following Hoare triple valid?

```
{{ X = Y }}
X ::= X + Y;;
Y ::= X - Y
{{ Q }}
```

`Q` is ...

- `True` ← *this one*
- `X = Y + Y` ← *this one*
- `Y = X - Y` ← *this one*
- `False`

Grading scheme: 1 point for each bullet.

2. (12 points) Given the following programs, group together those that are equivalent in Imp by drawing boxes around their names. 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: $\boxed{a, b, c, d, e, f, g, h}$ $\boxed{i}$.

The definition of program equivalence is repeated on page 15, for reference.

(a)
```
SKIP
```

(b)
```
X ::= X + 1;;
Y ::= 0
```

(c)
```
WHILE Y <> 0 DO
  Y ::= Y - 1;;
  X ::= X + 1
END
```

(d)
```
WHILE Y = Y DO
  X ::= X + 1
END;;
Y ::= 0
```

(e)
```
WHILE Y <> 0 DO
  Y ::= Y - 1
END;;
X ::= X + 1
```

(f)
```
X ::= X + Y;;
WHILE Y <> 0 DO
  Y ::= Y - 2
END
```

(g)
```
IFB X <> 0 \/ Y <> 0 THEN
  SKIP
ELSE
  Y ::= X
FI
```

(h)
```
X := X;;
Y := Y;;
Z := Z
```

(i)
```
WHILE True DO
  SKIP
END
```

Answer:
$\boxed{a, g, h}$ $\boxed{b, e}$ $\boxed{c, f}$ $\boxed{d, i}$

Grading scheme: -2 points for each group in the correct answer that did not appear in the student's answer

3. **[Standard]** (6 points) Recall that the notion of program equivalence (`cequiv`) is also a *congruence* relation. Briefly explain what a congruence relation is (in the context of Imp programs) why it is relevant to program optimization that `cequiv` is a congruence relation.

(a) A congruence relation is ...

A relation that is preserved by (or respects) the syntactic structure of the Imp language.

(b) The fact that `cequiv` is a congruence is useful for program optimizations because ...

if `c1` is equivalent to some optimized version `c2` we may replace any occurrence of the subprogram `c1` by `c2` without changing the meaning of the resulting program.

Grading scheme: 3 points for each part. Partial credit possible, particulary for part (b).

4. **[Standard]** (6 points) Are the following Hoare logic rules of inference valid? If not, give a counterexample.

(a)

$$\frac{\{\!\{ P \}\!\}\ \texttt{IF b THEN c1 ELSE c2 FI}\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\}}\ (\texttt{hoare\_revif})$$

Invalid: Counterexample: `b = BFalse`, `c1 = X ::= 1`, `c2 = X ::= 0`, $P = True$, $Q =$`X = 0`.

(b)

$$\frac{\{\!\{ P \}\!\}\ \texttt{c}\ \{\!\{ P \}\!\}}{\{\!\{ P \}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{ P \}\!\}}\ (\texttt{hoare\_whilealt})$$

Valid

Grading scheme: 3 points for each part

5. (20 points) In this exercise we consider extending Imp with "nondeterministic choice of commands" of the form

```
DO c1 OR c2 OD
```

where `c1` and `c2` are commands. The idea is that evaluating such a command results in nondeterministically running *exactly one* of `c1` or `c2` but not both.

To formalize the extended language, we first add a clause to the definition of commands:

```
Inductive com : Type :=
  ...
  | COr : com -> com -> com.

Notation "'DO' c1 'OR' c2 'OD'" := (COr c1 c2).
```

(a) Refer to the definition of `ceval` (page 15) for the evaluation relation of Imp. What rule(s) must be added to this definition to formalize the behavior of `IF1`? Write out the additional rule(s) in formal Coq notation.

Answer:

```
| E_Or1 : forall (st st' : state) (c1 c2:com),
            c1 / st || st' ->
            (DO c1 OR c2 OD) / st || st'
| E_Or2 : forall (st st' : state) (c1 c2:com),
            c2 / st || st' ->
            (DO c1 OR c2 OD) / st || st'
```

Grading scheme: 6 points. 4 points for using two rules (or equivalent), 1 point for each correct rule.

(b) Can the operational semantics for this version of Imp be implemented using Coq's `Fixpoint` functions with the following signature?

```
Fixpoint ceval (st : state) (c : com) : state := ...
```

Briefly explain why or why not.

Answer: No, the operational semantics doesn't represent a function—a given input command like `DO X ::= 1 OR X ::= 2 OD` does not yield a single output state.

