Name:____

CSE505, Winter 2012, Midterm Examination
February 7, 2012

Rules:

- The exam is closed-book, closed-notes, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **unevenly** among **5** questions (which have multiple parts).

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. **Skip around.** In particular, make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

For your reference:

$$\begin{array}{rcl} s & ::= & \mathsf{skip} \mid x := e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \\ e & ::= & c \mid x \mid e + e \mid e * e \\ (c & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\ (x & \in & \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{z}_1, \mathtt{z}_2, \ldots, \ldots\}) \end{array}$$

$\boxed{H\ ;\ e\ \Downarrow\ c}$

$$\textsc{const}\ \frac{}{H\ ;\ c\ \Downarrow\ c} \qquad \textsc{var}\ \frac{}{H\ ;\ x\ \Downarrow\ H(x)} \qquad \textsc{add}\ \frac{H\ ;\ e_1\ \Downarrow\ c_1 \qquad H\ ;\ e_2\ \Downarrow\ c_2}{H\ ;\ e_1 + e_2\ \Downarrow\ c_1 + c_2} \qquad \textsc{mult}\ \frac{H\ ;\ e_1\ \Downarrow\ c_1 \qquad H\ ;\ e_2\ \Downarrow\ c_2}{H\ ;\ e_1 * e_2\ \Downarrow\ c_1 * c_2}$$

$\boxed{H_1\ ;\ s_1\ \to\ H_2\ ;\ s_2}$

$$\textsc{assign}\ \frac{H\ ;\ e\ \Downarrow\ c}{H\ ;\ x := e\ \to\ H, x \mapsto c\ ;\ \mathsf{skip}} \qquad \textsc{seq1}\ \frac{}{H\ ;\ \mathsf{skip}; s\ \to\ H\ ;\ s} \qquad \textsc{seq2}\ \frac{H\ ;\ s_1\ \to\ H'\ ;\ s_1'}{H\ ;\ s_1; s_2\ \to\ H'\ ;\ s_1'; s_2}$$

$$\textsc{if1}\ \frac{H\ ;\ e\ \Downarrow\ c \qquad c > 0}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \to\ H\ ;\ s_1} \qquad \textsc{if2}\ \frac{H\ ;\ e\ \Downarrow\ c \qquad c \leq 0}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \to\ H\ ;\ s_2} \qquad \textsc{while}\ \frac{}{H\ ;\ \mathsf{while}\ e\ s\ \to\ H\ ;\ \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ \mathsf{skip}}$$

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \\ \tau & ::= & \mathsf{int} \mid \tau \to \tau \end{array}$$

$\boxed{e \to e'}$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$\boxed{e[e'/x] = e''}$

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{}{c[e/x] = c}$$

$$\frac{e_1[e/x] = e_1' \qquad y \neq x \qquad y \notin FV(e)}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'} \qquad \frac{e_1[e/x] = e_1' \qquad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

- Preservation: If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.
- Progress: If $\cdot \vdash e : \tau$, then e is a value or there exists an e' such that $e \to e'$.
- Substitution: If $\Gamma, x{:}\tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[e'/x] : \tau$.

Name:____

1. (**35** points) This problem adds a single *toggle* to IMP. The *toggle* has two states: up and down. A new *expression* form read evaluates to 1 if the toggle is currently up and 0 if the toggle is currently down. A new *statement* form toggle switches the state of the toggle. The judgment forms for the operational semantics are adapted accordingly.

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{read} \\ s & ::= & \ldots \mid \mathsf{toggle} \\ t & ::= & \mathsf{up} \mid \mathsf{down} \end{array} \qquad \boxed{H\ ;\ t\ ;\ e \Downarrow\ c} \qquad \boxed{H\ ;\ t\ ;\ s \to\ H'\ ;\ t';\ s'}$$

   (a) Give *all* the inference rules for large-step expression evaluation.

   (b) Give *all* the inference rules for small-step statement evaluation.

   (c) If this statement is true, prove it formally, else give a counterexample:
   If $H\ ;\ \mathsf{up}\ ;\ e \Downarrow\ c$, then $H\ ;\ \mathsf{up}\ ;\ e' \Downarrow\ c$ where e' is e with every read replaced by 1.

   (d) If this statement is true, prove it formally, else give a counterexample:
   (Notice the $*$ for 0 or more steps)
   If $H\ ;\ \mathsf{up}\ ;\ s \to^*\ H'\ ;\ \mathsf{up};\ \mathsf{skip}$, then $H\ ;\ \mathsf{up}\ ;\ s' \to^*\ H'\ ;\ \mathsf{up};\ \mathsf{skip}$ where s' is s with every read (in every expression) replaced by 1.

