Practice Final Exam Questions

Help

Warning: The hard deadline has passed. You can attempt it, but **you will not get credit for it**. You are welcome to try it as a learning exercise.

These are the instructions that will also be on the real final, but of course this is just a practice final:

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.)

- [x] **In accordance with the Coursera Honor Code, I (KL Tah) certify that the answers here are my own work.** **Thank you!**

Question 1

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

- [ ] Removing parentheses from around a function call can change a program's meaning, but it is always okay to add an extra set of parentheses.
- [ ] The library function `set-mcar!` can be implemented in terms of `set!`.
- [ ] An anonymous function can be written anywhere an expression is allowed.
- [ ] Racket is a weakly typed language.
- [ ] Thunking is a programming idiom that does not need special support in the language beyond first-class function closures.

- [ ] At the top-level in a file, an earlier function definition can refer to a later function definition.

Question 2

[5 points] Which of the following Racket functions correctly takes a stream `s` and returns a stream that repeats each stream element from `s` twice? So, for example, calling the function with the stream that generates `1 2 3` ... would return a stream that generates `1 1 2 2 3 3` ...

- ```
  (define (twice-each s)
    (lambda ()
      (let ([pr (s)])
        (cons (car pr)
              (lambda ()
                (cons (car pr)
                      (twice-each (cdr pr))))))))
  ```

- ```
  (define (twice-each s)
      (let ([pr (s)])
        (cons (car pr)
              (lambda ()
                (cons (car pr)
                      (twice-each (cdr pr)))))))
  ```

- ```
  (define (twice-each s)
    (lambda ()
      (let ([pr (s)])
        (cons (car pr)
              (cons (car pr)
                    (twice-each (cdr pr)))))))
  ```

- ```
  (define (twice-each s)
      (let ([pr (s)])
        (cons (car pr)
              (lambda ()
                (cons (cdr pr)
                      (twice-each ((cdr pr))))))))
  ```

Question 3

[8 points] Which of the following is true of the *promises* we studied and defined using `my-force` and `my-delay` in Racket?

- [ ] Evaluating `(my-force (my-delay (lambda () e)))` always produces the same result as evaluating `e`.
- [ ] Evaluating `(my-delay (my-force (lambda () e)))` always produces the same result as evaluating `e`.
- [ ] Evaluating `(my-force (my-delay (my-delay (lambda () e))))` always produces the same result as evaluating `e`.
- [ ] Evaluating `(my-force (my-delay (my-delay (lambda () e))))` will *not* evaluate `e`.

Question 4

[8 points] Let `e` be a Racket expression. Consider these two versions of function `f`:

```
(define (f) ; call this version A
   (let ([x e])
       (if x x 42)))
```

```
(define (f) ; call this version B
   (if e e 42))
```

Check the box below if and only if there exists any `e` such that the statement is true. Note different choices below can be true by choosing a different `e` for each, but in the code above we mean the same expression `e` wherever `e` appears.

- [ ] Calling version A is equivalent to calling version B.
- [ ] Calling version A evaluates `e` more times than calling version B.
- [ ] Calling version A does not terminate but calling version B does terminate.
- [ ] Version A and version B both always return `42`.

Question 5

[5 points] Suppose a MUPL interpreter implements lexical scope for function calls *incorrectly* by evaluating the function body using the environment where the function is called rather than where it is defined. Which of the MUPL programs below is a good test for this bug because `(eval-exp ...)` will produce the wrong answer if `...` is replaced with the MUPL program.

A few relevant Racket struct definitions for MUPL are repeated here as a convenient reminder:

```
(struct var  (string) #:transparent)              ;; a variable, e.g., (va
r "foo")
(struct int  (num) #:transparent)                 ;; a constant number, e.
g., (int 17)
(struct fun  (nameopt formal body) #:transparent)  ;; a recursive(?) 1-argu
ment function
(struct call (funexp actual) #:transparent)       ;; function call
(struct mlet (var e body) #:transparent)          ;; a local binding (let
var = e in body)
```

- ```
  (mlet "x" (int 0)
        (mlet "x" (int 1)
              (mlet "f" (fun #f "y" (var "x"))
                    (call (var "f") (var "x")))))
  ```

- ```
  (mlet "f" (mlet "x" (int 0)
                  (fun #f "y" (var "x")))
        (mlet "x" (int 1)
              (call (var "f") (var "x"))))
  ```

- ```
  (mlet "x" (int 0)
        (mlet "f" (fun #f "y"
                       (mlet "x" (int 1) (var "x")))
              (call (var "f") (var "x"))))
  ```
  ```