Grading scheme: 2 points

(c) For each purported theorem about Imp with `OR` commands below, write either "provable" if the claim is provable, or give a brief (one sentence) explanation, with a counterexample if possible, of why the claim is not provable. For your reference, the definition of `cequiv`, which remains unchanged from standard Imp, is found on page 15.

i.
```
Theorem thm1 : forall (c:com),
  cequiv   c    (DO c OR c OD).
```
Answer: Provable

ii.
```
Theorem thm2 :
  cequiv  (WHILE BTrue DO SKIP END)     (X ::= 0;;
                                         WHILE X = 0 DO
                                           DO X ::= 0 OR X ::= 1 OD
                                         END).
```

Answer: Not provable because the right-hand command may terminate in a state where `X` is `1` whereas the left-hand command always diverges.

iii.
```
Theorem thm3 :
  cequiv  (X ::= 1)     (X ::= 0;;
                         WHILE X = 0 DO
                           DO X ::= 0 OR X ::= 1 OD
                         END).
```

Answer: Provable

iv.
```
Theorem thm4 : forall (c:com),
  cequiv   c    (DO c OR SKIP OD).
```

Answer: Not provable when `c` is not equivalent to `SKIP`, for example `X ::= 1`.

Grading scheme: 2 points each

(d) Write a Hoare proof rule for the `OR` command. (For reference, the standard Hoare rules for Imp are provided on page 16.)

Try to come up with a rule that is both sound and as precise as possible.

Answer:

$$\frac{\{\!\{ P1 \}\!\}\ \texttt{c1}\ \{\!\{ Q1 \}\!\} \quad \{\!\{ P2 \}\!\}\ \texttt{c2}\ \{\!\{ Q2 \}\!\}}{\{\!\{\ \texttt{fun st => P1 st} \wedge \texttt{P2 st}\ \}\!\}\ \texttt{DO c1 OR c2 OD}\ \{\!\{\ \texttt{fun st => P1 st} \vee \texttt{P2 st}\ \}\!\}} \quad (\texttt{hoare\_or})$$

Grading scheme: 4 points: 3 points for a sound rule, including approximations of the above such as:

$$\frac{\{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\} \quad \{\!\{ P \}\!\}\ \texttt{c2}\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{DO c1 OR c2 OD}\ \{\!\{ Q \}\!\}} \quad (\texttt{hoare\_or})$$

1 point for more precise versions (including disjunction in the postcondition)

6. (10 points)
The following Imp program computes the minimum of `a` and `b`, placing the answer into `Z`.

```
{{ True }}
X ::= a;;
Y ::= b;;
Z ::= 0;;
WHILE (X <> 0 /\ Y <> 0) DO
  X := X - 1;;
  Y := Y - 1;;
  Z := Z + 1
END
{{ Z = min a b }}
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Use informal notations for mathematical formulae and assertions, but please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

The implication steps in your decoration may rely (silently) on the following facts, as well as the usual rules of arithmetic:

- `(X = 0 \/ Y = 0) ->  min X Y = 0`
- `min (X-1) (Y-1) = (min X Y) - 1`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 16 and 17, for reference.

```
  {{ True }} ->>
  {{ 0 + min a b = min a b }}
X ::= a;;
  {{ 0 + min X b = min a b }}
Y ::= b;;
  {{ 0 + min X Y = min a b }}
Z ::= 0;;
  {{ Z + min X Y = min a b }}
WHILE (X <> 0 /\ Y <> 0) DO
    {{ Z + min X Y = min a b /\ (X<>0 /\ Y<>0) }} ->>
    {{ Z+1 + min (X-1) (Y-1) = min a b }}
  X := X - 1;;
    {{ Z+1 + min X (Y-1) = min a b }}
  Y := Y - 1;;
    {{ Z+1 + min X Y = min a b }}
```

```
  Z := Z + 1
    {{ Z + min X Y = min a b }}
END
  {{ Z + min X Y = min a b /\ ~(X<>0 /\ Y<>0) }} ->>
  {{ Z = min a b }}
```

Grading scheme:

- *1 point per implication*
- *3 points for correct "back propagation" of the mechanical parts of the annotation process*
- *4 points for the loop invariant*

7. [**Standard**] (10 points)

Recall that

$$\sum_{\mathtt{X}=1}^{\mathtt{X}=\mathtt{n}} \mathtt{X} \quad = \quad \frac{\mathtt{n} * (\mathtt{n}+1)}{2}$$

The following Imp program calculates the sum above into the variable `Y`.

