Name:_____

CSE341 Spring 2017, Final Examination
June 8, 2017

Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, etc. except for *both* sides of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- There are **125 points**, distributed **unevenly** among **9** questions (all with multiple parts):
- **The exam is printed double-sided.**

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**17** points) (Racket programming) This problem uses the two Racket struct definitions below for defining binary trees of numbers where `(empty)` is an empty tree (contains zero numbers). Note: A binary tree is not necessarily a binary *search* tree — it is fine for any numbers to appear at any position in a tree.

```
(struct node (left num right) #:transparent)
(struct empty () #:transparent)
```

This problem has parts (a)–(e), some of them on the next page, which should provide more than enough space.

   (a) Draw a simple picture of the tree represented by the value bound to `part-a` in this code:

```
(define (single n) (node (empty) n (empty)))
(define part-a (node (single 3) 4 (single 5)))
```

   (b) Now write a Racket definition such that `part-b` is bound to the tree represented by this picture (where 5 is the root and any not-shown children are empty trees):

```
    5
   /
  4
 /
3
```

   (c) Write a Racket function `forall` that takes two arguments: (1) a function taking one argument and (2) a binary tree. `forall` should evaluate to `#f` if and only if there is some number (anywhere) in the second argument such that the first argument returns `#f` when passed that number. (Note `forall` should return `#t` if the second argument is the empty tree.)

Name:_____

(d) Use `forall`, which you wrote in the previous problem, to write a Racket function `all-in-range` that takes two numbers `i` and `j` and one tree `t` and evaluates to `#t` if all numbers in `t` are greater than or equal to `i` and less than or equal to `j`.

(e) Consider this code that uses the `forall` function you wrote:

```
(define (gross-abuse-of-forall t)
  (let ([x 0])
    (begin (forall (lambda (n)
                     (begin (set! x (+ x 1))
                            #t))
                   t)
           x)))
```

i. Describe in at most one English sentence what `gross-abuse-of-forall` computes. (Do not describe how it does so — describe what clients would "see".)

ii. Write a Racket function `foo` that is equivalent to `gross-abuse-of-forall` but does not use `forall` and does not use mutation.

Solution:

(a)
```
    4
   / \
  3   5
```

(b)
```
(define part-b (node (node (single 3) 4 (empty)) 5 (empty)))
```

(c)
```
(define (forall f t)
  (if (empty? t)
      #t
      (and (f (node-num t))
           (forall f (node-left t))
           (forall f (node-right t)))))
```

(d)
```
(define (all-in-range i j t)
  (forall (lambda (n) (and (<= i n) (<= n j))) t))
```

(e) It returns the number of numbers in the tree given as the one argument.

```
(define (foo t)
  (if (empty? t)
      0
      (+ 1 (foo (node-left t)) (foo (node-right t)))))
```

Name:_____

2. (**9** points) Consider this (silly) Racket code:

```
(define x 7)

(define fa
  (lambda (x)
    (let ([y (+ x 1)])
      (+ y x))))

(define fb
  (let ([y (+ x 1)])
    (lambda (x)
      (+ y x))))

(define-syntax fc
  (syntax-rules ()
    [(fc y)
     (+ x y 1)]))

(set! x 15)

(define a (fa 4))
(define b (fb 4))
(define c (fc 4))
```

After all this code is evaluated:

(a) What is `a` bound to?

(b) What is `b` bound to?

(c) What is `c` bound to?

Solution:

(a) 9
(b) 12
(c) 20

Name:_____

3. (**10** points) (Streams) Remember a stream is a thunk that returns a pair where the cdr is a stream.

   (a) Write a Racket function `n-at-once` that takes a stream `s` and a number `n` and returns a pair where the car is a list holding the first `n` elements of `s` in order and the cdr is the stream that is like `s` but with the first `n` elements of `s` removed. Assume `n` is a non-negative number and `s` is a stream.

   Hint: Sample solution is about 6 lines and uses recursion but does not define any helper functions.

