Name:_____

CSE 341, Spring 2004, Final Examination
10 June 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 **98 points**, distributed unevenly among **8 questions** (with subparts).
- 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. Answer every question as best you can!
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. In each part below, we define a Scheme function and a Scheme macro. If *every* use of the macro and the function would behave equivalently (same result and effects), write "equivalent". If not, give an example in which using the macro would print different text than using the function. (Recall the function `print` takes a string and prints it.)

   (a) **(3 pts)**

```
(define (unless-num e1 e2)
   (if (number? e1) e1 e2))
(define-syntax unless-num
   (syntax-rules ()
     [(unless-num e1 e2)
      (let ([x e1] [y e2])
        (if (number? x) x y))]))
```

   (b) **(3 pts)**

```
(define (unless-num e1 e2)
   (if (number? e1) e1 e2))
(define-syntax unless-num
   (syntax-rules ()
     [(unless-num e1 e2)
      (let ([x e2] [y e1])
        (if (number? y) y x))]))
```

   (c) **(3 pts)**

```
(define (unless-num e1 e2)
   (if (number? e1) e1 e2))
(define-syntax unless-num
   (syntax-rules ()
     [(unless-num e1 e2)
      (let ([x e1])
        (if (number? x) x e2))]))
```

   (d) **(3 pts)**

```
(define (never-num e1)
   (if (number? e1) #t e1))
(define-syntax never-num
   (syntax-rules ()
     [(never-num e1)
      (if (number? e1) #t e1)]))
```

Name:_____

2. For each program below, fill in the blanks to create a program such that all the following are true:

   - The program will be "accepted" (typecheck, etc.)
   - The program will not go into an infinite loop
   - *The program will **not** print anything.* (The underlaying language implementation may print something, but not because the `print` function or `show:` method was called.)
   - The program does not contain the empty-string (`""` or `''`)
   - The program does not declare a variable named `print`

   There are additional requirements for the Scheme program.

   (a) **(2 pts)** ML:

   ```
   _____
   val x = print _____
   ```

   (b) **(3 pts)** Smalltalk:

   ```
   Transcript show: _____
   ```

   (c) **(4 pts)** Scheme: You must not cause an error (such as a dynamic-typing error or a use of `error`).

   ```
   (_____
     (print _____))
   ```

Name:_____

3. Recall the language of propositional logic, summarized and simplified as follows:

```
var  ::= <<variables such as x, y, etc.>>
form ::= true | false | var | (not form) | (form1 and form2) | (form1 or form2)
```

The evaluation of a formula uses a *truth-table* to map *variables* to `true` or `false`. The ordinary rules of logic apply to evaluate *formulas* to `true` or `false`.

This problem considers "non-standard type systems" for formulas. The goal of each type system is to prevent *contradictions*. By definition, a contradiction is a formula that evaluates to `false` for every truth-table.

For each type system below, choose one of the following: (1) it is *sound-but-not-complete*, (2) it is *complete-but-not-sound*, (3) it is *sound-and-complete*, or (4) it is *neither-sound-nor-complete*. If (1), give a formula that demonstrates incompleteness. If (2), give a formula that demonstrates unsoundness. If (4), give a formula demonstrating incompleteness and a different formula demonstrating unsoundness.

(a) **(3 pts)** A type system that rejects every formula except the single formula "`true`"

(b) **(3 pts)** A type system that accepts every formula except the single formula "`false`"

(c) **(4 pts)** A type system that accepts a formula if and only if the formula has no uses of the constant `false`.

(d) **(4 pts)** A type system that for each formula:

- Finds all the variables used in the formula.
- Generates every truth-table containing those variables.
- Evaluates the formula under every such truth-table.
- Accepts if and only if one or more evaluations produces `true`.

(e) **(2 pts)** Why is part (d) a legitimate "type system" (despite being weird and impractical) whereas an analogous approach (evaluation under all possibilities) to type-checking a general-purpose programming language would not work?

Name:_____

4. This problem considers closure-conversion, as investigated in homework 5. A full sample solution appears on the last page of the exam, but only the translation of function applications is relevant.

   Review: Closure-conversion produces an *equivalent program with no free variables*. Pseudocode for the result of translating `(app ea eb)` follows, where `e1` and `e2` are the translations of `ea` and `eb`.

```
let pr = e1 in
let arg = e2 in
(app (fst pr) (list (snd pr) arg))
```

   But minfun does not have let, so the sample solution actually uses a two-argument function to encode the two uses of let. Pseudocode:

```
(app (fun (pr arg) (app (fst pr) (list (snd pr) arg))) (list e1 e2))
```

   But we learned ways to simulate multiple arguments with one argument, so maybe we can use a one-argument function instead.

