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.

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.)

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.

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`?

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.

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?

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

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.

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"):

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

   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 $\mathit{stk}_1; \mathit{lst} \Downarrow \mathit{stk}_2$, 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.

$$\frac{\mathit{stk}_1, \mathit{str};\ \mathit{lst} \Downarrow \mathit{stk}_2}{\mathit{stk}_1;\ (\mathsf{push}\ \mathit{str}) :: \mathit{lst} \Downarrow \mathit{stk}_2}$$

   (b) The semantics can *get stuck*, i.e., there exists stacks stk_1 and command-lists *lst* such that we cannot derive $\mathit{stk}_1; \mathit{lst} \Downarrow \mathit{stk}_2$ for any stk_2. In English, describe why there may be not be a derivation.

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

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.)

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.

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.

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 $\tau_1 \neq \tau_2$.

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?

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.

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

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

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`)?

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?

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.]

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?