Name:_____

CSE 341, Spring 2011, Final Examination
9 June 2011

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. You may also have the sheet of notes you had at the midterm.
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **100 points** total, distributed **unevenly** among **9** questions (most with multiple parts).
- When writing code, style matters, but don't worry too much 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. (a) (**11** points) A binary tree can be defined in Racket as

```
(define-struct tree (val left right))
```

where `val` is any value and `left` and `right` are subtrees or `'()`.
Write a scheme function `sum-tree` that takes one `tree` argument.

- If every node in the tree contains only numbers as values, return the sum of the node values in the tree.
- If the argument is null (`'()`), the result should be 0.
- If any node contains a value that is not a number, or if a subtree is something other than a `tree` struct or null (`'()`) the function should return `#f`.
- Sample solution is about 9 lines; this is just a rough guide.

(b) (**4** points) Give two reasons why a similar function in ML that does not define any additional datatypes other than `tree` would not type-check.

Solution:

(a)
```
(define (sum-tree t)
  (cond [(null? t) 0]
        [(tree? t) (let ([val (tree-val t)]
                         [left (sum-tree (tree-left t))]
                         [right (sum-tree (tree-right t))])
                     (if (and (number? val) (number? left) (number? right))
                         (+ val left right)
                         #f))]
        [#t #f]))
```

(b) Here are three reasons: `sum-tree` can return either a boolean or an integer, but ML functions need one return type. `sum-tree` can have an argument of any type, but ML functions need one argument type. The argument to `sum-tree` can be a tree whose nodes contain values of any type, while the corresponding ML datatype would require nodes that all contain values of the same type.

Name:_____

2. (a) (**5** points) Suppose we execute the following code in standard Scheme.

```
(define a 1)

(define dill (lambda () (begin (set! a (- a 1)) a)))

(define (pickles f) (list (f) (f) (f)))

(define result1 (pickles dill))
```

What is the value of `result1`?

(b) (**5** points) Now suppose we execute the following code, which is similar but not quite the same.

```
(define a 1)

(define dill (lambda () (begin (set! a (- a 1)) a)))

(define (relish f) (list f f f))    ;; this is the major difference

(define result2 (relish dill))
```

What is the value of `result2`?

(c) (**3** points) Explain why `result1` and `result2` are the same or different values.

Solution:

(a) `(0 -1 -2)`

(b) `(#<procedure:dill> #<procedure:dill> #<procedure:dill>)`
(Don't worry about the exact syntax - as long as you said that it was a list of three procedure or closure values with an appropriate example or explanation we gave credit for it.)

(c) The difference is that function `pickles` evaluates its argument three times, while `relish` simply returns a list of unevaluated function closures.

Name:_____

3. (**10** points) Suppose we execute the following code in standard Scheme:

```
(define evil
  (let ([x 2])
    (lambda (y)
      (let ([z 3])
        (begin (set! x (+ 1 x))
               (set! z (+ 1 z))
               (+ x y z))))))

(define ans (list (evil 1) (evil 1) (evil 1)))
```

What is the value of `ans`?

Solution:

```
(8 9 10)
```

Name:_____

4. (**14** points) In this problem, we consider an interpreter for a language RUPL (for really useless programming language) which is like MUPL from Homework 5. However we would like to include a version of let with two bindings instead of one. Programs are built using these structs, among others:

```
(define-struct var  (string))              ;; a variable, e.g., (make-var "foo")
(define-struct int  (num))                 ;; a constant number, e.g., (make-int 17)
(define-struct add  (e1 e2))               ;; add two expressions
(define-struct fun  (nameopt formal body)) ;; a recursive(?) 1-argument function
(define-struct app  (funexp actual))       ;; function application
(define-struct let2 (var1 e1 var2 e2 e3))  ;; a 2-element let expression
```

The RUPL expression `(make-let2 "x" e1 "y" e2 e3)` has *exactly* the same meaning as the Scheme expression `(let ([x e1] [y e2]) e3)`, although, of course, any other names could be used instead of `x` and `y`. Below is the core of the RUPL interpreter. In the space provided, add the code needed to implement this new 2-element `let2` construct.

```
(define (envlookup env str)
  (cond [(null? env) (error "unbound variable during evaluation" str)]
        [(equal? (caar env) str) (cdar env)]
        [#t (envlookup (cdr env) str)]))

(define (eval-prog p)
  (letrec ([f (lambda (env p)
                (cond [(var? p) (envlookup env (var-string p))]
                      [(int? p) p]
                      [(add? p) (let ([v1 (f env (add-e1 p))]
                                      [v2 (f env (add-e2 p))])
                                  (if (and (int? v1) (int? v2))
                                      (make-int (+ (int-num v1) (int-num v2)))
                                      (error "rupl addition applied to non-number")))]
                      [(let2? p)

                      ]
                      ;; remaining rupl code omitted
                      [#t (error "bad rupl expression")]))])
    (f () p)))
```

Solution:

```
[(let2? p)
 (let ([var1 (let2-var1 p)]
       [e1 (f env (let2-e1 p))]
       [var2 (let2-var2 p)]
       [e2 (f env (let2-e2 p))])
   (f (cons (cons var2 e2)
            (cons (cons var1 e1) env))
      (let2-e3 p)))]
```

Note: Both `e1` and `e2` need to be evaluated in the original environment, not one that has been updated with a binding to either variable.

