Name:____

CSE P505, Spring 2006, Final Examination
6 June 2006

Rules:

- **Please do not turn the page until everyone is ready.**
- The exam is closed-book, closed-note, except for two sides of one 8.5x11in piece of paper.
- **Please stop promptly at 8:30.**
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **very unevenly** among 7 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 and focus on the questions worth more points.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

For your reference (page 1 of 2):

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

$\boxed{H\ ;\ e\ \Downarrow\ i \text{ and } H\ ;\ s\ \Downarrow\ H'}$

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

$$
\textsc{skip}\ \frac{}{H\ ;\ \mathsf{skip}\ \Downarrow\ H}
\qquad
\textsc{assign}\ \frac{H\ ;\ e\ \Downarrow\ i}{H\ ;\ x := e\ \Downarrow\ H, x \mapsto i}
\qquad
\textsc{seq}\ \frac{H\ ;\ s_1\ \Downarrow\ H'' \qquad H''\ ;\ s_2\ \Downarrow\ H'}{H\ ;\ s_1; s_2\ \Downarrow\ H'}
$$

$$
\textsc{if}1\ \frac{H\ ;\ e\ \Downarrow\ i \quad i \neq 0 \quad H\ ;\ s_1\ \Downarrow\ H'}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \Downarrow\ H'}
\qquad
\textsc{if}2\ \frac{H\ ;\ e\ \Downarrow\ 0 \quad H\ ;\ s_2\ \Downarrow\ H'}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \Downarrow\ H'}
\qquad
\textsc{while}\ \frac{H\ ;\ \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ \mathsf{skip}\ \Downarrow\ H'}{H\ ;\ \mathsf{while}\ e\ s\ \Downarrow\ H'}
$$

$$
\begin{array}{rcl}
e & ::= & \lambda x.\ e \mid x \mid e\ e \\
v & ::= & \lambda x.\ e
\end{array}
$$

$\boxed{e\ \Downarrow\ v \text{ and substitution}}$

$$
\frac{}{\lambda x.\ e\ \Downarrow\ \lambda x.\ e}
\qquad
\frac{e_1\ \Downarrow\ \lambda x.\ e_3 \qquad e_2\ \Downarrow\ v_2 \qquad e_3\{v_2/x\}\ \Downarrow\ v}{e_1\ e_2\ \Downarrow\ v}
\qquad
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\lambda x.\ e) & = & FV(e) - \{x\}
\end{array}
$$

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

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

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

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

$$
\frac{}{\Gamma \vdash x : \Gamma(x)}
\qquad
\frac{}{\Gamma \vdash c : \mathsf{int}}
$$

$$
\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2}
\qquad
\frac{\Gamma, \alpha \vdash e : \tau_1}{\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau_1}
\qquad
\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}
\qquad
\frac{\Gamma \vdash e : \forall\alpha.\tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash e[\tau_2] : \tau_1\{\tau_2/\alpha\}}
$$

Name:______________________________________

$$
\begin{array}{rcl}
e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \mid (e,e) \mid e.1 \mid e.2 \mid \mathsf{A}\ e \mid \mathsf{B}\ e \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}\ x \to e | \mathsf{B}\ x \to e \mid \mathsf{letrec}\ f\ x.\ e \\
 & \mid & \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \\
v & ::= & \lambda x.\ e \mid c \mid (v,v) \mid \mathsf{A}\ v \mid \mathsf{B}\ v \mid \{l_1 = v, \ldots, l_n = v\} \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau \mid \{l_1 = \tau, \ldots, l_n = \tau\}
\end{array}
$$

$\boxed{e \to e'}$

$$
\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{}{(\lambda x.\ e)\ v \to e\{v/x\}} \qquad \frac{}{(\mathsf{letrec}\ f\ x.\ e)\ v \to e\{v/x\}\{(\mathsf{letrec}\ f\ x.\ e)/f\}}
$$

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

$$
\frac{e \to e'}{\mathsf{A}\ e \to \mathsf{A}\ e'} \qquad \frac{e \to e'}{\mathsf{B}\ e \to \mathsf{B}\ e'} \qquad \frac{e_1 \to e_1'}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3 \to \mathsf{match}\ e_1'\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3}
$$

$$
\frac{}{(\mathsf{match}\ (\mathsf{A}\ v)\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3) \to e_2\{v/x\}} \qquad \frac{}{(\mathsf{match}\ (\mathsf{B}\ v)\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3) \to e_3\{v/y\}}
$$

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

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

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

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

$$
\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{letrec}\ f\ x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}
$$

$$
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_3 \qquad \Gamma, y : \tau_2 \vdash e_3 : \tau_3}{\Gamma \vdash (\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_{3)} : \tau_3}
$$

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

$$
\frac{\Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \qquad \frac{}{\tau \leq \tau} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \qquad \frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4}
$$

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

$$
\frac{}{\{l_1{=}\tau_1, \ldots, l_i{=}\tau_i, l_j{=}\tau_j, \ldots, l_n{=}\tau_n\} \leq \{l_1{=}\tau_1, \ldots, l_j{=}\tau_j, l_i{=}\tau_i, \ldots, l_n{=}\tau_n\}}
$$

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

Name:____

1. (**20** points) Suppose we add *division* to our IMP expression language. In Caml, the expression syntax becomes:

```
type exp =
 Int of int | Var of string | Plus of exp * exp | Times of exp * exp | Div of exp * exp
```

Our interpreter (not shown) raises a Caml exception if the second argument to `Div` evaluates to 0. We are ignoring statements; assume an IMP program is an expression that takes an unknown heap and produces an integer.

