Name:____

CSE P505, Autumn 2016
Sample Exam Questions

Notes:

- Most of these questions are from old exams given in slightly different classes, so don't worry too much if one or two problems feel only tangentially related to what you learned. Others were written a bit quickly, so they may be more "study problems" than "exam problems" but they are still in the *style* of exam problems.
- All of these problems were written assuming "closed notes" (but allowing each student to bring one sheet of paper with anything wanted on it). Our exam is "open notes" as explained elsewhere.
- The actual exam will cover some of the same topics and some different ones. There will be roughly 10 questions on the exam, so not everything can be covered. The real exam will cover Haskell and related topics, but none of these sample questions do.

Name:____

1. (a) Write a function `wgo_help1` such that this function:

```
let wgo1 max lst = wgo_help1 0 max lst
```

behaves as follows: If `lst` is a list of integers `[i1;i2;...;in]`, then `wgo1` returns the sum of a prefix of the list `i1 ... ij` such that:

- The sum is less than `max`.
- Either the sum of the next-larger prefix `i1 ... ij i` is not less than `max` or there is no next-larger prefix (i.e., the entire list has a sum less than `max`).

Do not define any other helper functions. Note wgo stands for, "without going over."

(b) What is the type of `wgo_help1`?

(c) Write a function `wgo_help2` such that this function:

```
let wgo2 max lst = wgo_help2 (fun x -> x < max) (fun x y -> x+y) 0 lst
```

has the same observable behavior as `wgo1`. Do not define any other functions. `wgo_help2` should not contain an explicit addition.

(d) What is the type of `wgo_help2`?

(e) Rewrite `wgo1` and `wgo2` to be shorter.

Solution:

(a)
```
let rec wgo_help1 acc max lst =
   match lst with
     [] -> acc
   | hd::tl -> if acc+hd < max
               then wgo_help1 (acc+hd) max tl
               else acc
```

(b) `int -> int -> int list -> int`

(c)
```
let rec wgo_help2 f g acc lst =
   match lst with
     [] -> acc
   | hd::tl -> if f (g hd acc)
               then wgo_help2 f g (g hd acc) tl
               else acc
```

(d) `('a -> bool) -> ('b -> 'a -> 'a) -> 'a -> 'b list -> 'a` Note that it also works to use `(g acc hd)`, which changes the type of the second argument to `'a -> 'b -> 'a`. Substantial partial credit was given for a type with only one type variable.

(e)
```
let wgo1 = wgo_help1 0
let wgo2 max = wgo_help2 (fun x -> x < max) (fun x y -> x+y) 0
```

Name:____

2. Consider this OCaml 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 OCaml 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. (a) Consider this OCaml 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 OCaml 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:____

4. Consider the following OCaml 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 OCaml give `catch_all1`?
   (d) What type does OCaml give `catch_all2`?

Solution:

   (a) when calling its first argument raises an exception *and* calling its second argument raises an exception
   (b) never
   (c) OCaml: $(\mathsf{unit} \to \alpha) \to (\mathsf{unit} \to \alpha) \to \alpha$
   (d) OCaml: $(\mathsf{unit} \to \alpha) \to \alpha \to \alpha$

Name:____

5. Here is our OCaml abstract syntax for IMP with one new kind of statement described below:

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

Recall that in the semantics the expression in an if-statement or while-statement is true if it is not zero.

In this problem, we consider a new kind of statement in IMP. The semantics of `CompareAndSwap(s,e1,e2)` is as follows:

- If evaluating `e1` under the current heap produces the same value that variable `s` holds under the current heap, then update the heap so `s` holds the value that `e2` evaluates to under the current heap.
- Otherwise make no change to the heap.

In OCaml, write a *translation* from IMP-including-compare-and-swap statements to IMP-not-including-compare-and-swap statements. In other words, write a function `translate` of type `stmt -> stmt` such that (1) the result contains no compare-and-swap statements and (2) the result is equivalent to the argument.

Note: Some of you might recognize compare-and-swap as related to concurrency, but this problem has nothing to do with concurrency.

Solution:

```
let rec translate s =
  match s with
    Skip -> s
  | Assign(str,e) -> s
  | Seq(s1,s2) -> Seq(translate s1, translate s2)
  | If(e,s1,s2) -> If(e, translate s1, translate s2)
  | While(e,s) -> While(e, translate s)
  | CompareAndSwap(str,e1,e2) -> If(Plus(e1,Times(Int(-1),Var(str))),
                                    Skip,
                                    Assign(str,e2))
```

Name:____

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

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

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

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

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:____

7. Consider this OCaml 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 an OCaml 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:____

8. In this problem we define a small language that manipulates a stack of strings. You are given the syntax and the informal semantics.

   The language syntax is a command list. A program state contains a command list and a stack of strings (the stack "grows to the right"):

$$\begin{array}{rrcl} \text{string} & str & ::= & \textit{(any string)} \\ \text{command} & c & ::= & \mathsf{push}\ str \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{swap} \\ \text{command-list} & lst & ::= & [] \mid c{::}lst \\ \text{stack} & stk & ::= & \cdot \mid stk, str \end{array}$$

   Informally, the commands behave as follows:

   - push *str* makes a bigger stack with *str* on top.
   - pop makes a smaller stack by removing the top element.
   - dup (short for duplicate) makes a bigger stack by placing a copy of the top stack-element on top.
   - swap swaps the order of the top two elements on the stack.

   A command list executes the commands in order.

