Name:_____

CSE P505, Spring 2006
Some Sample Final-Exam Questions

Caveats:

- These questions probably aren't as good as those on the exam. Your instructor wrote some of them quickly, so they may be more "study problems" than "exam problems", but they're certainly in the style of exam problems. Feel free to email if they're ambiguous and/or inscrutable.
- The actual exam will cover some of the same topics and some different ones.
- Some of these sample questions are from old exams given in slightly different classes.
- There are more questions here than on the exam.

Name:_____

1. (Bad statement rules)

   (a) Why do we not have this rule in our IMP statement semantics?

$$\frac{H_0 \ ; \ s_1 \ \Downarrow \ H_1 \qquad H_1 \ ; \ s_2 \ \Downarrow \ H_2 \qquad H_2 \ ; \ s_3 \ \Downarrow \ H_3}{H_0 \ ; \ s_1; (s_2; s_3) \ \Downarrow \ H_3}$$

   (b) Why do we not have this rule in our IMP statement semantics?

$$\frac{H_0 \ ; \ s_2 \ \Downarrow \ H_1 \qquad H_1 \ ; \ s_1 \ \Downarrow \ H_2}{H_0 \ ; \ s_1; s_2 \ \Downarrow \ H_2}$$

Solution:

   (a) It is unnecessary because we can use one of the rules we have twice to derive the same result.

   (b) It is not what we "want" – the purpose of a sequence of statements is to execute the statements *in order*. This rule would make our language non-deterministic in a way we don't want because it lets us execute the two parts of a sequence in either order.

Name:_____

2. (Functional programming)

   (a) Consider this Caml code:

```
type t = A of int | B of (int->int)
let x = 2
let f y = x + y
let ans1 = (let x = 3 in
            let a = A (f 4) in
            let x = 5 in
            match a with A x -> x | B x -> x 6)
let ans2 = (let x = 3 in
            let b = B f in
            let x = 5 in
            match b with A x -> x | B x -> x 6)
```

   After evaluating this code, what values are `ans1` and `ans2` bound to?

   (b) Consider this Caml code:

```
let rec g x =
  match x with
    [] -> []
  | hd::tl -> (fun y -> hd + y)::(g tl)
```

   i. What does this function do?

   ii. What is this function's type?

   iii. Write a function `h` that is the *inverse* of `g`. That is, `fun x -> h (g x)` would return a value equivalent to its input.

Solution:

(a) `ans1` is bound to 6 and `ans2` is bound to 8.

(b) This function takes a list of integers and returns a list of functions where the i^{th} element in the output list returns the sum of its input and the i^{th} element of the input list.

(c) `int list -> ((int -> int) list)`

(d)
```
let rec h x =
  match x with
    [] -> []
  | hd::tl -> (hd 0)::(h tl)
```

Name:_____

3. Assume a typed lambda-calculus with records, references, and subtyping. For each of the following, describe exactly the conditions under which the subtyping claim holds.

   Example question: $\{l_1 : \tau_1,\ l_2 : \tau_2\} \leq \{l_1 : \tau_3,\ l_2 : \tau_4\}$

   Example answer: "when $\tau_1 \leq \tau_3$ and $\tau_2 \leq \tau_4$"

   Your answer should be "fully reduced" in the sense that if you say $\tau \leq \tau'$, then τ or τ' or both should be τ_i for some number i where τ_i appears in the question.

   Note: We did not discuss much (at all?) in P505 that references are like records with one mutable field.

   (a) $(\{l_1{:}\tau_1, l_2{:}\tau_2\}) \to \mathsf{int} \quad \leq \quad (\{l_1{:}\tau_3, l_2{:}\tau_4\}) \to \mathsf{int}$

   (b) $\{l_1{:}(\tau_1\ \mathsf{ref})\} \quad \leq \quad \{l_1{:}\tau_2\}$

   (c) $(\tau_1 \to \tau_2) \to (\tau_3 \to \tau_4) \quad \leq \quad (\tau_5 \to \tau_6) \to (\tau_7 \to \tau_8)$

   (d) $(\tau_1 \to \tau_2)\ \mathsf{ref} \quad \leq \quad (\tau_3 \to \tau_4)\ \mathsf{ref}$

   Solution:

   (a) when $\tau_3 \leq \tau_1$ and $\tau_4 \leq \tau_2$

   (b) when τ_2 has the form $\tau_3\ \mathsf{ref}$, $\tau_3 \leq \tau_1$, and $\tau_1 \leq \tau_3$

   (c) when $\tau_1 \leq \tau_5$, $\tau_6 \leq \tau_2$, $\tau_7 \leq \tau_3$, and $\tau_4 \leq \tau_8$

   (d) when $\tau_1 \leq \tau_3$, $\tau_3 \leq \tau_1$, $\tau_2 \leq \tau_4$, and $\tau_4 \leq \tau_2$

Name:_____

4. (Simply-Typed λ calculus)
For all subproblems, assume the simply-typed λ calculus.