(a) Write a Caml function `nsz` (stands for "no syntactic zero") of type `exp->bool` that returns false if and only if its argument contains a division where the second argument *is the integer constant 0*. Note we are *not* interpreting the input; `nsz` is *not* even passed a heap.

(b) If we consider division-by-zero at run-time a "stuck state" and `nsz` a "type system" (where true means "type-checks"), then:

i. Is `nsz` sound? Explain.

ii. Is `nsz` complete? Explain.

Solution:

```
let rec nsz e =
  match e with
    Int _ -> true
  | Var _ -> true
  | Plus(e1,e2) -> nsz e1 && nsz e2
  | Times(e1,e2) -> nsz e1 && nsz e2
  | Div(e1,Int 0) -> false
  | Div(e1,e2) -> nsz e1 && nsz e2
```

The type system is not sound: It may accept a program that would get stuck at run-time. For example, `Div(3,x)` would get stuck for any heap that mapped `x` to 0.

The type system is complete: All programs it rejects will get stuck at run-time under any heap. That is because expression evaluation always evaluates all subexpressions, so the division-by-zero will execute. (Substantial partial credit for explaining that code that doesn't execute leads to incompleteness. It just happens that IMP *expressions* do not have code that doesn't execute.)

Name:____

2. (**20** points) Consider this Caml code. It uses `strcmp`, which has type `string->string->bool` and the expected behavior.

```
exception NoValue
let empty = fun s -> raise NoValue
let extend m x v = fun s -> if strcmp s x then v else m s
let lookup m x = m x
```

   (a) *What functionality* do these three bindings provide a client?

   (b) *What types* do each of the bindings have?
   (Note: They are all polymorphic and may have more general types than expected.)

Solution:

   (a) They provide maps from strings to values (where the client chooses the type of the values). `empty` is the empty-map; calling `lookup` with it and any string raises an exception. `extend` creates a larger map from a smaller one (`m`) by having `x` map to `v` (shadowing any previous mapping for `x`) and otherwise using the map `m`.

   (We didn't ask *how* the code works: A map is represented by a Caml function from strings to values, so `lookup` is just function application. `extend` creates a new function that uses `m`, `x`, and `v` as free variables: If the string it is passed is not equal to `x`, then it just applies the smaller map `m` to `s`.)

   (b)
```
empty : 'a -> 'b
extend : (string -> 'a) -> string -> 'a -> (string -> 'a)
lookup : ('a -> 'b) -> 'a -> 'b
```

Name:____

3. (**16** points) When we added sums (syntax $\mathsf{A}\ e$, $\mathsf{B}\ e$, and $\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3$) to the λ-calculus, we gave a small-step semantics and had exactly two constructors.

   (a) Give sums a large-step semantics, still for exactly two constructors. That is, extend the call-by-value large-step judgment $e \Downarrow v$ with new rules. (Use 4 rules.)

   (b) Suppose a program is written with *three* constructors (A, B, and C) and match expressions that have exactly *three* cases:

$$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2\ | \mathsf{B}\ y \to e_3\ | \mathsf{C}\ z \to e_4$$

   Explain a possible *translation* of such a program into an equivalent one that uses only two constructors. (That is, explain how to *translate* the 3 constructors to use 2 constructors and how to *translate* match expressions. Do *not* write inference rules.)

Solution:

(a)

$$\frac{e \Downarrow v}{\mathsf{A}\ e \Downarrow \mathsf{A}\ v} \qquad \frac{e \Downarrow v}{\mathsf{B}\ e \Downarrow \mathsf{B}\ v}$$

$$\frac{e_1 \Downarrow \mathsf{A}\ v_1 \qquad e_2\{v_1/x\} \Downarrow v_2}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3 \Downarrow v_2} \qquad \frac{e_1 \Downarrow \mathsf{B}\ v_1 \qquad e_3\{v_1/y\} \Downarrow v_2}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2 | \mathsf{B}\ y \to e_3 \Downarrow v_2}$$

(b) One solution: Replace every `B e` with `B(A e)` and `C e` with `B(B e)`. Replace every:

$$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2\ | \mathsf{B}\ y \to e_3\ | \mathsf{C}\ z \to e_4$$

with:

$$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{A}\ x \to e_2\ | \mathsf{B}\ q \to (\mathsf{match}\ q\ \mathsf{with}\ \mathsf{A}\ y \to e_3\ | \mathsf{B}\ z \to e_4)$$

Name:____

4. (**14** points) Consider a λ-calculus with *tuples* (i.e., "pairs with any number of fields"), so we have expressions $(e_1, e_2, ..., e_n)$ and $e.i$ and types $\tau_1 * \tau_2 * ... * \tau_n$. For each of our subtyping rules for records, explain whether or not an analogous rule for tuples makes sense.