   (a) **(5 pts)** Suppose we change the application case to build a one-argument function, where the argument is a pair of the two arguments in the current solution. The underlined portion of the sample solution becomes:

```
(make-app
  (make-fun (list prsym) ; now one argument
            (make-app (make-fst (make-fst prsym))
                      (list (make-snd (make-fst prsym)) (make-snd prsym))))
  (list (make-pr e1 e2))) ; now pass a pair
```

   Is this modified solution a correct closure-conversion implementation? Explain briefly.

   (b) **(5 pts)** Suppose we change the application case to use curried one-argument functions. The underlined portion of the sample solution becomes:

```
(make-app
(make-app
  (make-fun (list prsym) ; now one argument
     (make-fun (list argsym)
        (make-app (make-fst prsym)
                  (list (make-snd prsym) argsym))))
  (list e1)) ; now pass one argument at a time
  (list e2))
```

   Is this modified solution a correct closure-conversion implementation? Explain briefly.

Name:_____

5. Write Smalltalk methods as described. It does not really matter what class the methods belong to, but assume they are class methods for class `A`.

   (a) **(6 pts)** Write a method `compose:with:` that composes two blocks expecting one argument. That is, if x and y are blocks expecting one argument, then `A compose:`x `with:`y should return a block z expecting one argument. Thinking of x, y, and z as functions, applying z to an argument should be equivalent to applying x to the result of applying y to the same argument.

   (b) **(3 pts)** Write a method `addN:` that returns a block expecting one argument. The returned block should evaluate to the argument it is passed plus the argument to `addN`.

   (c) **(3 pts)** Write a method `mulN:` that returns a block expecting one argument. The returned block should evaluate to the argument it is passed times the argument to `mulN`.

   (d) **(2 pts)** What is the result of evaluating the following?

```
(A compose: (A addN: 3) with: (A mulN: 4)) value: 5
```

Name:_____

6. This problem considers a Smalltalk-like language with static types. For example, the type `[x:t1]` describes objects with a field `x` of type `t1`. All access to `x` (or any field) is via a getter method `x` that returns an object of type `t1` and a setter method `x:` that takes an object of type `t1`. Subtypes can have more fields than supertypes. For example, `[x:t1, a:t2]` is a subtype of `[x:t1]`.

   Suppose we have a method m that is type-safe if m is given an argument of type `[x:[y:int,z:int]]`. Suppose the following code is type-safe *assuming* it is sound to call m with arguments of type `[x:t]`. (What `t` equals is different in different problems.)

   e_1`.`
   m`(`e_2`).`
   e_3

   (a) **(6 pts)** Suppose executing m may cause the getter method `x` to be called before m returns, but we somehow know that the setter method `x:` will *not* be called before m returns.

      i. Given these assumptions, is it always sound to allow calls to m with arguments of type `[x:t]` if `t<:[y:int,z:int]`? Explain briefly.

      ii. Given these assumptions, is it always sound to allow calls to m with arguments of type `[x:t]` if `[y:int,z:int]<:t`? Explain briefly.

   (b) **(6 pts)** Suppose executing m may cause the setter method `x:` to be called before m returns, but we somehow know that the getter method `x` will *not* be called before m returns.

      i. Given these assumptions, is it always sound to allow calls to m with arguments of type `[x:t]` if `t<:[y:int,z:int]`? Explain briefly.

      ii. Given these assumptions, is it always sound to allow calls to m with arguments of type `[x:t]` if `[y:int,z:int]<:t`? Explain briefly.

Name:_____

7. Consider this Java code, where underlining is used only to highlight some differences:

```
class Pt {
  int x;
  int y;
  boolean equals(Pt p) {
    return (this.x == p.x && this.y == p.y && true);
  }
}
class Pt3D extends Pt {
  int z;
  boolean equals(Pt3D p) {
    return (this.x == p.x && this.y == p.y && this.z == p.z);
  }
  boolean f(Pt p1, Pt p2, Pt p3) {
    return (p1.equals(p2) && p2.equals(p3) && p3.equals(p1));
  }
  boolean g(Pt p1, Pt p2, Pt p3) {
    return (p1.equals(p2) && p2.equals(p3) && true);
  }
}
```

   (a) **(6 pts)** (Recall Java has static overloading.) Do `f` and `g` always produce the same answer given the same arguments? Explain briefly.

   (b) **(6 pts)** Suppose we change Java so all methods are multimethods. Do `f` and `g` always produce the same answer given the same arguments? Explain briefly.

