Name:____

CSE 341, Fall 2004, Final Examination
16 December 2004

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please write your name on each page.
- There are a total of **6 questions** (each with multiple parts) and **85 points**. The questions are worth between 12 and 15 points.
- When writing code, style matters, but don't worry about indentation.

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.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:______________________________________

1. For each of the following parts of Scheme programs, decide if the function bound to `f` is *pure*, i.e., it has no *visible effects* and it always returns the same answer (or produces the same error) given the same argument. (Note: We are asking about the function that is (initially) bound to `f`, so it doesn't matter if the variable `f` is mutated.)
   Briefly explain your answers.

   (a) **(3 points)**

   ```
   (define f
     (let ([y 3])
       (lambda (x)
         (+ x y))))
   ```

   (b) **(3 points)**

   ```
   (define y 3)
   (define f
     (lambda (x)
       (+ x y)))
   ```

   (c) **(3 points)**

   ```
   (define f
     (lambda (x)
       (begin (set! x (+ x 3))
              x)))
   ```

   (d) **(3 points)**

   ```
   (define f
     (lambda (x)
       (begin (set-cdr! x (+ (cdr x) 3))
              (cdr x))))
   ```

   (e) **(3 points)**

   ```
   (define f
     (lambda (x)
       (begin (set-cdr! (cons (car x) (cdr x)) (+ (cdr x) 3))
              (cdr x))))
   ```

   Solution:

   (a) Pure: The function always adds 3 to its argument because nothing it uses from its environment can ever be mutated.

   (b) Not pure if subsequent code mutates `y`.

   (c) Pure: The mutation is only to a local variable; the function always adds 3 to its argument.

   (d) Not pure: The function mutates the pair it is passed; this effect is visible to callers.

   (e) Pure: The mutation is to a fresh object that is not returned; the functions always returns the cdr of its argument.

Name:____

2. Consider this definition for RIBIT. (It is similar to FROG from homework 5, but functions are anonymous and take one argument (not a list), there are no lists, and there are no exceptions.)

```
(define-struct var  (str)) ; variables
(define-struct int  (num)) ; constants
(define-struct add  (e1 e2)) ; addition (like e1 + e2 in ML)
(define-struct alet (str e1 e2)) ; let-binding (like let str = e1 in e2 in ML)
(define-struct fun  (str e1)) ; anonymous function (like fn str=>e1 in ML)
(define-struct app  (e1 e2)) ; application (like e1(e2) in ML)
; a ribit expression is made from one of the 6 constructors defined above
; semantics is lexically scoped functions, like ML, Scheme, and frog
```

You may *assume* that fields named `str` hold Scheme strings, fields named `num` hold Scheme integers, and fields named `e1` or `e2` hold RIBIT expressions.

(a) **(7 points)** Write a Scheme function `no-ribit-negs` that takes a Ribit program *p* and returns `#t` if *p* contains no negative constants and `#f` otherwise. (Do *not* evaluate the program; just compute whether it contains a negative constant.)

(b) **(4 points)** Suppose we want to forbid RIBIT programs that *evaluate* to negative constants. (A run-time failure or another final value is fine.) Is `no-ribit-negs` a sound compile-time check? Is `no-ribit-negs` a complete compile-time check? **Explain.**

(c) **(4 points)** Suppose we want to forbid *Scheme* programs that *evaluate* to negative constants and `no-scheme-negs` determines if a Scheme progrm *p* has no negative constants. Is `no-scheme-negs` a sound compile-time check? Is `no-scheme-negs` a complete compile-time check? **Explain.**

Solution:

(a)
```
(define (no-ribit-negs p)
        (cond [(var? p) #t]
              [(int? p) (>= (int-num p) 0)]
              [(add? p) (and (no-ribit-negs (add-e1 p))
                             (no-ribit-negs (add-e2 p)))]
              [(alet? p) (and (no-ribit-negs (alet-e1 p))
                              (no-ribit-negs (alet-e2 p)))]
              [(fun? p) (and (no-ribit-negs (fun-e1 p)))]
              [(app? p) (and (no-ribit-negs (app-e1 p))
                             (no-ribit-negs (app-e2 p)))]))
```

(b) It is sound: The only numbers in a ribit program can be constants and the result of adding ribit numbers. So if none of the constants are negative, no number in the program can be negative, so the final result cannot be negative. It is *not* complete: For example, it rejects `(make-add (make-int -2) (make-int 3))`, but this program does not evaluate to a negative constant.

