Name:____

CSE 505, Fall 2009, Final Examination
14 December 2009

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 12:20.**
- You can rip apart the pages.
- There are **100 points** total, distributed **unevenly** among **7** questions.
- The questions 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 (page 1 of 2):

$$
\begin{array}{rcl}
e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \mid \mathsf{fix}\ e \\
v & ::= & \lambda x.\ e \mid c \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{array}
$$

$\boxed{e \to e' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2}$

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

$$
\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}
$$

$$
\frac{e_i \to e_i'}{\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \to \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i', \ldots, l_n = e_n\}}
$$

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

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n \qquad \text{labels distinct}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad 1 \leq i \leq n}{\Gamma \vdash e.l_i : \tau_i}
$$

$$
\frac{\Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'}
$$

$$
\frac{}{\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n, l{:}\tau\} \leq \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}
$$

$$
\frac{}{\{l_1{:}\tau_1, \ldots, l_{i-1}{:}\tau_{i-1}, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, l_{i-1}{:}\tau_{i-1}, \ldots, l_n{:}\tau_n\}}
$$

$$
\frac{\tau_i \leq \tau_i'}{\{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i', \ldots, l_n{:}\tau_n\}}
$$

$$
\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4} \qquad \frac{}{\tau \leq \tau} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}
$$

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e[\tau] \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\tau \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e
\end{array}
\qquad
\begin{array}{rcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

$\boxed{e \to e' \text{ and } \Delta; \Gamma \vdash e : \tau}$

$$
\frac{e \to e'}{e\ e_2 \to e'\ e_2} \qquad \frac{e \to e'}{v\ e \to v\ e'} \qquad \frac{e \to e'}{e[\tau] \to e'[\tau]} \qquad \frac{}{(\lambda x{:}\tau.\ e)\ v \to e[v/x]} \qquad \frac{}{(\Lambda\alpha.\ e)[\tau] \to e[\tau/\alpha]}
$$

$$
\frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau_1}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau_1}
$$

$$
\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_1} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}
$$

Name:____

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e \mid \mathsf{B}x.\ e) \mid \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e \mid (e,e) \mid e.1 \mid e.2 \\
\tau & ::= & \ldots \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \tau_1 * \tau_2 \\
v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \mid \mathsf{roll}_\tau\ v \mid (v,v)
\end{array}
$$

$\boxed{e \to e' \text{ and } \Delta;\Gamma \vdash e : \tau}$

$$\frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')} \qquad \frac{e \to e'}{\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to \mathsf{match}\ e'\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2}$$

$$\frac{}{\mathsf{match}\ \mathsf{A}(v)\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_1[v/x]} \qquad \frac{}{\mathsf{match}\ \mathsf{B}(v)\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_2[v/y]}$$

$$\frac{e \to e'}{\mathsf{roll}_{\mu\alpha.\tau}\ e \to \mathsf{roll}_{\mu\alpha\tau}\ e'} \qquad \frac{e \to e'}{\mathsf{unroll}\ e \to \mathsf{unroll}\ e'} \qquad \frac{}{\mathsf{unroll}\ (\mathsf{roll}_{\mu\alpha.\tau}\ v) \to v}$$

$$\frac{e_1 \to e_1'}{(e_1,e_2) \to (e_1',e_2)} \qquad \frac{e_2 \to e_2'}{(v,e_2) \to (v,e_2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2} \qquad \frac{}{(v_1,v_2).1 \to v_1} \qquad \frac{}{(v_1,v_2).2 \to v_2}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Delta;\Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau[(\mu\alpha.\tau)/\alpha]}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \qquad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash (e_1,e_2) : \tau_1 * \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 * \tau_2}{\Delta;\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 * \tau_2}{\Delta;\Gamma \vdash e.2 : \tau_2}$$

Module Thread:

```
type t
val create : ('a -> 'b) -> 'a -> t
val join : t -> unit
```

Module Mutex:

```
type t
val create : unit -> t
val lock : t -> unit
val unlock : t -> unit
```

Module Event:

```
type 'a channel
type 'a event
val new_channel : unit -> 'a channel
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val choose : 'a event list -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
val sync : 'a event -> 'a
```

Name:____

1. (**15** points) Consider a typed-lambda calculus with functions, integers, records, and *subtyping* as considered in class. Note this problem considers only the *subtyping judgment* with the *six* inference rules on page 2 of this exam, not the typing judgment. For each of the following claims, if it is true, prove it. If it is false, provide a counterexample.

   (a) If $\tau_1 \leq \tau_2$ and τ_1 *is* a record type, then τ_2 *is* a record type.

