Name:_____

CSE P505, Winter 2009
Sample Exam Questions

Notes:

- Some 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.
- The actual exam will cover some of the same topics and some different ones. There were be roughly 6–8 questions on the exam, so not everything can be covered.
- You can bring one sheet of paper to the exam with whatever you want on it (front and back). The idea is to make sure memorization isn't an issue, but the exam is not "open notes" which just encourages everyone to print everything remotely related to the course.
- The actual exam will have a reference page or two with some relevant inference rules and functions from class. That does not mean the whole exam is related to the reference material. You will also need to be able to read/write Caml programs, OO-style programs, etc. Most of the reference material will in fact not be directly helpful with the exam, but this way you know they will be there and you don't have to put them on your sheet of paper.
- The exam will likely be tough, perhaps with a mean around 70%. Such an exam is actually much more fair than an exam with a mean above 90%

Name:_____

1. 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:_____

2. (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. 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`?

   (d) What type does Caml give `catch_all2`?

   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:_____

4. (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:_____

5. 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:_____

6. 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:_____

7. 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:_____

8. 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:_____

9. 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:_____

10. 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:_____

11. 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:_____

12. 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:_____

13. 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 multimethods. 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:_____

14. 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:_____

15. 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

Name:_____

16. 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:_____

17. (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:_____

18. Use Concurrent ML to complete an implementation of "infinite" arrays (in the sense that no index is out of bounds), without using Caml's references or arrays. More specifically, implement the `new_array` function for this code:

```
(* Interface *)
type 'a myarray
val new_array : 'a -> 'a myarray (* Initially, every index maps to 'a *)
val set : 'a myarray -> int -> 'a -> unit (* change value in index *)
val get : 'a myarray -> int -> 'a  (* return current value in index *)

(* Implementation *)
open Event
open Thread
type 'a myarray = ((int * ('a channel)) channel) * ((int * 'a) channel)

let new_array init = (* for you *)
let set (_,c) i v =
   sync (send c (i,v))
let get (c,_) i =
   let ret = new_channel() in
   sync (send c (i,ret));
   sync (receive ret)
```

Hints: Do *not* worry about being efficient. Have `set` work in $O(1)$ time and `get` work in (worst-case) $O(n)$ time where n is the number of `set` operations preceding it. Use an association list. Sample solution is about 15 lines, including a short helper function for traversing a list. Use `choose` and `wrap`.

Solution:

```
let rec assoc lst i default =
  match lst with
    [] -> default
  | (j,v)::tl -> if i=j then v else assoc tl i default
let new_array init =
   let getter,setter = new_channel(), new_channel() in
   let rec server lst =
      sync (choose [
      wrap (receive getter)
           (fun (i,c) -> (sync (send c (assoc lst i init))); server lst);
      wrap (receive setter)
           (fun pr -> server (pr::lst))
      ]) in
   ignore (Thread.create server []); getter,setter
```

Name:_____

19. 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:_____

20. 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`.

Name:_____

21. 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;
  }
}
```