```
(mlet "x" (int 0)
      (call (fun #f "y" (var "x"))
            (var "x")))
```

Question 6

[5 points] Suppose we change ML so that its type system works as follows: If a function takes a tuple with *n* pieces, then you can call that function with any tuple with *m* pieces as long as *m* >= *n*. (At run-time, the extra tuple components would be evaluated before calling the function and then ignored by the function body.) We do not change what the ML type system is supposed to prevent except for this particular change of allowing tuples that are "too big" for function calls. Which of the following is true?

- The type system before this change is sound and complete and after this change is sound and complete.
- The type system before this change is sound and not complete and after this change is sound and not complete.
- The type system before this change is sound and complete and after this change is sound and not complete.
- The type system before this change is sound and complete and after this change is not sound but is complete.
- The type system before this change is sound and not complete and after this change is not sound and not complete.

Question 7

[10 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.

- [ ]

- [ ] If you are writing a function/method that should be called only with values of one type, it is more convenient because you do not need to add a run-time type test to the function/method body.
- [ ] Static checking catches all the simple bugs, so your testing only has to test how multiple functions/methods are used together.
- [ ] If you change the return type of a function/method, the type-checker will give a list of callers that still assume the old return type.
- [ ] Languages with static type systems cannot have security bugs where "anything might happen."
- [ ] A language with static typing always supports generic types, which are more powerful than subtyping.

Question 8

[12 points] This question uses this Ruby code, where `...` is assumed to be some correct code not relevant to the question.

```
class A
  def m1
    self.m2()
  end
  def m2
    ...
  end
end
module M
  def m3
    self.m4()
  end
end
class B < A
  def m2
    ...
  end
end
class C < A
```

```
  include M
  def m4
    ...
  end
end
class D < B
  include M
end
```

For each Ruby expression below, check the box if evaluating the expression would lead to a method-missing error.

- [ ] `B.new.m1`
- [ ] `B.new.m3`
- [ ] `C.new.m1`
- [ ] `C.new.m3`
- [ ] `D.new.m1`
- [ ] `D.new.m3`

Question 9

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

- [ ] It should be a run-time error in Ruby to send the `push` message to an array holding 0 elements.
- [ ] In Ruby, a method can use `yield` to call the block provided by the caller even if the method did not indicate that it expected a block.
- [ ] In Ruby, the class of object `nil` is `nil`, so no methods are defined on this object.
- [ ] In OOP, defining a method in a class that just returns the value of a field (what Ruby calls an instance variable) is not useful: it is analogous to unnecessary function wrapping.
- [ ] Ruby has a notion of subclassing but no notion of subtyping.
- [ ] Dynamic dispatch means that a call like `self.foo` in a method defined in class `A` could call code in a subclass of `A` that the writer of the class `A` code does not know exists.

- [ ] If class `C` overrides method `m` from a superclass, then in the body of `C`'s definition of `m`, the keyword `super` is just syntactic sugar for `m`.

Question 10

[5 points] Which of the following programming problems would lead to a solution using double dispatch in Ruby if the programmer wanted to maintain a "full" commitment to OOP?

- ( ) Provide a reusable `Button` class for a graphical interface that subclasses can specialize by changing the size, color, and text of the button.
- ( ) Provide a class like ML's lists that has a `sort` method that reuses methods provided by standard-library mixins.
- ( ) Provide classes that represent each of the "essential" amino acids, each with a method `resultOfCombining` that takes an argument that is another "essential" amino acid and returns the protein resulting from combining the two amino acids.
- ( ) Provide a mixin that is like `Enumerable` except it raises an error if any of its methods are used on an empty collection, where the notion of emptiness is provided by classes via an `empty?` method.

Question 11

[16 points] In this problem, suppose we add record subtyping and function subtyping to ML. Because ML records are immutable (there is no way to assign to a field after a record is created), depth subtyping is sound for records. So assume record subtyping supports width, permutation, and depth, and that function subtyping supports contravariant arguments and covariant results.

For each of the function calls below, check the box if and only if the function call should type-check assuming these variables have the types indicated:

```
(* assume these variables are bound to functions with the given types; they
```

```
 are used below *)
val f1 : { a:int, b : { c:int, d:int } } -> { a:int } = ...
val f2 : { a:int } -> { a:int, b : { c:int, d:int } } = ...
val f3 : { a:int, b : { c:int, d:int } } -> { a:int, b : { c:int, d:int } }
 = ...
val f4 : (({ a:int, b : { c:int} } -> { a:int }) * int) -> { a:int } = ...
val r1 : { a:int } = { a = 1 }
val r2 : { a:int, b : { c:int} } = { a=1, b = { c=2 } }
val r3 : { a:int, b : { c:int, d:int}, e:int } = { a=1, b = { c=2, d=3}, e=
4 }
val r4 : { a:int, b : { c:int, d:int, e:int }} = { a=1, b = { c=2, d=3, e=4
} }
```

- [ ] f1 r1
- [ ] f1 r2
- [ ] f1 r3
- [ ] f1 r4
- [ ] f2 r1
- [ ] f2 r2
- [ ] f2 r3
- [ ] f2 r4
- [ ] f4(f1,42)
- [ ] f4(f2,42)
- [ ] f4(f3,42)

- [x] **In accordance with the Coursera Honor Code, I (KL Tah) certify that the answers here are my own work.** **Thank you!**

Submit Answers Save Answers