   (b) If $\tau_1 \leq \tau_2$ and τ_1 *contains* a function type somewhere in it, then τ_2 *contains* a function type somewhere in it.

   Solution:

   (a) True. Proof by induction on the derivation of $\tau_1 \leq \tau_2$ proceeding by cases on the bottom-most rule instantiated in the derivation:

   - If width subtyping, then τ_2 is a record type.
   - If permutation subtyping, then τ_2 is a record type.
   - If depth subtyping on records, then τ_2 is a record type.
   - If function subtyping, then τ_1 is not a record type so the claim holds vacuously.
   - If reflexivity, then $\tau_1 = \tau_2$, so τ_1 being a record type implies τ_2 is a record type.
   - If transitivity, then by inversion there is some τ such that $\tau_1 \leq \tau$ and $\tau \leq \tau_2$. Assume τ_1 is a record type. Then by induction and $\tau_1 \leq \tau$, τ is a record type. Then by induction and $\tau \leq \tau_2$, τ_2 is a record type.

   (b) False. One example: Let $\tau_1 = \{l_1 : \tau_2 \rightarrow \tau_3\}$ and $\tau_2 = \{\}$. Width subtyping suffices.

Name:____

2. (**10** points) Consider System F.

   (a) Write a well-typed term that implements function composition and is as polymorphic as possible. Function composition takes two (curried) arguments (say f and g) and returns a function that given x returns $f(g(x))$.

   (b) Give the type for the term you wrote in part (a).

   Be sure to use parentheses appropriately.

   Solution:

   (a) $\Lambda\alpha_1.\ \Lambda\alpha_2.\ \Lambda\alpha_3.\ \lambda f : \alpha_2 \to \alpha_3.\ \lambda g : \alpha_1 \to \alpha_2.\ \lambda x : \alpha_1.\ f\ (g\ x)$

   (b) $\forall\alpha_1.\forall\alpha_2.\forall\alpha_3.(\alpha_2 \to \alpha_3) \to (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_3$

Name:____

3. (**10** points) For each of the following Caml definitions, does it type-check in Caml? If so, what type does it have? If not, why not?

   (a) `let part_a = (fun g -> (fun x y -> x) (g 0) (g 17))`

   (b) `let part_b = (fun g -> (fun x y -> x) (g 0) (g true))`

   (c) `let part_c = (fun g -> (fun x y -> x) (g 0) (g (g 17)))`

   Solution:

   (a) Type-checks: `(int -> 'a) -> 'a`

   (b) Does not type-check: The type-inferencer will conclude that `g` must be a function takes an `int` and a function that takes a `bool`, and these cannot both hold.

   (c) Type-checks: `(int -> int) -> int`

Name:____

4. (**15** points) Consider a typed λ-calculus with recursive types, sums, pairs, int, string, and unit. Assume the language uses explicit roll and unroll coercions (not subtyping) for recursive types.

   (a) Give a recursive type for binary trees where interior nodes have no data and each leaf has either a `string` or an `int`.

   (b) In this part, you can use tr as an abbreviation for the type you gave in part (a). Using fix for recursion, write a program of type $\mathsf{tr} \rightarrow (\mathsf{int} + \mathsf{unit})$. When called with a tree, the program should return $\mathsf{A}(i)$ if i is the left-most integer in the tree and $\mathsf{B}(())$ if the tree has no integers. Give explicit types to all function arguments. If you get confused by fix, use let rec instead for significant partial credit.

Solution:
Answer to (b) depends on answer to (a).

(a) One possible answer: $\mu\alpha.((\mathsf{string} + \mathsf{int}) + (\alpha * \alpha))$.

(b)
```
fix \f : tr -> (int + unit)) .
    \x : tr .
        match unroll x with
           A y. -> (match y with
                     A z. -> B (())
                   | B z. -> A (z))
         | B y. -> (match f y.1 with
                     A z. -> A (z)
                   | B z. -> f y.2)
```

Name:____

5. (**20** points) In this problem you will use CML to implement a server for "rock-paper-scissors". Rock-paper-scissors is a game where normally there are two players who each pick "rock", "paper", or "scissors" and either one player wins or there is a tie. This Caml code defines the rules for this two-player game:

```
type play = Rock | Scissors | Paper

type winner = Left | Right | Tie

let pick_winner p1 p2 = (* useful helper function *)
  match (p1,p2) with
    (Rock,Rock) -> Tie
  | (Scissors,Scissors) -> Tie
  | (Paper,Paper) -> Tie
  | (Rock,Scissors) -> Left
  | (Rock,Paper) -> Right
  | (Scissors,Paper) -> Left
  | (Scissors,Rock) -> Right
  | (Paper,Rock) -> Left
  | (Paper,Scissors) -> Right
```

You will implement the `new_game` function in this interface:

```
type game
type play = Rock | Scissors | Paper
type result = Win | Lose
val new_game : unit -> game
val play_game : game -> play -> result
```

`new_game` creates a new server. Players can play by calling `play_game`. There are two differences from the two-player version:

- Players do not know their opponent. The server can choose any opponent. A player simply learns whether he/she won or lost.
- There are no ties. The server must match up players so that each player wins or loses. For example, the server can pair a "rock" with a "scissors" or a "paper" but not with another "rock."