```
  {{ True }}
X ::= 0;;
Y ::= 0;;
WHILE X <> n DO
  X ::= X + 1;;
  Y ::= Y + X
END
  {{ Y = (n * (n+1))/2 }}
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Use informal notations for mathematical formulae and assertions, but please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

The implication steps in your decoration may rely (silently) on the following fact about natural number division, which follows from the usual rules of arithmetic:

- `(n + 2*m)/2 = n/2 + m`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 16 and 17, for reference.

```
  {{ True }} ->>
  {{ 0 = (0 * (0 + 1))/2 }}
X ::= 0 ;;
  {{ 0 = (X * (X + 1))/2 }}
Y ::= 0 ;;
  {{ Y = (X * (X + 1))/2 }}
WHILE X <> n DO
    {{ Y = (X * (X + 1))/2 /\ X <> n }} ->>
    {{ Y + (X + 1) = ((X + 1) * ((X  + 1) + 1))/2 }}
  X ::= X + 1;;
    {{ Y + X = (X * (X + 1))/2 }} ;;
  Y ::= Y + X
    {{ Y = (X * (X + 1))/2 }}
END
  {{ Y = (X * (X + 1))/2  /\  X = n }} ->>
  {{ Y = (n * (n + 1))/2 }}
```

Grading scheme:

- *1 point per implication*
- *3 points for correct "back propagation" of the mechanical parts of the annotation process*
- *4 points for the loop invariant*

8. [**Advanced**] (12 points)

Let us write `exp m n` for the exponentiation function m^n. The following program computes `exp m n` and stores the answer into `Y`. For $n = 0$, the program is trivial.

For $n > 0$, the program works in two stages. It first uses "repeated squaring" to get close to m^n quickly. That is, the first loop computes `Y` $= m^1$ then `Y` $= m^2$, `Y` $= m^4$, `Y` $= m^8$ and so on, where `Y` $= m^{\texttt{X}}$ and `X` $= 2^k$ for some k. This first stage terminates when `X` $= 2^k \leq n < 2^{k+1}$. The remaining $n - 2^k$ factors of m are completed using the usual "iterative" method of multiplying `Y` by m until the desired power is reached.

```
{{  True }}
IFB n = 0 THEN
  Y ::= 1
ELSE
  Y ::= m;;
  X ::= 1;;
  WHILE n <= (2 * X)  DO
    Y ::= Y * Y;;
    X ::= 2 * X
  END;;
  WHILE X <> n DO
    Y ::= Y * m;;
    X ::= X + 1
  END
FI
{{  Y = exp m n }}
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid.

The implication steps in your decoration may rely (silently) on the following facts about `exp`, which follow from the usual rules of arithmetic:

- `exp m 0 = 1`
- `(exp m n) * (exp m n) = exp m (2 * n)`

Use informal notations for mathematical formulae and assertions, but please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules for decorated programs (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

The Hoare rules and the rules for well-formed decorated programs are provided on pages 16 and 17, for reference.

```
{{  True }}
IFB n = 0 THEN
  {{  n = 0 }} ->>
  {{  1 = exp m n }}
```

```
  Y ::= 1
  {{  Y = exp m n }}
ELSE
  {{  n <> 0 }} ->>
  {{  m = exp m 1 }}
  Y ::= m;;
  {{  Y = exp m 1 }};;
  X ::= 1;;
  {{  Y = exp m X }};;
  WHILE (2 * X) <= n DO
    {{  Y = exp m X /\ (2 * X <= n) }} ->>
    {{  Y * Y = exp m (2 * X) }}
    Y ::= Y * Y;;
    {{  Y = exp m (2 * X) }} ;;
    X ::= 2 * X;;
    {{  Y = exp m X }}
  END;;
  {{  Y = exp m X /\ ~(2 * X <= n) }} ->>
  {{  Y = exp m X }} ;;
  WHILE X <> n DO
    {{  Y = exp m X /\ X <> n }} ->>
    {{  Y * m = exp m (X + 1) }}
    Y ::= Y * m;;
    {{  Y = exp m (X + 1) }} ;;
    X ::= X + 1
    {{  Y = exp m X }}
  END
  {{  Y = exp m X /\  X = n }} ->>
  {{  Y = exp m n }}
FI
{{  Y = exp m n }}
```

Grading scheme:

- *1 point per implication*
- *3 points for correct "back propagation" of the mechanical parts of the annotation process*
- *4 points for the loop invariant*

9. [**Advanced**] (10 points) Give a careful informal proof of the following theorem that states that program equivalence is a congruence for `WHILE` statements.