   (a) Give large-step inference rules for the judgment $stk_1; lst \Downarrow stk_2$, meaning, "running *lst* starting from stk_1 produces stk_2." One rule is given to you as an example. You need to write down 4 other rules.

$$\frac{stk_1, str;\ lst \Downarrow stk_2}{stk_1;\ (\mathsf{push}\ str){::}lst \Downarrow stk_2}$$

   (b) The semantics can *get stuck*, i.e., there exists stacks stk_1 and command-lists *lst* such that we cannot derive $stk_1; lst \Downarrow stk_2$ for any stk_2. In English, describe why there may be not be a derivation.

   (c) Give a complete derivation that concludes $\cdot; (\mathsf{push}\ \text{“pl”}){::}\mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{“pl”}, \text{“pl”}$

Solution:

(a)

$$\frac{stk_1; lst \Downarrow stk_2}{stk_1, str;\ \mathsf{pop}{::}lst \Downarrow stk_2} \qquad \frac{stk_1, str, str;\ lst \Downarrow stk_2}{stk_1, str;\ \mathsf{dup}{::}lst \Downarrow stk_2} \qquad \frac{stk_1, str_2, str_1;\ lst \Downarrow stk_2}{stk_1, str_1, str_2;\ \mathsf{swap}{::}lst \Downarrow stk_2}$$

$$\frac{}{stk;\ [] \Downarrow stk}$$

(b) Evaluation could require popping or duplicating when the stack is empty or swapping when the stack has zero or one elements.

(c)

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\cdot, \text{“pl”}, \text{“pl”};\ [] \Downarrow \cdot, \text{“pl”}, \text{“pl”}}}{\cdot, \text{“pl”}, \text{“pl”};\ \mathsf{swap}{::}[] \Downarrow \cdot, \text{“pl”}, \text{“pl”}}}{\cdot, \text{“pl”};\ \mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{“pl”}, \text{“pl”}}}{\cdot; (\mathsf{push}\ \text{“pl”}){::}\mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{“pl”}, \text{“pl”}}$$

Name:____

9. 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:____

10. Suppose we add *division* to our IMP expression language. In OCaml, 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 OCaml 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 an OCaml 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:____

11. (a) Recall this typing rule, one of the three rules we added for sums:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}\ e : \tau_1 + \tau_2}$$

Explain this rule in English. In particular, what expressions can this typing rule be used for and what types can it give to such expressions?

(b) Recall this typing rule for functions:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

Explain this rule in English. In particular, what expressions can this typing rule be used for and what types can it give to such expressions?

(c) Suppose we changed the typing rule for functions to the following:

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_2 \to \tau_2}$$

Explain why this change would not violate type safety. Explain why it is a bad idea anyway.

Solution:

(a) This typing rule applies to any expression that uses "tag" A with a subexpression that type-checks under some context. The overall expression then type-checks under the same context. If the subexpression has type τ_1, then the overall expression has type $\tau_1 + \tau_2$ for any type τ_2.

(b) The typing rule applies to any lambda expression, provided the body type-checks when extending the context to map x to some type τ_1. If the body can have type τ_2, then the function can have type $\tau_1 \to \tau_2$.

(c) This change would require the argument and result type for every function to be the same type. As examples, we could have $\mathsf{int} \to \mathsf{int}$ and $(\mathsf{int} * \mathsf{int}) \to (\mathsf{int} * \mathsf{int})$, but not $(\mathsf{int} * \mathsf{int}) \to \mathsf{int}$. This is type-safe: Before the change, only safe programs were accepted and the change causes the type-system to accept only strictly fewer programs. Nonetheless, it is a bad idea because functions that take and return value of different types are useful and we would not want to program in a language that did not allow such functions.

Name:____

12. For all subproblems, assume the simply-typed λ calculus.

    (a) 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) 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) 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:____

13. Suppose you design a new type system for Java to prevent null-pointer dereferences. However, due to poor design, your type system has the strange property that there are exactly 47 programs that your type system accepts; it rejects all others.

    Explain your answers *briefly.*

    (a) Is it possible that your type system is sound with respect to null-pointer dereferences?

    (b) Is it possible that your type system is complete with respect to null-pointer dereferences?

    (c) Is it definitely the case given just the information above that your type system is sound with respect to null-pointer dereferences?

    (d) Is it definitely the case given just the information above that your type system is complete with respect to null-pointer dereferences?

    Solution:

    (a) Yes, soundness means the type system accepts no programs that dereference null. That might be the case for the 47 accepted programs.

    (b) No, completeness means the type system rejects no programs that definitely do not dereference null. There are an infinite number of such programs, and we are rejecting all but 47 of them.

    (c) No, some of the 47 programs might be able to dereference null.

    (d) No, it is not even possible (see part (b)).

Name:____

14. 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:____

15. 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) $(\{l_1{:}\tau_1, l_2{:}\tau_2\}) \rightarrow \mathsf{int} \quad \leq \quad (\{l_1{:}\tau_3, l_2{:}\tau_4\}) \rightarrow \mathsf{int}$

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

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

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

    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:____

16. 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:____

17. Consider a typical class-based OOP language like we did in class. Suppose somewhere in a program that type-checks we have `e.m((D)(new C()))` where `C` is a subclass/subtype of `D`. Notice the argument in the method call is an explicit upcast. Consider modifying the program by removing this explicit upcast, i.e., replacing the call with `e.m(new C())`.

    Explain your answers *briefly.*

    (a) If every class in the program has at most one method named `m`, can this change cause the program not to type-check?

    (b) If every class in the program has at most one method named `m`, can this change cause the program to produce a different result?

    Solution:

    (a) No. The program type-checked when the argument had type `D` and it can still have type `D` via subsumption since `C`≤`D`.

    (b) No, the same method would still be called with the same object.

Name:____

18. 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. [WE DIDN'T STUDY THESE TOPICS.]

    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:____

19. THIS PROBLEM ALSO PROBABLY COVERS MATERIAL WE WON'T GET TO.

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