Feedback — Actual Final Exam

Help

You submitted this exam on **Mon 22 Dec 2014 4:51 PM PST**. You got a score of **35.00** out of **100.00**. However, you will not get credit for it, since it was submitted past the deadline.

You have 90 minutes to complete the exam. Only your first submission will count toward your grade.

You may use any course materials (videos, slides, reading notes, etc.). You may use the ML, Racket, and Ruby REPLs. You may use text editors. You may use the standard-library documentation for the languages.

You may not use the discussion forum. You may not use other websites related to programming. (Sites like dictionaries for translating English words are okay to use.)

Question 1

[14 points] Check a box if and only if it is an accurate description of Racket.

Your Answer		Score	Explanation
☐ `(define (f x y) e)` is syntactic sugar for the curried function definition `(define f (lambda (x) (lambda (y) e)))`.	✓	2.00	
☐ Without `let*`-expressions, Racket programmers could just use nested `let`-expressions, but the result would have more parentheses.	✗	0.00	
☐ It is a compile-time error for the first argument to an `if` expression not to be a boolean.	✓	2.00	
☐ It is a run-time error for the first argument to an `if` expression not to be a boolean.	✓	2.00	
☐ A struct definition for a struct with *n* (immutable) fields adds *n*+2 functions to the environment.	✗	0.00	

☐ A struct definition is syntactic sugar for introducing several functions that operate over Racket lists.	✓	2.00
☐ A function call always evaluates each argument exactly once, but a macro use may not evaluate each argument exactly once.	✗	0.00
Total		8.00 / 14.00

Question 2

[4 points] This incorrect Racket code is supposed to bind to `longer-strings` a stream where the *N*th element of the stream is the string containing the character `A` *N* times.

```
(define longer-strings
  (lambda ()
    (letrec ([f (lambda(s)
                  (cons s (f (string-append "A" s))))])
      (f "A"))))
```

What is wrong with this code?

Your Answer	Score	Explanation
○ `longer-strings` is bound to a function, but it should be bound to a pair.		
○ Calls to `longer-strings` will never terminate because there is too little thunking.		
○ Calls to `longer-strings` will never terminate because the function bound to `f` needs a conditional.		
○ Calls to `longer-strings` will never terminate because the function bound to `f` is returning a procedure somewhere where it needs to return a call to the procedure.		
Total	0.00 / 4.00	

Question 3

[4 points] Which statement below accurately describes the function bound to `mystery` in this Racket code?

```
(define (mystery s)
  (lambda ()
     (let ([pr (s)])
        (if (car pr)
            (cons (car pr) (mystery (cdr pr)))
            ((mystery (cdr pr)))))))
```

Your Answer	Score	Explanation
○ It takes a stream and generates all its elements, causing an infinite loop for any call to `mystery`.		
○ It takes a stream and generates a list of its elements up to the first `#f` in the stream.		
○ It takes a list and returns a stream that repeatedly generates the elements in the list in order.		
○ It takes a stream and returns a stream that is like the stream it takes except all `#f` elements are removed.		
Total	0.00 / 4.00	

Question 4

[5 points] What is the difference between these two pieces of Racket code? (Here we assume `e1` and `e2` are arbitrary, unspecified Racket expressions. We also assume `e1` does not contain a use of `y`.)

```
(define f (let ([x e1]) (lambda (y) e2))) ; call this code A

(define f (lambda (y) (let ([x e1]) e2))) ; call this code B
```

Your Answer	Score	Explanation
○ Code A evaluates `e1` once whereas Code B evaluates `e1` once every time the function bound to `f` is called.		
○ Code B evaluates `e1` once whereas Code A evaluates `e1` once every time the function bound to `f` is called.		
○ Code A evaluates `e1` only if, at run-time, `e2` uses the variable `x`, but Code B always evaluates `e1` assuming `f` is used at least once.		
○ Code B evaluates `e1` only if, at run-time, `e2` uses the variable `x`, but Code A always evaluates `e1` assuming `f` is used at least once.		
○ There is no semantic difference: although the order of the code is different, code A and code B are equivalent for any `e1` and `e2`.		
Total	0.00 / 5.00	