```
Theorem CWhile_congruence : forall b1 b1' c1 c1',
  bequiv b1 b1' -> cequiv c1 c1' ->
  cequiv (WHILE b1 DO c1 END) (WHILE b1' DO c1' END).
Proof.
```

For your reference, the evaluation relation `ceval` and the definitions of `bequiv` and `cequiv` are given on page 15.

Answer: See the course notes for a solution to this problem.

Grading scheme:

- *2 points: induction on derivation of* `ceval`
- *3 points: proper handling of* `WhileEnd` *case*
- *5 points:* `WhileLoop` *case, broken down as*
  - *1 point: properly stating the assumptions, including existence of intermediate state*
  - *1 point: use of* `bequiv` *and* `cequiv` *for b1' and c1'*
  - *3 points: correct use of the induction hypothesis*
- *"too formal": -1 point*
- *didn't state conclusion: -1 point*
- *no induction / wrong case analysis: -5 points*

Formal definitions for Imp

Syntax

```
Inductive aexp : Type := | ANum : nat -> aexp | AId : id -> 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.

Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
  CSkip.
Notation "l '::=' a" :=
  (CAss l 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).
```

Evaluation relation

```
Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st || st
  | E_Ass  : forall st a1 n X,
      aeval st a1 = n ->
      (X ::= a1) / st || (update st X n)
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  || st' ->
      c2 / st' || st'' ->
      (c1 ; c2) / st || st''
  | E_IfTrue : forall st st' b1 c1 c2,
      beval st b1 = true ->
      c1 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : forall st st' b1 c1 c2,
      beval st b1 = false ->
      c2 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : forall b1 st c1,
      beval st b1 = false ->
      (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true ->
      c1 / st || st' ->
      (WHILE b1 DO c1 END) / st' || st'' ->
      (WHILE b1 DO c1 END) / st || st''

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

Program equivalence

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

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (c1 / st || st') <-> (c2 / st || st').
```

Hoare triples

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st || st' -> P st  -> Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q).
```

Implication on assertions

```
Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

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

(ASCII `->>` is typeset as a hollow arrow in the rules below.)

Hoare logic rules

$$\frac{}{\{\!\!\{\texttt{assn\_sub X a } Q\}\!\!\}\ \texttt{X := a}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_asgn})$$

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

$$\frac{\{\!\!\{P\}\!\!\}\ \texttt{c1}\ \{\!\!\{Q\}\!\!\} \qquad \{\!\!\{Q\}\!\!\}\ \texttt{c2}\ \{\!\!\{R\}\!\!\}}{\{\!\!\{P\}\!\!\}\ \texttt{c1; c2}\ \{\!\!\{R\}\!\!\}} \quad (\texttt{hoare\_seq})$$

$$\frac{\{\!\!\{P \wedge b\}\!\!\}\ \texttt{c1}\ \{\!\!\{Q\}\!\!\} \qquad \{\!\!\{P \wedge \sim b\}\!\!\}\ \texttt{c2}\ \{\!\!\{Q\}\!\!\}}{\{\!\!\{P\}\!\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_if})$$

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

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

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

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

Decorated programs

(a) `SKIP` is locally consistent if its precondition and postcondition are the same:

```
{{ P }}
SKIP
{{ P }}
```

(b) The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions `P` and `R`) if `c1` is locally consistent (with respect to `P` and `Q`) and `c2` is locally consistent (with respect to `Q` and `R`):

```
{{ P }}
c1;
{{ Q }}
c2
{{ R }}
```

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

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

(d) A conditional is locally consistent (with respect to assertions `P` and `Q`) if the assertions at the top of its "then" and "else" branches are exactly `P /\ b` and `P /\ ~b` and if its "then" branch is locally consistent (with respect to `P /\ b` and `Q`) and its "else" branch is locally consistent (with respect to `P /\ ~b` and `Q`):

```
{{ P }}
IFB b THEN
  {{ P /\ b }}
  c1
  {{ Q }}
ELSE
  {{ P /\ ~b }}
  c2
  {{ Q }}
FI
{{ Q }}
```

(e) A while loop with precondition `P` is locally consistent if its postcondition is `P /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

```
{{ P }}
WHILE b DO
  {{ P /\ b }}
  c1
  {{ P }}
END
{{ P /\ ~b }}
```

(f) A pair of assertions separated by `->>` is locally consistent if the first implies the second (in all states):

```
{{ P }} ->>
{{ P' }}
```