Name:_____

8. Here are three instance methods for a class `B` with one instance variable, `bag` (which you can assume other code does not access):

```
f: x
   ^ ... "... is some complicated computation"
g: x
   |ans|
   (bag isNil) ifTrue: [bag := Bag new]. "Bag is a Collection with add: and do:"
   bag do: [:pr |  ((pr at:1) = x) ifTrue: [ans := (pr at:2)]].
   (ans isNil) ifFalse:  [^ans].
   ans := self f:x.
   bag add: {x. ans}. "send an array object with two elements"
   ^ans
h
  i := 0.
  [i < 1000000] whileTrue: [self g: i. i := i+1].
  [i < 2000000] whileTrue: [self f: i. i := i+1]
```

(a) **(2 pts)** Under what conditions are `f:` and `g:` equivalent? Assume these conditions hold in remaining problems.

(b) **(3 pts)** Describe a client of an instance of `B` where replacing uses of `f:` with uses of `g:` would make the client faster.

(c) **(3 pts)** Describe a client of an instance of `B` where replacing uses of `g:` with uses of `f:` would make the client faster.

(d) **(5 pts)** Consider this instance method of another class and assume a program calls `foo` once.

```
foo
   |x|
   x := B new.
   x h
```

i. After the call `x h` in `foo`, approximately how many objects are reachable from `x` (not including integers or the 1 object bound to `self`)? Explain your answer.

ii. When is the last time any of these objects reachable from `x` are accessed?

iii. When is the earliest the garbage collector could reclaim the objects reachable from `x`?

```
(define-struct fun  (args body)) ;; args a list of identifiers
(define-struct app  (e1 args))   ;; args a list of minfun expressions
(define-struct if1  (test iftrue iffalse)) ;; 3 subexpressions
(define-struct mul   (e1 e2))
(define-struct add   (e1 e2))
(define-struct is-eq (e1 e2)) ;; allowed only on numbers
(define-struct pr (e1 e2))
(define-struct fst (e1))
(define-struct snd (e1))
(define-struct set-fst! (e1 e2))
(define-struct set-snd! (e1 e2))
(define-struct is-pr (e1))

(define (get-env-exp e arg-stack env-exp)
  (cond [(null? arg-stack) (error e "unbound minfun variable")]
        [(eq? (car arg-stack) e) (make-fst env-exp)]
        [#t (get-env-exp e (cdr arg-stack) (make-snd env-exp))]))
(define (convert-body arg-stack arg e env-var)
  (letrec ([f
            (lambda (e)
              (cond [(number? e) e]
                    [(symbol? e)
                     (if (eq? e arg)
                         e
                         (get-env-exp e arg-stack env-var))]
                    [(fun? e) ;; only supports 1-arg functions
                     (let ([new-env-var (gensym)]
                           [new-arg-stack
                            (if arg
                                (cons arg arg-stack)
                                arg-stack)])
                       (make-pr (make-fun (cons new-env-var (fun-args e))
                                          (convert-body new-arg-stack
                                                        (car (fun-args e))
                                                        (fun-body e)
                                                        new-env-var))
                                (if arg (make-pr arg env-var) env-var)))]
                    [(app? e) ;; only supports 1-arg functions
                     (let ([e1 (f (app-e1 e))] ; convert e1
                           [e2 (f (car (app-args e)))] ; convert the "old" argument
                           [prsym (gensym)] ; a name for the closure
                           [argsym (gensym)]) ; a name for the "old" argument
                       (make-app
                        (make-fun (list prsym argsym)
                                  (make-app (make-fst prsym)
                                            (list (make-snd prsym) argsym)))
                        (list e1 e2))
                        )]
                    [(if1? e) (make-if1 (f (if1-test e)) (f (if1-iftrue e)) (f (if1-iffalse e)))]
                    [(mul? e)      (make-mul      (f (mul-e1 e))      (f (mul-e2 e)))]
                    [(add? e)      (make-add      (f (add-e1 e))      (f (add-e2 e)))]
                    [(is-eq? e)    (make-is-eq    (f (is-eq-e1 e))    (f (is-eq-e2 e)))]
                    [(pr? e)       (make-pr       (f (pr-e1 e))       (f (pr-e2 e)))]
                    [(set-fst!? e) (make-set-fst! (f (set-fst!-e1 e)) (f (set-fst!-e2 e)))]
                    [(set-snd!? e) (make-set-snd! (f (set-snd!-e1 e)) (f (set-snd!-e2 e)))]
                    [(fst? e)   (make-fst   (f (fst-e1 e)))]
                    [(snd? e)   (make-snd   (f (snd-e1 e)))]
                    [(is-pr? e) (make-is-pr (f (is-pr-e1 e)))]))])
    (f e)))
(define (convert e)
  (convert-body () #f e 99))
```