Name:____

CSE 505, Fall 2006, Final Examination
11 December 2006

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, but please write your name on each page.
- There are **95 points** total, distributed among **5** questions (most of 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 (page 1 of 2):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Name:____

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

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

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

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

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

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

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 λ-calculus with recursive types where we use explicit expressions of the form $\mathsf{roll}_\tau\ e$ and $\mathsf{unroll}\ e$ (as opposed to subtyping). For each of the following typing rules, explain why it makes little if any sense to add the rule to our type system.

   (a)

$$\frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau}$$

   (b) Let $FTV(\tau)$ mean the free type variables in τ. Assume it has been defined correctly.

$$\frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau \qquad \alpha \notin FTV(\tau)}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau}$$

Solution:

(a) This rule is unsound. The result type could have free type variables α which could then be captured by some outer binding, such as type-abstraction. It is enough to say that the type τ may make no sense without α being properly bound. For example, if in some context f has type $\alpha \to \beta$ and x has type $\mu\alpha.\alpha$, then $f\ \mathsf{unroll}\ x$ would type-check, but x does not have the type f expects.

(b) This rule is trivially admissable. The existing typing rule for $\mathsf{unroll}\ e$ already gives the result type τ when $\alpha \notin FTV(\tau)$ by the definition of type substitution.

Name:______________________________________

2. (**20** points) Consider a typed λ-calculus with a more flexible version of sum types than we considered in class:

   - There are an infinite number of constructors, not just A and B. Let C range over constructors. So an example expression is $C_7\ (\lambda x.\ x)$.
   - A single sum type $+\{C_1{:}\tau_1, \ldots, C_n{:}\tau_n\}$ can list any finite number of constructors and the types of the values they wrap. So one example type would be $+\{C_3{:}\mathsf{int}, C_7{:}\mathsf{int} \rightarrow \mathsf{int}, C_2{:}\mathsf{unit}\}$. Like in Caml, the order of constructors in a type is not signficiant. Unlike in Caml, we are using structural typing and different types can use the same constructors (with possibly different types they wrap).
   - As you should expect, a match expression can have any finite number of branches, with a different constructor for each branch. Informally (it can be formalized), a match expression has type τ if (1) the matched expression has type $+\{C_1{:}\tau_1, \ldots, C_n{:}\tau_n\}$, (2) for each C_i in the type there is a branch of the form $C_i\ x_i \rightarrow e_i$ where e_i has type τ assuming x_i has type τ_i.
   - The typing rule for constructor expressions can just be:

     $$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash C\ e : +\{C\ \tau\}}$$

     If that seems odd, read on.

   Come up with **three** sound and generally useful **subtyping rules** for these sum types and **justify informally** why each rule is sound. Write the rules formally.

   Hint: We have three sound and generally useful subtyping rules for record types. Some of your rules might be almost identical to those and others might be analogous but crucially different.

   Note: We already have rules like reflexivity and transitivity. Your rules should specifically deal with the new sum types.

   Solution:

   - Permutation:

     $$\frac{}{+\{C_1\ \tau_1, \ldots, C_i\ \tau_i, C_j\ \tau_j, \ldots, C_n\ \tau_n\} \leq +\{C_1\ \tau_1, \ldots, C_j\ \tau_j, C_i\ \tau_i, \ldots, C_n\ \tau_n\}}$$

     The order of constructors does not matter because the type-checking of match expressions such requires that there is a branch for each constructor.
   - Depth:

     $$\frac{\tau \leq \tau'}{+\{C_1\ \tau_1, \ldots, C\ \tau, \ldots, C_n\ \tau_n\} \leq +\{C_1\ \tau_1, \ldots, C\ \tau', \ldots, C_n\ \tau_n\}}$$

     Depth subtyping is sound because values built from constructors are immutable. If a value is actually a C wrapping a τ and we read the wrapped value and treat it as a τ', we can do only less with it.
   - Anti-width:

     $$\frac{}{+\{C_1\ \tau_1, \ldots, C_n\ \tau_n\} \leq +\{C_1\ \tau_1, \ldots, C_n\ \tau_n, C_{n+1}\ \tau_{n+1}\}}$$

     If a value is built from one of n constructors, then we can add a constructor and conclude that value is still definitely built from one of the $n+1$ constructors. Someone matching against the supertype just has to provide a branch that will never be taken if the matched-value actually has the subtype.