   (b) Fill in the three blanks below so that `part-b` is bound to `'(7 8 9 10)`.

```
(define (f i) (lambda () (cons i (f (+ i 1)))))


(define part-b (_________ (n-at-once (f _________) ________)))
```

Solution:

(a)
```
(define (n-at-once s n)
  (if (= n 0)
      (cons null s)
      (let* ([pr (s)]
             [r (n-at-once (cdr pr) (- n 1))])
        (cons (cons (car pr) (car r)) (cdr r)))))
```

(b)
```
(define part-b (car (n-at-once (f 7) 4)))
```

Name:_____

4. (**25** points) (Interpreter implementation) Below is an implementation for a *subset* of MUPL, but `eval-under-env` contains *several bugs*, which means the language implementation is *wrong*.

```
(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 ifnz   (e1 e2 e3)   #:transparent) ;; if not zero e1 then e2 else e3
(struct mlet   (var e body) #:transparent) ;; a local binding (let var = e in body)
(struct apair  (e1 e2)      #:transparent) ;; make a new pair
(struct first  (e)          #:transparent) ;; get first part of a pair
(struct second (e)          #:transparent) ;; get second part of a pair

(define (envlookup env str) ; envlookup is exactly what we gave you for Homework 5
  (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 "MUPL addition applied to non-number")))]
        [(ifnz? e) (let ([v1 (eval-under-env (ifnz-e1 e) env)]
                         [v2 (eval-under-env (ifnz-e2 e) env)]
                         [v3 (eval-under-env (ifnz-e3 e) env)])
                     (if (int? v1)
                         (if (= (int-num v1) 0) v2 v3)
                         (error "MUPL ifnz applied to non-number")))]
        [(mlet? e) (let* ([v (eval-under-env (mlet-e e) env)])
                     (eval-under-env (mlet-body e)
                                     (list (cons (mlet-var e) v))))]
        [(apair? e) (apair (eval-under-env (apair-e1 e) env)
                           (eval-under-env (apair-e2 e) env))]
        [(first? e) (if (apair? (first-e e))
                        (apair-e1 (first-e e))
                        (error "MUPL first applied to non-apair"))]
        [(second? e) (if (apair? (second-e e))
                         (apair-e2 (second-e e))
                         (error "MUPL second applied to non-apair"))]
        [#t (error (format "bad MUPL expression: ~v" e))]))
(define (eval-exp e) ; eval-exp is exactly what we gave you for Homework 5
  (eval-under-env e null))
```

For each of the expressions *on the next page*, give two things:

- What the correct answer is, i.e., what the interpreter *should* produce.
- What the actual answer is, i.e., what the interpreter above *actually* produces. For some of the expressions, the interpreter does the correct thing, so the answer is the same as the correct answer. However, due to bugs, the interpreter could either produce a different answer or fail with an error.

For example, an answer for `(eval-exp (add (int 2) (int 2)))` could be:

- correct: `(int 4)`
- actual: `(int 4)`

Name:_____

(a) `(eval-exp (first (apair (apair (int 3) (int 4)) (int 5))))`

(b) `(eval-exp (first (apair (add (int 3) (int 4)) (int 5))))`

(c) `(eval-exp (first (first (apair (apair (int 3) (int 4)) (int 5)))))`

(d) `(eval-exp (mlet "x" (int 4) (ifnz (var "x") (var "x") (var "y"))))`

(e) `(eval-exp (mlet "x" (int 4) (ifnz (var "x") (int 5) (int 6))))`

(f) `(eval-exp (mlet "x" (int 4) (mlet "y" (int 4) (ifnz (var "x") (int 5) (int 6)))))`

(g) `(eval-exp (mlet "y" (int 4) (mlet "x" (int 4) (ifnz (var "x") (int 5) (int 6)))))`

Solution:

(a) correct: `(apair (int 3) (int 4))`
actual: `(apair (int 3) (int 4))`

(b) correct: `(int 7)`
actual: `(add (int 3) (int 4))`

(c) correct: `(int 3)`
actual: the error `MUPL first applied to non-apair`

(d) correct: `(int 4)`
actual: the error `undefined variable during evaluation "y"`

(e) correct: `(int 5)`
actual: `(int 6)`

(f) correct: `(int 5)`
actual: the error `undefined variable during evaluation "x"`

(g) correct: `(int 5)`
actual: `(int 6)`

Name:_____

5. (**14** points) (Static vs. Dynamic Typing) Consider this Racket function:

```
(define (sort-pair p)
  (cond [(or (not (pair? p))
             (not (number? (car p)))
             (not (number? (cdr p))))
         (error "I am so very sorry but sort-pair cannot handle your argument")]
        [(>= (car p) (cdr p)) p]
        [#t (cons (cdr p) (car p))]))
```

(a) Port `sort-pair` to ML by defining an analogous function `sort_pair` type `int * int -> int * int`, removing any logic that serves no purpose in ML.