(c) It is *not* sound: For example, it accepts `(- 3 4)`, but this program evaluates to a negative constant. It is *not* complete: For example it rejects `(lambda (x) -3)`, but this program evaluates to a function.

Name:____

3. (a) **(6 points)** Define a Smalltalk class `CounterBlock` with one method `newCounter`, which returns a block taking no arguments. When sent the `value` message, the block should return a 3-element array:

- First element: how many times that block has been sent `value`.
- Second element: how many times any block created by the *same* instance of `CounterBlock` has been sent `value`.
- Third element: how many times any block created by *any* instance of `CounterBlock` has been sent `value`.

Remember `Object` accepts the message
`subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`
Just assume your `newCounter` method is "in `CounterBlock`".

(b) **(6 points)** Suppose `SubCounterBlock` is some subclass of `CounterBlock`.
Determine if each claim below is "always true", "sometimes true", or "never true". **Explain.**
Assume `x := (((CounterBlock new) newCounter) value) at: 3`.

i. `SubCounterBlock` does *not* override `newCounter` and
`x` *does* include sends to blocks created by instances of `SubCounterBlock`.

ii. `SubCounterBlock` does *not* override `newCounter` and
`x` does *not* include sends to blocks created by instances of `SubCounterBlock`.

iii. `SubCounterBlock` *does* override `newCounter` and
`x` *does* include sends to blocks created by instances of `SubCounterBlock`.

iv. `SubCounterBlock` *does* override `newCounter` and
`x` does *not* include sends to blocks created by instances of `SubCounterBlock`.

(c) **(3 points)** After the following code executes, can any objects that it allocates be garbage-collected? Explain.
`x := CounterBlock new. y := x newCounter. x := nil`

Solution:

(a)
```
Object subclass: #CounterBlock
        instanceVariableNames: 'c2'
        classVariableNames: 'C1'
        poolDictionaries: '' category: 'final'
newCounter
     |c3|
     C1 ifNil: [C1 := 0].
     c2 ifNil: [c2 := 0].
     c3 := 0.
     ^[ C1 := C1 + 1. c2 := c2 + 1. c3 := c3 + 1. {c3. c2. C1} ]
```

(b) i. "always true": The behavior of `newCounter` will not change; it is not overridden and it does not call any overridden methods.

ii. "never true": Same reason as previous part.

iii. "sometimes true": It depends what the overriding is. In the extreme, it could just be `super newCounter`.

iv. "sometimes true": It depends what the overriding is. If it is something like `0`, then the counts won't be adjusted.

(c) No. The block bound to `y` is still reachable via `y`. The `Counterblock` instance (in particular, its field that holds a counter) is still reachable from the block's environment, so it cannot be collected even though `x` is `nil`. Put another way, if we collected the `Counterblock` instance, a subsequent `y value` would access reclaimed memory.

Name:____

4. Suppose Smalltalk had multiple inheritance, with the following restriction: Resolving a message to a method must never be ambiguous. That is, if message m is sent to an instance of class C, then we get the *same* method by starting at C and following *any* sequence of superclass references. The Smalltalk implementation must reject any operation that makes this "all paths are the same" property false.

   (a) **(9 points)** For each of these operations, explain how the operation could make the "all paths are the same" property false. (Assume it was true to begin with.)

   - A new class `C` is created with superclasses `A` and `B`.
   - A method `m` is added to a class `D`.
   - A method `m` is removed from a class `E`.

   (b) **(4 points)** Suppose classes `F` and `G` both have a method `m` and no subclasses. You want to create a class `H` with superclasses `F` and `G`, but at no point can you violate the "all paths are the same" property. Describe a sequence of six operations such that:

   - None of the operations are rejected.
   - *After the sequence*, `H` exists and instances of `F` and `G` behave just as they did before.