Name:_____

5. (**12** points) Write a Ruby program that reads text from standard input and prints the word that occurs most frequently in the input and how often it occurs. For example, if the input is

```
to be or not to be
to do is to be
do be do be do
```

the expected output is `be 5` . In this case "be" appears 5 times, and all other words appear fewer times. If more than one word is the most frequent you may pick any one of them arbitrarily to print.

To simplify things you can assume that all words are lower-case and are separated by whitespace, with no leading or trailing whitespace on an input line. You can use `gets` to read lines from standard input.

You may find it convenient to use the Ruby string method `split`, which returns an array containing the whitespace-separated words in the string. Example:

```
"one two three".split  =>  ["one", "two", "three"]
```

For full credit you should use Ruby iterators like `each` to process collections like arrays and hashes. Hint: Keep it simple — it's not particularly complex.

Solution:

```
freq = Hash.new(0)

while line = gets
  words = line.split
  words.each{|w| freq[w] += 1}
end

max_freq = 0
max_word = ""
freq.each do |word, n|
  if n > max_freq
    max_freq = n
    max_word = word
  end
end

print max_word, " ", max_freq
```

This solution uses a argument to `Hash.new` to give a default value (0) for each key that has never been stored in the hash. If this is not used, there needs to be a separate case to handle the first time a new word is stored in the table, because there will be no existing count to be incremented in that case.

Name:_____

6. (**10** points) Consider the following very simple Ruby class that maintains a value and allows the value to be changed and retrieved.

```
class Thing
  def initialize val
    @val = val
  end
  def val
    @val
  end
  def val= newval
    @val = newval
  end
end
```

Write a subclass `CountedThing` that works exactly like `Thing` with the following additions:

(a) The subclass counts how many times an assignment to the instance variable `@val` is executed (including the initial assignment when it is created).

(b) The subclass incudes a new method `nchanges` that returns the number of times an assignment to the instance variable has been executed.

The subclass should not duplicate code contained in the original `Thing` class if at all possible. Use `super` as appropriate.

Solution:

```
class CountedThing < Thing
  def initialize val
    super
    @changes = 1
  end
  def val= newval
    super newval
    @changes = @changes + 1
  end
  def nchanges
    @changes
  end
end
```

It is possible to omit or include an argument to `super` in both cases, and the order of the use of `super` and the other statement(s) in the methods can be interchanged without affecting the result.

Name:_____

7. For each of the following definitions, does it type-check in ML? If so, what type does it have? If not, why not?

   (a) (**4** points) `val a = fn g => (fn x => fn y => x) (g 0) (g 7);`

   (b) (**4** points) `val b = fn g => (fn x => fn y => x) (g 0) (g "happy");`

   (c) (**4** points) `val c = fn g => (fn x => fn y => x) (g 0) (g (g 7));`

   Solution:

   (a) `(int -> 'a) -> 'a`

   (b) Fails since `g` cannot have a parameter that is both an `int` and `string`.

   (c) `(int -> int) -> int`

8. (**6** points) Many people assume that programs written in a type-safe language with automatic garbage collection cannot suffer from "space leaks" (or memory leaks, as they are sometimes called). Is this true? If so, why or why not? (Be brief, please)

   Solution:
   No. A space leak can still occur if a program stops using some data while still retaining a reference to it somewhere, for example in a global variable or a container like a list. Even if the program never uses the data again, it is still reachable, so it cannot safely be reclaimed by an automatic garbage collector.

Name:_____

9. (**8** points) This question concerns the typing rules for Java arrays that store references to objects (i.e., not arrays of primitive types like `int` or `double`).

   The original version of Java did not have generics (polymorphic types). In order to allow arrays to be useful to implement things like lists of objects with arbitrary types, array types were defined to be covariant in their element types. If type S is a subtype of T, then the array type $S[\,]$ is a subtype of the array type $T[\,]$. In other words, if $S <: T$, then $S[\,] <: T[\,]$, and vice versa.

   This rule allows us to, for instance, use an array of type `Object[]` to store references to objects of any type, and a variable of type `Object[]` can hold a reference to an array of any other reference type. For backwards compatibility, if nothing else, this typing rule is still used in Java, even though generic types (type parameters) now provide other ways to implement lists and other containers.

   Is this rule sound? If it is, give a convincing (but brief) argument why. If not, give an example that shows why not.

   Solution:
   No, this is not sound. Consider the following example:

```
String[] sa = new String[10];
Object[] oa = sa;
oa[0] = new Integer(13);
```

   The second line is legal because `String[]` is a subtype of `Object[]`. The third line is also allowed by the type checker because any `Object` can be stored in an `Object` array. But the third line is an error because it attempts to store a reference to an `Integer` object into a `String` array.

   [Aside: In actual Java implementations the type checker inserts a runtime check that will generate an `ArrayStoreException` if an error like this occurs. It cannot guarantee at compile-time that such errors are not possible.]