Question 5

[5 points] In this question, RUPL is like the language MUPL except it is *r*eally small, containing only integers, variables, additions, and let-expressions. What is wrong with this implementation?

```
(struct var  (string) #:transparent)  ;; a variable, e.g., (var "foo")
(struct int  (num)    #:transparent)  ;; a constant number, e.g., (int 17)
(struct add  (e1 e2)  #:transparent)  ;; add two expressions
(struct mlet (var e body) #:transparent) ;; a local binding (let var = e in
 body)

(define (envlookup env str)
  (cond [(null? env) (error "unbound variable during evaluation" str)]
        [(equal? (car (car env)) str) (cdr (car env))]
        [#t (envlookup (cdr env) str)]))
(define (eval-under-env e env)
  (cond [(var? e) (envlookup env (var-string e))]
        [(int? e) e]
        [(add? e)
         (let ([v1 (eval-under-env (add-e1 e) env)]
               [v2 (eval-under-env (add-e2 e) env)])
           (if (and (int? v1)
                    (int? v2))
               (int (+ (int-num v1)
                       (int-num v2)))
               (error "RUPL addition applied to non-number")))]
        [(mlet? e)
         (let ([v (eval-under-env (mlet-e e) env)])
           (eval-under-env (mlet-body e) env))]
        [#t (error "bad RUPL expression")]))
(define (eval-exp e)
  (eval-under-env e null))
```

Your Answer	Score	Explanation
○ The case for variables is wrong because we should recursively call `eval-under-env` in this case.		
○ The case for integer expressions is wrong: we should return `(int-num e)`.		
○ The case for addition expressions is wrong: we should write `e1` and `e2` where we have `(add-e1 e)` and `(add-e2 e)`.		
○ The case for mlet-expressions is wrong: we do not use the correct environment to evaluate the let-expression body.		
Total	0.00 / 5.00	

Question 6

[8 points] In this question, we consider what would happen if we ported (i.e., rewrote) Racket code to ML. Assume we write the code by only changing the syntax as follows: Racket functions become ML functions, Racket conditionals become ML conditionals, Racket addition becomes ML addition, Racket `car` becomes ML `hd`, and Racket `null` becomes ML `[]`. For each function below, check the box if and only if the ML rewrite of the function *would* type-check (with some type). (Always assume we port the code so that the ML code parses correctly.)

Your Answer	Score	Explanation

□ `(define (f1 x) (if x 37 42))`	✗	0.00	
□ `(define (f2 x) (if x x x))`	✗	0.00	
□ `(define (f3 x) (if x 42 x))`	✓	1.00	
□ `(define (f4 x) (car null))`	✗	0.00	
□ `(define (f5 x) (+ (car x) 42))`	✗	0.00	
□ `(define (f6 x) (car (+ x 42)))`	✓	2.00	
Total		3.00 / 8.00	

Question 7

[13 points] For each of the following, check the box if and only if it is an accurate description of an advantage of static typing over dynamic typing.

Your Answer		Score	Explanation
□ Static typing catches some simple bugs without having to test your code.	✗	0.00	
□ Static typing can produce faster code because the language implementation does not need to perform type tests at run time.	✗	0.00	
□ Static typing lets you change the type of a function as its requirements evolve without ever having to change any of the function's callers.	✓	3.00	
□ Static typing is necessary to avoid the security and reliability problems of weak typing.	✓	2.00	
□ Static typing does not make sense for OOP.	✓	2.00	
Total		7.00 / 13.00	

Question 8

[9 points] This question uses this Ruby class definition:

```
class A
  attr_accessor :x
  def m1
    @x = 4
  end
  def m2
    m1
    @x > 4
  end
  def m3
    @x = 4
    @x > 4
  end
  def m4
    self.x = 4
    @x > 4
  end
end
```

For each statement below, check the box if and only if the statement is true. In all cases, consider only a definition of class `B`, not code that makes any changes to class `A`.