   (a) (**6** points) Give a Γ, e_1, e_2, and τ such that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ and $e_1 \neq e_2$.

   (b) (**6** points) Give a Γ_1, Γ_2, e, and τ such that $\Gamma_1 \vdash e : \tau$ and $\Gamma_2 \vdash e : \tau$ and $\Gamma_1 \neq \Gamma_2$.

   (c) (**8** points) Give a Γ, e, τ_1, and τ_2 such that $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$ and $\tau_1 \neq \tau_2$.

   Solution:

   (a) $\Gamma = x{:}\mathsf{int}, y{:}\mathsf{int}$, $e_1 = x$, $e_2 = y$, $\tau = \mathsf{int}$.

   (b) $\Gamma_1 = x{:}\mathsf{int}$, $\Gamma_2 = x{:}\mathsf{int}, y{:}\mathsf{int}$, $e = x$, $\tau = \mathsf{int}$.

   (c) $\Gamma = \cdot$, $e = \lambda x.\ x$, $\tau_1 = \mathsf{int} \to \mathsf{int}$, $\tau_2 = (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$

Name:_____

5. Our formal inference rule for typing letrec allowed only one recursive function. Give a typing rule (an inference rule for the judgment $\Gamma \vdash e : \tau$) for the extension below. It allows two *mutually recursive functions*. Assume it evaluates to a pair of functions (the first function and then the second function).

$$\mathsf{letrec}\ f\ x.\ e1\ \mathsf{and}\ g\ y.\ e2$$

Solution:

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1, g{:}\tau_3 \to \tau_4 \vdash e_1 : \tau_2 \qquad \Gamma, f{:}\tau_1 \to \tau_2, y{:}\tau_3, g{:}\tau_3 \to \tau_4 \vdash e_2 : \tau_4}{\Gamma \vdash \mathsf{letrec}\ f\ x.\ e1\ \mathsf{and}\ g\ y.\ e2 : (\tau_1 \to \tau_2) * (\tau_3 \to \tau_4)}$$

Name:_____

6. Consider this Caml syntax for a λ-calculus:

```
type exp = Var of string
         | Lam of string * exp
         | Apply of exp * exp
         | Int of int
         | Pair of exp * exp
         | First of exp
         | Second of exp
```

   (a) Write a Caml function `swap` of type `exp->exp` that changes all `Pair` expressions by switching the order of the subexpressions, changes all `First` expressions into `Second` expressions, and changes all `Second` expressions into `First` expressions.

   (b) True or false: Given an implementation of the λ-calculus, `interp(swap(`e`))` is always that same as `interp(`e`)`.

   (c) True or false: Given an implementation of the λ-calculus, if `interp(swap(`e`))` returns `Int` i, then `interp(`e`)` returns `Int` i.

Solution:

(a)
```
let swap e =
  match e with
    Var _ -> e
  | Lam(s,e) -> Lam(s,swap e)
  | Apply(e1,e2) -> App(swap e1, swap e2)
  | Int _ -> e
  | Pair(e1,e2) -> Pair(swap e2, swap e1)
  | First e -> Second (swap e)
  | Second e -> First (swap e)
```

(b) False. For example `First 3` becomes `Second 3`, which is not the same.

(c) True. We consistently swap everything.

Name:_____

7. Consider the following Caml code.

```
let catch_all1 t1 t2 = try t1 () with x -> t2 ()

let catch_all2 t1 t2 = try t1 () with x -> t2
```

   (a) Under what conditions, if any, does using `catch_all1` raise an exception?

   (b) Under what conditions, if any, does using `catch_all2` raise an exception?

   (c) What type does Caml give `catch_all1`? (You can give your answer in Caml notation or System-F notation.)

   (d) What type does Caml give `catch_all2`? (You can give your answer in Caml notation or System-F notation.)

   Solution:

   (a) when calling its first argument raises an exception *and* calling its second argument raises an exception

   (b) never

   (c) Caml: $(\mathsf{unit} \to \alpha) \to (\mathsf{unit} \to \alpha) \to \alpha$

   (d) Caml: $(\mathsf{unit} \to \alpha) \to \alpha \to \alpha$

Name:_____

8. Consider these definitions in a class-based OO language:

```
class C1 {
   int g() { return 0;   }
   int f() { return g(); }
}
class C2 extends C1 {
   int g() { return 1;   }
}
class D1 {
   private C1 x = new C1();
   int g() { return 0;     }
   int f() { return x.f(); }
}
class D2 extends D1 {
   int g() { return 1; }
}
```

```
class Main {
   int m1(C1 x) { return x.f() }
   int m2(C2 x) { return x.f() }
   int m3(D1 x) { return x.f() }
   int m4(D2 x) { return x.f() }
}
```

Assume this is not the entire program, but the rest of the program does not declare subclasses of the classes above.