   Solution:

   - The permutation rule does *not* make sense. Tuple fields are accessed by position so subsuming `string*int` to `int*string` would allow $e.2$ to have type `string` when it should not.
   - The width and depth rules *do* make sense for the same reasons as records: Forgetting about fields on the right means only that fewer expressions of the form $e.i$ will type-check. Assuming tuple-fields are read-only just like record fields, covariant subtyping is correct.

Name:____

5. (**14** points) Assume a class-based object-oriented language as in class, and a program that contains the call `e.f((C)e1)` where `e1` is a (compile-time) subtype of `C` and the whole call type-checks.

   (a) If calls are resolved with static overloading, is it possible that removing the cast `C` (i.e., changing the call to `e.f(e1)`) could cause the program to still type-check but behave differently? Explain.

   (b) If calls are resolved with static overloading and we have multiple inheritance, is it possible that removing the cast `C` (i.e., changing the call to `e.f(e1)`) could cause the program to no longer type-check? Explain.

   (c) If calls are resolved with multimethods, is it possible that removing the cast `C` (i.e., changing the call to `e.f(e1)`) could cause the program to behave differently? Explain.

Solution:

(a) Yes, it is possible. For example, suppose:

- `e2` has type `A`, which is a subtype of `C`.
- `e` has type `D` and class `D` defines methods `f(C)` and `f(A)`.

Now removing the cast results in a different method being called.

(b) Yes, it is possible. For example, suppose:

- `e2` has type `A`, which is a subtype of `C` and `B`.
- `e` has type `D` and class `D` defines methods `f(C)` and `f(B)`, but not `f(A)`.

Now removing the cast results in an ambiguous call.

(c) No, it is not possible. The method called depends on the run-time types of the values that `e` and `e1` evaluate to, and `(C)e1` evaluates to the same value as `e1`.

Name:____

6. (**9** points) Here are two large-step interpreters for the untyped lambda-calculus. The one on the right uses parallelism. Recall `Thread.join` blocks until the thread described by its argument terminates. Only the lines between the `(*----------*)` comments differ.

```
type exp =  Var of string | Lam of string*exp | Apply of exp * exp
let subst e1_with e2_for x = ... (* unimportant *)
exception UnboundVar
```

```
let rec interp e =
  match e with
   Var _ -> raise UnboundVar
 | Lam _ -> e
 | Apply(e1,e2) ->
    (*----------*)


    let v2 = interp e2 in
    let v1 = interp e1 in


    (*----------*)
    match v1 with
     Lam(x,e3) -> interp(subst e3 v2 x)
   | _ -> failwith "impossible"
```

```
let rec interp e =
  match e with
   Var x -> raise UnboundVar
 | Lam _ -> e
 | Apply(e1,e2) ->
    (*----------*)
    let v2r = ref (Var "dummy") in
    let t = Thread.create
       (fun () -> v2r := interp e2) () in
    let v1 = interp e1 in
    Thread.join t;
    let v2 = !v2r in
    (*----------*)
    match v1 with
     Lam(x,e3) -> interp(subst e3 v2 x)
   | _ -> failwith "impossible"
```

(a) Describe an input to these functions for which the interpreter on the right would raise an exception and the interpreter on the left would not. (Note: Evaluation of expressions may not terminate.)

(b) Explain why moving the line "`let v2r = ref (Var "dummy") in`" out to the top-level (and removing the keyword "`in`") would make the interpreter on the right behave unpredictably (even for inputs with no free variables).

Solution:

(a) An argument that applies an expression with an unbound variable to an expression that doesn't terminate shows the difference. For example:

```
App(Var("x"),
    App(Lam ("x", App(Var "x", Var "x")),
        Lam ("x", App(Var "x", Var "x"))))
```

(b) Interpretation could lead to more than two threads running concurrently because of nested applications: An expression like `App(App(e1,e2),App(e3,e4))` would lead to four threads, and using a shared reference leads to a *race condition*: The thread evaluating `App(e1,e2)` may not read the reference set by the thread evaluating `e2` until another thread (e.g., the thread evaluating `e4`) sets the reference to hold another value.

Name:____

7. (**7** points) You can do this problem in one of Caml, C, C++, Java, or C#. Your choice does not really change the problem.

   (a) Write a short program that will exhaust memory if there is no garbage collector but take almost no space if there is a garbage-collector.

   (b) Write a short program that will exhaust memory even if there is a garbage collector. Create only small objects.

Solution:

(a)

```
#include <stdlib.h>
int main() {
  for(;;)
    malloc(4);
}
```

(b)

```
#include <stdlib.h>
struct L { struct L * x; };
struct L * p = NULL;
int main() {
  for(;;) {
    struct L * q = malloc(sizeof(struct L));
    q->x = p;
    p = q;
  }
}
```