Your Answer		Score	Explanation
☐ It is possible to define a class `B` such that evaluating `B.new.m2` causes the method `m2` defined in class `A` (*not* an override of `m2`) to return `true`.	✗	0.00	
☐ It is possible to define a class `B` such that evaluating `B.new.m3` causes the method `m3` defined in class `A` (*not* an override of `m3`) to return `true`.	✓	3.00	
☐ It is possible to define a class `B` such that evaluating `B.new.m4` causes the method `m4` defined in class `A` (*not* an override of `m4`) to return `true`.	✗	0.00	
Total		3.00 / 9.00	

Question 9

[4 points] This problem uses this Ruby class definition, which includes a mixin:

```
class MyRange
  include Enumerable
  def initialize(low,high)
    @low = low
    @high = high
  end
  def each
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```

Given this definition, the expression `MyRange.new(4,2).any? {|i| i <= 4}` evaluates to `false`. Why?

Your Answer	Score	Explanation
○ Because instances of `MyRange` do not have a method `any?`.		
○ Because the `each` method for the object created by `MyRange.new(4,2)` never calls its block.		
○ Because the superclass of `MyRange` is `Object`, which has an `any?` method that always returns `false`.		
○ Because the `each` method in `MyRange` implicitly returns `nil` and in Ruby `nil` is like `false`.		
Total	0.00 / 4.00	

Question 10

[14 points] Check the box if and only if the statement is true.

Your Answer		Score	Explanation
☐ In Ruby, it is a run-time error to create an array holding instances of different classes.	✓	2.00	
☐ In Ruby, you cannot store a block in an array, but you can pass a block to `lambda` and store the result in an array.	✗	0.00	
☐ It does not make sense to consider adding multiple inheritance to a dynamically typed language because the purpose of multiple inheritance is to make type-checking less restrictive.	✓	2.00	
☐ In Ruby, `is_a?` and `instance_of?` are synonyms: the two methods are defined for every object and compute the same result.	✓	2.00	
☐ In Ruby, anything returned by a method is an object.	✗	0.00	
☐ Double dispatch is special to Ruby -- it is a programming pattern that does not work in most other OOP languages.	✓	2.00	
☐ A Ruby mixin method included in a class can get and set instance variables of `self`.	✗	0.00	
Total		8.00 / 14.00	

Question 11

[4 points] This problem and the next problem relate to this Ruby code:

```
class A
  def initialize a
    @arr = a
  end
  def get i
    @arr[i]
  end
  def sum
    @arr.inject(0) {|acc,x| acc + x}
  end
end

class B < A
  def initialize a
    super
    @ans = false
  end
  def sum
    if !@ans
      @ans = @arr.inject(0) {|acc,x| acc + x}
    end
    @ans
  end
end
```

Which technique that we studied is mostly closely related to the code in class B?

Your Answer	Score	Explanation
○ Thunking		
○ Memoization		
○ Mixins		
○ Double dispatch		
Total	0.00 / 4.00	

Question 12

[4 points] This problem uses the code in the previous problem. Class `A` and class `B` are not equivalent. In particular, there are ways to fill in the ... in the code below so that `s3` and `s4` hold different numbers. Which change would make the two classes equivalent?