Solution:

(a)

$$\frac{}{H\ ;\ t\ ;\ c \Downarrow\ c} \qquad \frac{}{H\ ;\ t\ ;\ x \Downarrow\ H(x)} \qquad \frac{H\ ;\ t\ ;\ e_1 \Downarrow\ c_1 \qquad H\ ;\ t\ ;\ e_2 \Downarrow\ c_2}{H\ ;\ t\ ;\ e_1 + e_2 \Downarrow\ c_1 + c_2}$$

$$\frac{H\ ;\ t\ ;\ e_1 \Downarrow\ c_1 \qquad H\ ;\ t\ ;\ e_2 \Downarrow\ c_2}{H\ ;\ t\ ;\ e_1 * e_2 \Downarrow\ c_1 * c_2} \qquad \frac{}{H\ ;\ \mathsf{up}\ ;\ \mathsf{read} \Downarrow\ 1} \qquad \frac{}{H\ ;\ \mathsf{down}\ ;\ \mathsf{read} \Downarrow\ 0}$$

(b)

$$\frac{H\ ;\ t\ ;\ e \Downarrow\ c}{H\ ;\ t\ ;\ x := e \to\ H, x \mapsto c\ ;\ t;\ \mathsf{skip}} \qquad \frac{}{H\ ;\ t\ ;\ \mathsf{skip}; s \to\ H\ ;\ t;\ s}$$

$$\frac{H\ ;\ t\ ;\ s_1 \to\ H'\ ;\ t';\ s_1'}{H\ ;\ t\ ;\ s_1; s_2 \to\ H'\ ;\ t';\ s_1'; s_2} \qquad \frac{H\ ;\ t\ ;\ e \Downarrow\ c \qquad c > 0}{H\ ;\ t\ ;\ \mathsf{if}\ e\ s_1\ s_2 \to\ H\ ;\ t;\ s_1}$$

$$\frac{H\ ;\ t\ ;\ e \Downarrow\ c \qquad c \leq 0}{H\ ;\ t\ ;\ \mathsf{if}\ e\ s_1\ s_2 \to\ H\ ;\ t;\ s_2} \qquad \frac{}{H\ ;\ t\ ;\ \mathsf{while}\ e\ s \to\ H\ ;\ t;\ \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ \mathsf{skip}}$$

$$\frac{}{H\ ;\ \mathsf{up}\ ;\ \mathsf{toggle} \to\ H\ ;\ \mathsf{down};\ \mathsf{skip}} \qquad \frac{}{H\ ;\ \mathsf{down}\ ;\ \mathsf{toggle} \to\ H\ ;\ \mathsf{up};\ \mathsf{skip}}$$

(c) see next page

(d) see next page

Name:____

(Extra space for answering problem 1)

Solution:

(c) This statement is true. We prove it by induction on the derivation of $H\ ;\ \mathsf{up}\ ;\ e \Downarrow\ c$, proceeding by cases on the bottommost rule in the derivation:

- If e is a constant, then $e' = e$ so the assumed derivation is the derivation we need.
- If e is a variable, then $e' = e$ so the assumed derivation is the derivation we need.
- If e is $e_1 + e_2$ for some e_1 and e_2, then $H\ ;\ \mathsf{up}\ ;\ e_1 \Downarrow\ c_1$ and $H\ ;\ \mathsf{up}\ ;\ e_2 \Downarrow\ c_2$ where $c = c_1 + c_2$. So by induction $H\ ;\ \mathsf{up}\ ;\ e'_1 \Downarrow\ c_1$ and $H\ ;\ \mathsf{up}\ ;\ e'_2 \Downarrow\ c_2$ where e'_1 and e'_2 are e_1 and e_2 with read replaced by 1. So we can use the rule for addition to derive $H\ ;\ \mathsf{up}\ ;\ e'_1 + e'_2 \Downarrow\ c_1 + c_2$. This is what we need because $e'_1 + e'_2$ is e with read replaced by 1 and $c = c_1 + c_2$.
- If e is $e_1 * e_2$ for some e_1 and e_2, then $H\ ;\ \mathsf{up}\ ;\ e_1 \Downarrow\ c_1$ and $H\ ;\ \mathsf{up}\ ;\ e_2 \Downarrow\ c_2$ where $c = c_1 * c_2$. So by induction $H\ ;\ \mathsf{up}\ ;\ e'_1 \Downarrow\ c_1$ and $H\ ;\ \mathsf{up}\ ;\ e'_2 \Downarrow\ c_2$ where e'_1 and e'_2 are e_1 and e_2 with read replaced by 1. So we can use the rule for multiplication to derive $H\ ;\ \mathsf{up}\ ;\ e'_1 * e'_2 \Downarrow\ c_1 * c_2$. This is what we need because $e'_1 * e'_2$ is e with read replaced by 1 and $c = c_1 * c_2$.
- If e is read and the toggle is up, then c is 1 and e' is 1 and we can use the rule for constants to derive $H\ ;\ \mathsf{up}\ ;\ 1 \Downarrow\ 1$.
- The rule where e is read and the toggle is down cannot end the derivation of $H\ ;\ \mathsf{up}\ ;\ e \Downarrow\ c$, so this case holds vacuously.

