Due Monday, November 13, at the start of class.

Take in a single, 5-hour period, self-timed.

You may refer to any of your notes, handouts, lecture notes, or course readings. You may **not** discuss these questions with anyone else (other than asking the course instructors for clarification), nor may you try to find solutions to these problems elsewhere, e.g., on the web.

You may develop your solutions on-line, including testing our your solutions using the Scheme and SML interpreters.

For short-answer questions, you shouldn't need to write more than 100-200 words or so, and for most questions a few dozen words in telegraphese should be sufficient.

Your solutions should be written electronically in plain ASCII text in one file and turned in via email to `todd@cs` and `lerns@cs`.

100 points total.

1) [4 pts] What are 3 good reasons that Scheme and ML rely on recursion instead of adding some sort of loop construct?

2) [5 pts] Discuss Scheme's `call-with-current-continuation` mechanism in light of Gabriel's The-Right-Thing vs. Worse-Is-Better conflict.

3) [5 pts] Why would it be harder to include dynamic scoping in a language like ML than in a language like Scheme?

4) a) [5 pts] Suppose ML didn't have lexically-nested first-class functions, but only top-level first-class functions (like C and C++ function pointers). What programming idioms (typical programming patterns) could still be used, and what idioms would be lost? How important is this language feature, in your opinion?

   b) [5 pts] Suppose ML didn't have first-class lexically-nested functions, but only lexically-nested functions that could be passed as arguments and invoked but not returned as results or stored in data structures (like functions in Pascal and Modula-3). What programming idioms could still be used, and what idioms would be lost? How important is this language feature, in your opinion?

5) [5 pts] Give a set of one or more ML declarations appropriate to represent Scheme values, as might be included in a Scheme interpreter written in ML. You should be able to represent Scheme integers, booleans, nil, symbols, pairs, primitive functions (which should be represented using ML functions from lists of argument Scheme values to single result Scheme values), and user-defined, lexically-scoped functions. Environments should be of type `Identifier->Value`. You do not need to write any functions to build or manipulate Scheme values.

6) [4 pts] Why are CoreML and real ML different?

7) a) [4 pts] Translate the following CoreML top-level declaration into the most natural corresponding ML datatype declaration.

```
type Tree =
  rec t = branded Tr [Empty:unit, Leaf:int, Node:t*t]
```

b) [4 pts] What else comes with the ML datatype you gave that isn't in the CoreML declaration?

c) [4 pts] Give the CoreML type corresponding to the following ML datatype declaration:

```
datatype Bar = Frob
             | Qux of Bar * int * Bar
             | Quibble of {n:string, b:Bar}
```

8) [10 pts] Define the right-associative, curried `reduce` function over `intLists` in real ML and in CoreML. Define an explicit `intList` datatype for your ML version, independently of ML's built-in list type. The CoreML `reduce` should return an `intList` (since we don't have polymorphic types yet). Show how to define the curried `map` function over `intList`'s using your `reduce` function, in both languages. Remember to include explicit types, explicit folds & unfolds, and explicit `rec`'s where necessary in your CoreML version.

9) Say you're designing a language, as we did in class with the CoreML language. You're confronted with the following issues:

a) [5 pts] You are thinking about tuples and records. Which would you build in to your language, or would you include both? Why?

b) [5 pts] You are thinking about records and unions. Which would you build in to your language, or would you include both? Why?

c) [10 pts] You'd like to add arrays (with dynamically determined lengths, as in Java and ML but not C) to CoreML. Define suitable type form(s), expression form(s), and pattern-matching form(s) for arrays, analogously to the way that we handled functions, tuples, records, unions, etc. Give an informal explanation of the meaning and use of your new constructs. Explain and justify your language design decisions, and analyze the strengths and weaknesses of your design.

10) a) [20 pts] Implement termination-style exception handling in Scheme. In particular, implement the following two functions:

`(raise expr)`: Evaluates `expr` to a value `v`, and then raises `v` as an exception. This locates the catch-block function from the closest dynamically enclosing `try` invocation and invokes the catch-block, passing it `v` as its argument. If there is no dynamically enclosing `try` invocation, the `error` procedure is invoked to report an unhanded exception. `raise` does not return to its caller.

`(try body catch-block)`: Evaluates `body` (a function of no arguments) in a context where the current catch-block is `catch-block` (a function of one argument). The result of evaluating `body` is returned as the result of the `try` invocation, unless `catch-block` is invoked by a `raise` during body's evaluation, in which case the result of evaluating `catch-block` is returned as the result of the `try` invocation (cutting back the call stack in the process). The evaluation of `catch-block` is done in a context where the nearest catch-block is not itself, but rather that of the next most enclosing `try` invocation, so that a `raise` invoked within a catch block allows exceptions to be reraised, propagated to the next catch-block up the stack.

Examples of use:

```
(define (inverse x)
  (if (= x 0) (raise 'div-by-zero) (/ 1 x)))

(define (mul x y)
  (if (>= (* x y) 1000000000) (raise 'overflow) (* x y)))

(define (foo y)
  (try
    ;; the body of the try block:
    (lambda() (mul (inverse y) 10000))
    ;; the catch handler:
    (lambda(exn)
      (if (equal? exn 'overflow)
            ;; handle the overflow, returning 10000:
            10000
          ;; not caught; reraise the exn:
          (raise exn)))))

(define (frob w)
  (if (< w 0) (raise 'negative) w))
```

```
(define (bar z)
  (try
    ;; the body of the try block:
    (lambda() (frob (foo z)))
    ;; the catch handler:
    (lambda(exn)
      (if (equal? exn 'div-by-zero)
          ;; handle the div-by-zero exception:
          'yikes
        ;; don't handle this exn; reraise:
        (raise exn)))))

;; no exceptions raised:
(bar 10)        ==> 1000

;; raises & handles overflow exn:
(bar .00000001) ==> 10000

;; raises, handles, & reraises div-by-zero exn:
(bar 0)         ==> yikes

;; raises negative exn, but doesn't handle it:
(bar -10)      ==> error: unhandled exception negative
```

You may use a global variable and side-effects to store any state associated with catch blocks. You may find the following helper functions useful in your solution:

```
;; helper functions to manage a stack of handlers

(define **-catch-handlers-** ())

(define (push-handler! handler)
  (set! **-catch-handlers-**
    (cons handler **-catch-handlers-**)))

(define (top-handler)
  (car **-catch-handlers-**))

(define (pop-handler!)
  (set! **-catch-handlers-** (cdr **-catch-handlers-**)))


;; define the default catch-handler
(push-handler!
  (lambda(exn) (error "unhandled exception: " exn)))
```

b) [5 pts] Imagine that `try` and `raise` are ML functions. What would their types be? (You can use `any` wherever you need to use a type that can handle any value, and `none` as the result of a function that does not return to its caller.) For 2 pts extra credit, use ML's polymorphic types (`'a`, `'b`, `'c`, etc.) in place of `any` to express the most general type of these functions.