(b) Suppose we now wish to change `sort-pair` in Racket so that `(sort-pair null)` evaluates to `null`. Describe exactly how to make this change. (You can write the whole new function if you wish, but you can also describe it precisely to avoid so much writing.)

(c) After making the change in part (b), would a previous caller of `sort-pair` now need to be modified? Answer "definitely", "maybe", or "definitely not." If "definitely" or "maybe", explain the easiest way to find a caller that needs changing.

(d) Make the corresponding change to `sort_pair` in ML by completing this code so that `sort_pair` has type `(int * int) option -> (int * int) option`.

```
fun sort_pair opt =
  case opt of
      SOME (x,y) =>

    | NONE =>
```

(e) After making the change in part (d), would a previous caller of `sort_pair` now need to be modified? Answer "definitely", "maybe", or "definitely not." If "definitely" or "maybe", explain the easiest way to find a caller that needs changing.

Solution:

(a) `fun sort_pair (x,y) = if x >= y then (x,y) else (y,x)`

(b) Immediately after the keyword `cond`, add `[(null? p) p]`. (Other answers possible.)

(c) "definitely not" is the expected answer, but answers that discuss callers that *expect* an error when calling `sort-pair` with `null` can be given full credit.

(d)
```
fun sort_pair opt =
  case opt of
      SOME (x,y) => SOME (if x >= y then (x,y) else (y,x))
    | NONE => NONE
```

(e) Definitely: no previous calls to `sort_pair` still type-check, so running the type-checker identifies them all. (Technically this might not be true with polymorphic expressions and such but that is not an answer we expect/consider, so with an appropriate explanation "maybe" can receive full credit.)

Name:_____

6. (**15** points) (OOP) This problem considers this Ruby code, which we put in two columns just to make it easier to see all at once.

```
class CoinCollection

end
class Penny < CoinCollection
  def value
    1
  end
end
class Nickel < CoinCollection
  def value
    5
  end
end
class Dime < CoinCollection
  def value
    10
  end
end
```

```
class Quarter < CoinCollection
  def value
    25
  end
end
class NoCoins < CoinCollection
  def value
    0
  end
end
class TwoCollections < CoinCollection
  def initialize (cc1,cc2)
    @cc1 = cc1
    @cc2 = cc2
  end
  def value
    @cc1.value + @cc2.value
  end
end
```

```
cs0 = TwoCollections.new(Dime.new,NoCoins.new)
cs1 = TwoCollections.new(Nickel.new,TwoCollections.new(cs0,Nickel.new))
part_a = cs1.value
```

(a) What is `part_a` bound to in the code above?

(b) Add three method definitions to the code above (indicate what classes you are adding them to) such that if `c` is an instance of any subclass of `CoinCollection`, then `c.numCoins` evaluates to the number of coins in the collection (e.g., `cs1.numCoins == 3`).

(c) Use Ruby's `Comparable` mixin and 1 added method definition so that any two instances of `CoinCollection` can be compared with `>`, `<`, `>=`, etc. in terms of their values. For example, `Dime.new > Nickel.new == true`. Hint: `<=>`

(d) After your additions in the previous two problems, indicate for each of the following whether evaluating the expression causes an error ("yes" for error and "no" for not an error).

i. `Penny.new > 0`

ii. `0 > Penny.new`

iii. `Penny.new.value > 0`

iv. `0 > Penny.new.value`

Solution:

(a) 20

(b)
```
class CoinCollection
  def numCoins
    1
  end
end
```

```
class NoCoins
  def numCoins
    0
  end
end
```

```
class TwoCollections
  def numCoins
    @cc1.numCoins + @cc2.numCoins
  end
end
```

(c)
```
class CoinCollection
  include Comparable
  def <=> other
    value <=> other.value
  end
end
```

(d)
i. yes
ii. yes
iii. no
iv. no

Name:_____

7. (**14** points) (OOP vs. FP) Consider the Ruby code in the previous problem, including all the provided code and your additions for (b), but *not* your additions for (c). Port this code to ML using a functional style by defining:

   - One `datatype` binding with six constructors.
   - Two functions `numCoins` and `value`.
   - Top-level bindings for `cs0`, `cs1`, and `part_a`.