Solution:

(a)
- If `A` and `B` both define methods `m`, then `(C new) m` is ambiguous.
- If `D` has a subclass `S1` that has another superclass `S2` and `S1` does not define `m`, then `(S1 new) m` is ambiguous.
- If `E` has supertypes `S1` and `S2` and at least defines `m`, then `(E new) m` is ambiguous.

(b)
- Remove `m` from `F`.
- Remove `m` from `G`.
- Create `H`.
- Add a method `m` to `H`.
- Add the method `m` removed from `F` back to `F`.
- Add the method `m` removed from `G` back to `G`.

Name:____

5. Consider these Java classes, but do *not* assume these are the only classes in the program.

```
class D {
  void m(A x) { System.out.println("A"); }
  void m(B x) { System.out.println("B"); }
  void m(C x) { System.out.println("C"); }
}
class A {
   void f() { (new D()).m(this); }
   void g(D d) { d.m(this); }
}
class B extends A {}
class C extends B {
   void f() { super.f(); (new D()).m(this); }
}
```

(a) **(6 points)** Recall Java has static overloading.

i. What does `(new A()).f()` print?
ii. What does `(new B()).f()` print?
iii. What does `(new C()).f()` print?
iv. Does `(new A()).g(`e`)` print the same thing for any possible e (that type-checks)? If so, what does it print? If not, why not?

(b) **(6 points)** Suppose Java had multimethods and not static overloading. In this situation:

i. What does `(new A()).f()` print?
ii. What does `(new B()).f()` print?
iii. What does `(new C()).f()` print?
iv. Does `(new A()).g(`e`)` print the same thing for any possible e (that type-checks)? If so, what does it print? If not, why not?

Solution:

(a) i. A
ii. A
iii. A
C
iv. No, we might pass a subclass of `D` which could override the method `m` that takes an `A`.

(b) i. A
ii. B
iii. C
C
iv. (same as part (a)) No, we might pass a subclass of `D` which could override the method `m` that takes an `A`.

Name:____

6. Suppose Java did *not* allow subsumption for array types. Consider this Java method, which under our assumption can never raise an exception:

```
final static void copy(C[] arr1, C[] arr2) {
  if(arr1 != null && arr2 != null) {
    for(int i=0; i < arr1.length && i < arr2.length; i++)
      arr1[i] = arr2[i];
  }
}
```

Suppose `C` extends `B` and `D` extends `C`.

In answering the questions below, remember that you *know* the implementation of `copy` and it cannot be overridden.

(a) **(3 points)** Is it sound to type-check a program as though `copy` had type `void copy(B[],C[])`? (It is not sound if it could lead to accepting a program that could have a message-not-understood error.)

(b) **(3 points)** Is it sound to type-check a program as though `copy` had type `void copy(C[],B[])`?

(c) **(3 points)** Is it sound to type-check a program as though `copy` had type `void copy(D[],C[])`?

(d) **(3 points)** Is it sound to type-check a program as though `copy` had type `void copy(C[],D[])`?

(e) **(3 points)** If Java had bounded quantification, what is the most general (but sound) type we could give to `copy`?

Solution:

(a) Yes, sound: We only write to `arr1`, so if we pass in an array of `B` objects, we will write `C` objects into it, but `C` is a subtype of `B`.

(b) No, unsound: It would allow us to write a `B` object into an array of `C` objects. If we then assumed an array-element was a `C` and not a `B` we could get a message-not-understood error.

(c) No, unsound: It would allow us to write a `C` object into an array of `D` objects. If we then assumed an array-element was a `D` and not a `C` we could get a message-not-understood error.

(d) Yes, sound: We only read from `arr2`, so if we pass in an array of `D` objects, we can treat each element we read as a `C` because `D` is a subtype of `C`.

(e) `forall 'a1 forall 'a2 where 'a2 <: 'a1. final static void copy('a1, 'a2);`