(d) This statement is false. There are an infinite number of countexamples, such as:
$.\ ;\ \mathsf{up}\ ;\ \mathsf{toggle}; (x := \mathsf{read}; \mathsf{toggle}) \rightarrow^*\ ., x \mapsto 0\ ;\ \mathsf{up};\ \mathsf{skip}$,
but
$.\ ;\ \mathsf{up}\ ;\ \mathsf{toggle}; (x := 1; \mathsf{toggle}) \rightarrow^*\ ., x \mapsto 1\ ;\ \mathsf{up};\ \mathsf{skip}$

Name:______________________________________

2. (**31** points) This problem uses Caml and continues using IMP-with-toggle from problem 1. You are given the type definitions for IMP-with-toggle and the "mysterious" function `foo`:

```
type exp = Int of int | Var of string | Plus of exp * exp | Times of exp * exp | Read
type stmt = Skip | Assign of string * exp | Seq of stmt * stmt
          | If of exp * stmt * stmt | While of exp * stmt | Toggle

let foo lst =
  let rec f lst s =
    match lst with
      [] -> true
    | hd::tl -> hd <> s && f tl s  in (* <> is "not equal" *)
  let rec g i =
    let t = "_t" ^ (string_of_int i) in (* ^ concatenates strings *)
    if f lst t then t else g (i+1) in
  g 0
```

(a) Document `foo`: What does it take and what does it return (in terms of types and values)? Do *not* describe *how* `foo` is implemented.

(b) Write a Caml function `allVars` of type `stmt -> string list` that returns all the variables appearing anywhere in the statement. Hints:

- Duplicate strings are fine; do *not* bother removing them.
- Sample solution is approximately 15 lines total.
- You will need a helper function.
- Caml's append operator `@` is very useful.

(c) IMP-with-toggle is kind of stupid because we can *encode* the concept in regular IMP. Describe in 1–3 English sentences how you could *translate* IMP-with-toggle to regular IMP.

(d) Implement the translation you described in part (c) with a Caml function `translate` of type `stmt -> stmt`. Hints:

- The result should not use `Toggle` or `Read`.
- The sample solution is approximately 20 lines total.
- You will need a helper function.
- There's a reason parts (a) and (b) are part of this problem.

Solution:
(next page)

Name:____

(Extra space for answering problem 2)

Solution:

(a) `foo` has type `string list -> string`. It returns the string `_t`i where i is (the string representation of) the smallest natural number such that `_t`i is not in the argument list.

(b)
```
let rec allVarsE e =
    match e with
      Int _ -> []
    | Var s -> [s]
    | Plus(e1,e2) -> (allVarsE e1) @ (allVarsE e2)
    | Times(e1,e2) -> (allVarsE e1) @ (allVarsE e2)
    | Read -> []
let rec allVars s =
   match s with
     Skip -> []
   | Toggle -> []
   | Assign(x,e) -> x :: (allVarsE e)
   | If(e,s1,s2) -> (allVarsE e) @ (allVars s1) @ (allVars s2)
   | While(e,s1) -> (allVarsE e) @ (allVars s1)
   | Seq(s1,s2) -> (allVars s1) @ (allVars s2)
```

(c) We can use a new IMP variable x not otherwise used in the program to hold the current toggle (1 for up, 0 for down). Then we can replace read with reading x and toggle with $x := 1 + (-1 * x)$ (or if x $x := 0$ $x := 1$).

(d)
```
let translate s =
  let v = foo (allVars s) in
  let rec xe e =
    match e with
      Int _ -> e
    | Var _ -> e
    | Plus(e1,e2) -> Plus(xe e1, xe e2)
    | Times(e1,e2) -> Times(xe e1, xe e2)
    | Read -> Var v in
  let rec xs s =
    match s with
      Skip -> Skip
    | Toggle -> Assign(v,Plus(Int 1,Times(Int (-1),Var v)))
          (* alternately If(Var(v),Assign(v,Int 0),Assign(v,Int 1)) *)
    | Assign(x,e) -> Assign(x, xe e)
    | If(e,s1,s2) -> If(xe e, xs s1, xs s2)
    | Seq(s1,s2) -> Seq(xs s1, xs s2)
    | While(e,s) -> While(xe e, xs s)
  in xs s
```