To avoid ties, the server may need to make players wait for other players to arrive. However, to avoid any unnecessary waiting, all the zero-or-more waiting players at any one time will have picked the same play — otherwise the server should have paired up two players that picked differently. So, when a new player arrives, if there are no waiters or the waiters "tie" with the new player, then the new player waits. Otherwise, the new player and one waiter complete.

Do not change the partial implementation below; just complete `new_game`. You do not need `choose` and `wrap`. The sample solution is 15–20 lines. Use `pick_winner`, defined above. Remember that when players are paired up the winner and the loser need to be informed.

```
type result = Win | Lose
type game = (play * result channel) channel

let new_game () = (* for you *)

let play_game g p =
  let c = new_channel () in
  sync (send g (p,c));
  sync (receive c)
```

Name:____

This page intentionally blank.

Solution:

```
let new_game () =
  let c = new_channel() in
  let send_results winner loser =
    sync(send winner Win);
    sync(send loser Lose) in
  let rec loop wait_kind lst =
    let play, response_ch = sync (receive c) in
    match lst with
      [] -> loop play [response_ch]
    | hd::tl ->
        match pick_winner play wait_kind with
          Tie   -> loop wait_kind (response_ch::lst)
        | Left  -> send_results response_ch hd; loop wait_kind tl
        | Right -> send_results hd response_ch; loop wait_kind tl in
  ignore(Thread.create (loop Rock) []);
  c
```

Name:____

6. (**15** points) Consider a class-based OOP language like we did in class where method-name reuse means overriding. Consider this code skeleton:

```
class A           { A m() { return self;    } ... }
class B extends A { B m() { return super(); } ... }
class C extends B { A m() { return new A(); } ... }

class Main { void main() { ... } }
```

   (a) Assuming all code not shown (i.e., the code in the `...`) type-checks, there are *two* reasons the code above does *not* type-check. What are they?

   (b) One of your two answsers to part (a) *cannot* actually lead to a program getting stuck. Explain why not.

   (c) The other of your two answers to part (a) *can* lead to a program getting stuck. Fill in the `...` as necessary (you may not need to use all of them) such that all the code you add type-checks but running the `main` method would produce a "method not found" error.

Solution:

   (a) First, the return type of B's m method is B, but the type of the super call is A, which is not a subtype of B. Second, C's m method has return type A, but it is overriding a method with return type B, and again A is not a subtype of B.

   (b) The first one (B's m method) cannot cause a problem. Although the static type of `super()` is `A`, in fact this method returns `self`, which will be a B for any instance of B.

   (c) Add to class B a method `void n(){}`. Then make the body of `main` be something like

```
B c = new C();
c.m().n();
```

   The first line type-checks using subsumption. The second line type-checks because B's m returns a B and instance of B have an n method. But at run-time, `c.m()` returns an A, which does not have an n method.

Name:____

7. (**15** points) Consider a single-inheritance class-based OOP language. Assume booleans are provided as primitives.

   Assume method-name reuse means either static overloading or multimethods. For each part give a single answer that is correct under either assumption. This is not intended to make the problem harder.

   (a) Write a program (class definitions and client code) such that:
   - The program type-checks.
   - The program evaluates to true.
   - There is one method definition you can remove from the program (comment-out) such that the program still type-checks but the program now evaluates to false.

   Clearly indicate which method should be commented out.

   (b) Write a program (class definitions and client code) such that:
   - The program type-checks.
   - The program evaluates to true.
   - There is one method definition you can remove from the program (comment-out) such that the program would now have a "no best match" error.

   Clearly indicate which method should be commented out.

   Solution:

   (a)
```
class A {}
class B extends A {}
class Main{
  bool m(A a) { return false; }
  bool m(B b) { return true; } // This one!
  bool main() {
     return self.m(new B());
  }
}
```

   (b)
```
class A {}
class B extends A{}
class Main{
  bool m(A a, B b) { return true; }
  bool m(B b, A a) { return true; }
  bool m(B b, B B) { return true; } // This one!
  bool main() {
    return self.m(new B(), new B());
  }
}
```