```
v = [4,19,74]
a = A.new v
b = B.new v
s1 = a.sum
s2 = b.sum
```

```
...
s3 = a.sum
s4 = b.sum
```

Your Answer	Score	Explanation
○ Have the `initialize` method in class `A` store a copy of its argument in `@arr`.		
○ Remove the method `get` from class `A`.		
○ Change the `sum` method in both classes to use an explicit loop instead of `inject` and a block.		
○ Change class `A` to use a class variable `@@arr` in place of the instance variable `@arr`.		
Total	0.00 / 4.00	

Question 13

[12 points] This problem uses the made-up language from the lectures for studying subtyping.

Recall:

- The language has records with mutable fields.
- We write types for records and functions like in ML.
- Records have width and permutation subtyping.
- Function subtyping has contravariant arguments and covariant results.

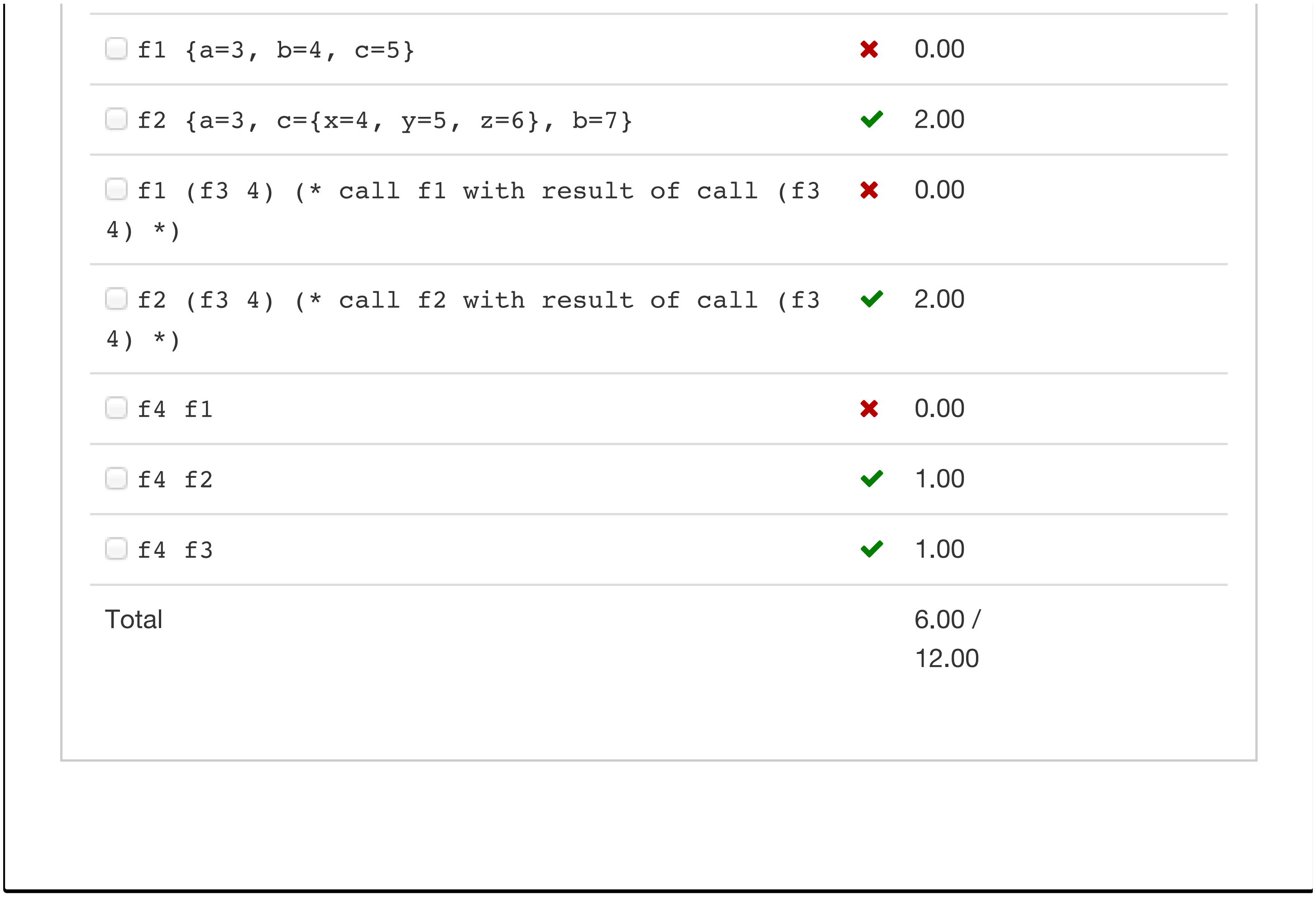
Assume these bindings for functions exist and have given types:

```
val f1 : {a:int, b:int} -> {a:int, b:int};
val f2 : {a:int, c:{x:int, y:int}, b:int} -> {a:int, b:int};
val f3 : int -> {a:int,b:int,c:int};
val f4 : ({a:int,b:int,c:int} -> {a:int,b:int}) -> int;
```

For example, `f1` is bound to a function that takes a record of type `{a:int, b:int}` and returns a record of the same type.

For each call below, check the box if and only if the call type-checks.

Your Answer	Score	Explanation



□ `f1 {a=3, b=4, c=5}`	✗	0.00
□ `f2 {a=3, c={x=4, y=5, z=6}, b=7}`	✓	2.00
□ `f1 (f3 4) (* call f1 with result of call (f3 4) *)`	✗	0.00
□ `f2 (f3 4) (* call f2 with result of call (f3 4) *)`	✓	2.00
□ `f4 f1`	✗	0.00
□ `f4 f2`	✓	1.00
□ `f4 f3`	✓	1.00
Total		6.00 / 12.00