Name:____

3. (**13** points) This problem uses the untyped lambda-calculus and full reduction. Recall this encoding of pairs:

   - "mkpair" $\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$
   - "fst" $\lambda p.\ p\ \lambda x.\ \lambda y.\ x$
   - "snd" $\lambda p.\ p\ \lambda x.\ \lambda y.\ y$

   We would expect a correct encoding to show "fst" ("mkpair" $z\ z$) evaluates to z. But this sequence of steps allegedly shows that "fst" ("mkpair" $z\ z$) evaluates to "fst":

$$
\begin{array}{ll}
 & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)((\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y)\ z\ z) \\
\to & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)((\lambda y.\ \lambda z.\ z\ z\ y)\ z) \\
\to & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)(\lambda z.\ z\ z\ z)) \\
\to & (\lambda z.\ z\ z\ z)\ \lambda x.\ \lambda y.\ x \\
\to & (\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ x) \\
\to & (\lambda y.\ (\lambda x.\ \lambda y.\ x))\ (\lambda x.\ \lambda y.\ x) \\
\to & \lambda x.\ \lambda y.\ x
\end{array}
$$

   (a) The sequence of steps is wrong. Which steps are wrong and why are they wrong?

   (b) Show a correct sequence of steps that produces z but is otherwise very similar to the sequence of steps shown above.

   Solution:

   (a) The first two steps both capture z. We should α-convert $\lambda z.\ z\ x\ y$ in order to perform these first two steps properly.

   (b)

$$
\begin{array}{ll}
 & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)((\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y)\ z\ z) \\
\to & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)((\lambda y.\ \lambda q.\ q\ z\ y)\ z) \\
\to & (\lambda p.\ p\ \lambda x.\ \lambda y.\ x)(\lambda q.\ q\ z\ z)) \\
\to & (\lambda q.\ q\ z\ z)\ \lambda x.\ \lambda y.\ x \\
\to & (\lambda x.\ \lambda y.\ x)\ z\ z \\
\to & (\lambda y.\ z)\ z \\
\to & z
\end{array}
$$

Name:____

4. (**10** points) In this problem, assume the simply-typed lambda calculus. For each of the following:

- If the answer is *yes*, give an example Γ and τ.
- If the answer is *no*, you can just say "no."

(a) Is there a Γ and τ such that $\Gamma \vdash (\lambda x.\ x)\ x : \tau$?

(b) Is there a Γ and τ such that $\Gamma \vdash \lambda x.\ (x\ x) : \tau$?

(c) Is there a Γ and τ such that $\Gamma \vdash x\ x : \tau$?

(d) Is there a Γ and τ such that $\Gamma \vdash x\ (\lambda x.\ x) : \tau$?

Solution:

(a) Yes, for example $\Gamma = \cdot, x{:}\mathsf{int}$ and $\tau = \mathsf{int}$. In general, the type of x in Γ has to be τ.

(b) No

(c) No

(d) Yes, for example $\Gamma = \cdot, x{:}(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ and $\tau = \mathsf{int}$. In general, the type of x in Γ has to have the form $(\tau' \to \tau') \to \tau$.

Name:____

5. (**11** points) Consider this lemma, which is slightly different from the Preservation Lemma we proved for the simply-typed lambda caculus:

   Differently Preserved: If $\cdot \vdash e : \tau$ and $e \to e'$, then there exists a τ' such that $\cdot \vdash e' : \tau'$.

   (a) Is the Differently Preserved Lemma *weaker*, *stronger*, or *incomparable* to the Preservation Lemma? Explain.

   (b) Is the Differently Preserved Lemma true? Explain.

   (c) Is the Differently Preserved Lemma (instead of the Preservation Lemma) and the Progress Lemma sufficient to prove Type Safety? Explain.

   (d) Explain why we proved the Preservation Lemma instead of just the Differently Preserved Lemma.

   Solution:

   (a) It is weaker: The Preservation Lemma implies the Differently Preserved Lemma just by choosing τ' to be τ. (Something is weaker than something else that implies it.)

   (b) Yes, the Preservation Lemma is true and it implies the Differently Preserved Lemma.

   (c) Yes, just like the Preservation Lemma, the Differently Preserved Lemma and induction on the number of steps taken ensure that no well-typed program can become ill-typed. And Progress ensures no well-typed program is stuck.

   (d) Because the Differently Preserved Lemma has too weak an induction hypothesis for the proof to go through. For example, when $e_1\ e_2 \to e_1'\ e_2$ because $e_1\ e_1'$, it's not enough to know that e_1' has *some type*. We need to know it has the *same type* as e_1 to show that $e_1'\ e_2$ still type-checks.