   Solution:

```
datatype coin_collection = Penny | Nickel | Dime | Quarter | NoCoins
                         | TwoCollections of coin_collection * coin_collection

fun numCoins cc =
  case cc of
      NoCoins => 0
    | TwoCollections(cc1,cc2) => numCoins cc1 + numCoins cc2
    | _ => 1

fun value cc =
  case cc of
     Penny => 1
   | Nickel => 5
   | Dime => 10
   | Quarter => 25
   | NoCoins => 0
   | TwoCollections(cc1,cc2) => value cc1 + value cc2

val cs0 = TwoCollections(Dime,NoCoins)
val cs1 = TwoCollections(Nickel,TwoCollections(cs0,Nickel))
val part_a = value cs1
```

Name:_____

8. (**6** points) (Arrays and Blocks)

Consider again the Ruby code for Problem 6. Write a method `m` that takes an argument `arr` that we *assume* is an array where every array element is either `nil` or an instance of a subclass of `CoinCollection`. Implement `m` so it returns the total value of the coins in the array.

For example, `m [Penny.new, nil, TwoCollections.new(Dime.new,Dime.new)]` should evaluate to 21.

Solution:
There are many solutions. Here are two:

```
def m arr
  arr.inject(0) {|x,y| x + (if y.nil? then 0 else y.value end)}
end
def m arr
  ans = 0
  arr.each {|x| if x.is_a? CoinCollection then ans = ans + x.value end}
  ans
end
```

Name:_____

9. (**15** points) (Subtyping) This problem considers a language like in lecture containing (1) records with mutable fields, (2) functions, and (3) subtyping. *Even though we learned in lecture that it is a bad idea, the language includes width subtyping, permutation subtyping, and depth subtyping.* The goal of the type system is to prevent accessing a field of a record that does not exist.

   (a) Is the type system sound? (Yes or no; no explanation needed.)

   (b) If possible, fill in the two blanks below such that the program type-checks but, when run, accesses a field of a record that does not exist. If this is impossible, just say "impossible."

```
fun f (x : {f1: {f11 : int}, f2 : int}) =

   -----------------
(* function f ends here; the next 3 lines (including the blank) are at top-level *)
val y : {f1 : {f11: int, f12:int}, f2 : int} = {f1 = {f11 = 0, f12 = 0}, f2 = 0};
f(y);

-------------------
```

For parts (c), (d), (e), and (f), we now consider a change to our type system. Suppose we extend our type system with the idea of "readonly" fields in record types. In addition to all the types we already had, each field in a record type can be marked "readonly" or not. For example, in `{f1 : int, readonly f2 : int, readonly f3 : int}` fields `f2` and `f3` are marked readonly. To type-check `e.f`, we still require `e` has a record type containing a field `f` (marked "readonly" or not) but for `e1.f = e2` we do not allow the `f` field in the type of `e1` to be "readonly.".

Note: *The exam as given had an unfortunate bug. The above parapgraph didn't indicate what Dan intended, namely, that with this change, we now should have depth subtyping for readonly fields and not for the other fields. As a result we gave everyone credit for part (c). The sample solution is what was intended, i.e., it assumes this "correction."*

   (c) Is this new type system sound? (Yes or no; no explanation needed.)

   (d) If possible, fill in the two blanks below such that the program type-checks but, when run, accesses a field of a record that does not exist. If this is impossible, just say "impossible."

```
fun f (x : {readonly f1: {f11 : int}, f2 : int}) =

   -----------------
(* function f ends here; the next 3 lines (including the blank) are at top-level *)
val y : {readonly f1 : {f11: int, f12:int}, f2 : int} = {f1 = {f11 = 0, f12 = 0}, f2 = 0};
f(y);

-------------------
```

   (e) Suppose `t1` and `t2` are almost the same record types. The only difference is for some field `f`, `t1` includes `f : ta` and `t2` includes `readonly f : ta`. Considering the concept of substitutability, should we allow `t1` to be a subtype of `t2`? (Yes or no.)

   (f) If we do allow the subtyping described in part (e), which use of `readonly` in part (d) could we delete and still have the code typecheck (assuming, of course, the code in the blanks typechecks)?

Solution:

(a) No

(b)
```
fun f (x : {f1: {f11 : int}, f2 : int}) =
   x.f1 = {f11 = 0}
   -----------------
(* function f ends here; the next 3 lines (including the blank) are at top-level *)
val y : {f1 : {f11: int, f12:int}, f2 : int} = {f1 = {f11 = 0, f12 = 0}, f2 = 0};
f(y);
y.f1.f12
-------------------
```

(c) Yes

(d) impossible

(e) Yes

(f) The one in the type given for `y`.

Name:_____

Here is an extra page where you can put answers. If you use this page (either side), please write "see also last page" or similar on the page with the question.