Name:____

3. (**20** points)

   (a) Consider this System F function. Note the comma in it, which creates a pair. (We also assume System F has pairs and strings.)

   $$\lambda x : \mathsf{int}.\ \lambda y : \mathsf{string}.\ \lambda z : \forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha).\ ((z\ [\mathsf{string}]\ [\mathsf{int}]\ y\ x),\ (z\ [\mathsf{int}]\ [\mathsf{string}]\ x\ y))$$

   i. What does this function do? Be as precise as possible.

   ii. Why is it not possible to write a function equivalent to this one in ML?

   (b) Consider this Caml function.

   ```
   let rec f x y = if x < y then (x,y) else f y x
   ```

   i. What does this function do? Be as precise as possible.

   ii. Why is it not possible to write a function equivalent to this one in System F?

Solution:

(a) This function takes three arguments x, y, and z and always returns the pair (y, x). (Note that z must be a function that always terminates and always returns its first argument.)

This function cannot be implemented in ML for two reasons. First, it cannot type-check because ML has only prenex quantification and z is used at two different types. Second, there is no way to require the caller to pass in a function that takes two arguments and always returns the first one.

(b) This function take two integers x and y. It returns (x, y) if x is less than y, (y, x) if y is less than x, and diverges if $x = y$. In System F, no function can ever diverge.

Name:____

4. (**20** points)

Consider this interface and partial Concurrent ML implementation:

```
(* Interface *)
type gt_or_tg (* "give then take or take then give" (forever) *)
val new_gt_or_tg : unit -> gt_or_tg
val give : gt_or_tg -> int -> unit
val take : gt_or_tg -> int

(* Implementation *)
open Event
type gt_or_tg = int channel
let give ch i = sync (send ch i)
let take ch   = sync (receive ch)
let new_gt_or_tg () = (* for you *)
```

Implement `new_gt_or_tg` so that this library behaves as follows:

- Each `gt_or_tg` can handle gives and takes. The integer passed to `give` is ignored and the integer returned from `take` is arbitrary (0 is fine). (This is silly, but fine on an exam.)
- For a given `gt_or_tg` consider the calls to `give` or `take` using *that* `gt_or_tg`. The first such call to *return* (i.e., finish evaluation) can be a give or a take. But if the first call to return is a give then the second call to return must be a take, and if the first call to return is a take, then the second call to return must be a give. Similarly, the third, fifth, seventh, etc. call to return can be a give or a take, but the next call to return must be a call to the other function.
- There is no additional guarantee that calls return in any particular order. However, if there are the same number of calls to give and take, then all should return. (The natural solution would do this; this is a technicality so you can't claim that a solution in which no call ever returns is correct.)

Hints:

- Use `choose` and `wrap`.
- Remember to put `sync` in all the right places.
- Sample solution is 10–11 lines.

Solution:

```
let new_gt_or_tg () =
  let ans = new_channel() in
  let rec loop () =
    sync (choose [
          (wrap (send ans 0) (fun () -> ignore(sync (receive ans))));
          (wrap (receive ans) (fun i -> sync (send ans 0)))
        ] );
     loop ()
  in
  Thread.create loop ();
  ans
```

Name:____

5. (**20** points) Consider this code in a class-based OOP language with multiple inheritance. A subclass overrides a method by defining a method with the same name and arguments.

```
class A              { }
class B extends A    { unit m1() { print "m1B" } }
class C extends B    { unit m1() { print "m1C" } }
class D extends A    { }
class E extends C, D { }
class Main {
  unit m2(D c) { print "m2D"; }
  unit m2(C c) { print "m2C"; c.m1() }
  unit m2(B b) { print "m2B"; b.m1() }
  unit main() {
     E e = new E();
     e.m1();        // 0
     ((B)e).m1();   // 1
     self.m2(e);    // 2
     self.m2((D)e); // 3
     self.m2((C)e); // 4
     self.m2((B)e); // 5
  }
}
```

(a) Assume the language has static overloading. For each of the lines 0–5, determine if the method call is ambiguous ("no best match") or not. If it is not, what does executing the call print?

(b) Assume the language has multimethods. For each of the lines 0–5, determine if the method call is ambiguous ("no best match") or not. If it is not, what does executing the call print?

Solution:

(a)
0 m1C
1 m1C
2 ambiguous
3 m2D
4 m2C m1C
5 m2B m1C

(b)
0 m1C
1 m1C
2 ambiguous
3 ambiguous
4 ambiguous
5 ambiguous