Explain your answers:

(a) True or false: Changing the body of `m1` to `return 0` produces an equivalent `m1`.

(b) True or false: Changing the body of `m2` to `return 1` produces an equivalent `m2`.

(c) True or false: Changing the body of `m3` to `return 0` produces an equivalent `m3`.

(d) True or false: Changing the body of `m4` to `return 1` produces an equivalent `m4`.

(e) How do your answers change if the rest of the program might declare subclasses of the classes above (excluding `Main`)?

Solution:

(a) false: If `m1` is passed an instance of `C2`, it will return 1.

(b) true: there are no subtypes of `C2`, so any call to `m2` will pass an instance of `C2`, and late-binding ensures the `f` method of a `C2` returns 1.

(c) true: Any call to `m3` will pass an instance of `D1` or `D2`. The `f` methods for both are the same: return the result of `C1`'s `f` method.

(d) false: same reason as previous question

(e) All claims become false because calls to `f` in `Main` could resolve to methods defined in subclasses we do not see above.

Name:_____

9. Suppose we *change the semantics* of Java so that method-lookup uses multimethods instead of static overloading.

   True or false. **Briefly explain your answers.**

   (a) If all methods in program P take 0 arguments (that is, all calls look like $e.m()$), then P definitely behaves the same after the change.

   (b) If all methods in program P take 1 argument (that is, all calls look like $e.m(e')$), then P definitely behaves the same after the change.

   (c) If a program P typechecks without ever using subsumption, then P definitely behaves the same after the change.

   (d) Given an arbitrary program P, it is decidable whether P behaves the same after the change.

   Solution:

   (a) True. The difference between multimethods and static overloading is whether method lookup uses the (compile-time) types or the (run-time) classes of non-receiver (i.e., non-self) arguments. Without any such arguments, this aspect of method-lookup is never used.

   (b) False. Same explanation as in previous part but there are now non-receiver arguments.

   (c) True. Without subsumption, the compile-time type is always the same as the run-time class of every object, so the different method-lookup rules will always produce the same answer (because they are always given the same "input").

   (d) False. We have seen examples of calls that resolve differently for static overloading and multi-methods. Suppose `e.m(e1)` is such a call and P' is a program whose behavior is the same under either semantics (e.g., maybe it has no subsumption). Then P'`;e.m(e1)` behaves the same if and only if P' does not halt. Halting is undecidable because Java is Turing-complete (even without subsumption).

Name:_____

10. Suppose we extend a class-based object-oriented language with a keyword `null`, which has type `NullType`, which is a subtype of any type.

    (a) Explain why the subtyping described above is backwards. How does some popular language you know deal with this?

    (b) With static overloading or multimethods (the issue is the same), show how `null` can lead to ambiguities.

    Solution:

    (a) `null` has no fields or methods, so width subtyping suggests it should be a supertype of other types. Indeed, trying to access a member leads to a "stuck" (message not understood) state. Most languages make this a run-time error (raise an exception in Java or C#; lead to arbitrary behavior in C++).

    (b) Suppose class `C` has two methods `void m(A)` and `void m(B)` where `A` and `B` are not subtypes of each other. Then a call that passes `null` is ambiguous since there are no grounds to prefer one method over the other.

Name:_____

11. (a) Write a Caml program using locks that will always deadlock, but only because the locks provided by the `Mutex` library are not reentrant. (If you forget the names of library functions, just make them up and explain; you'll get full credit.)

    (b) Write a Caml program that will always deadlock even if the locks provided by the `Mutex` library were reentrant. (Assume threads implicitly release all locks when they terminate.) (Note: The hard part is the word "always".)

    Solution:

(a)
```
let lk = Mutex.create()
let _ = Mutex.lock lk
let _ = Mutex.lock lk
```

(b)
```
let lk1 = Mutex.create()
let lk2 = Mutex.create()
let r1 = ref false
let r2 = ref false
let rec until_true r =
  if !r
  then ()
  else (Thread.yield(); until_true r)
let _ = Thread.create (fun () ->
            Mutex.lock lk1;
            r1 := true;
            until_true r2;
            Mutex.lock lk2)
let _ =  Mutex.lock lk2;
         r2 := true;
         until_true r1;
         Mutex.lock lk1
```

Name:_____

12. Suppose a bug in a garbage collector causes it to always treat memory address `0xDEADBEEF` as a root. Give two separate reasons that this single bug could cause a program to leak an arbitrary amount of memory.

Solution:

(a) A program might allocate an arbitrarily large object at this address. Once it becomes garbage, it is a leak.

(b) A program might put the head of an arbitrarily large linked list at this address. The entire list will never be garbage collected, even though each object in the list is small and none of them may be reachable except via the pointer in `0xDEADBEEF`.