Name: ***Solutions***

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?

   want to avoid side-effects; traditional iteration is useful only for its side-effects

   recursion is more general than iteration

   tail-recursion optimization makes iterative-style recursion as efficient as iteration

   prominence of recursive datatypes (lists, trees, etc.) makes recursive functions more natural

   keep the language simpler

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

   `call/cc` is an excellent example of The Right Thing Gone Berzerk, since it is a single, powerful construct that subsumes all non-trivial non-local control constructs. However, it's really hard to implement efficiently, and it's usually much more powerful than is needed for any particular application. A Worse-Is-Better language might include several different control constructs, e.g. goto, exceptions, threads, each with restricted functionality and relatively straightforward implementation. Users probably would be happier, too.

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

   Static typechecking would be much more difficult under dynamic scoping.

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?

You could still do all of the standard higher-order list iteration functions like map and filter, but the functions sent as parameters to these functions must have no free variables, thereby greatly reducing the flexibility of this idiom. Currying would become impossible, as would many other applications of returning functions from functions. Idioms like the menu data structure of functions and infinite streams via functions also become hard(er).

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?

You could still do all of the standard higher-order list iteration functions like map and filter (including the ability to have free variables from the lexical scope in the function parameters) but you lose the ability to make use of currying to partially evaluate a function. Idioms like the menu data structure of functions and infinite streams via functions also become impossible.

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.

```
datatype Value =
     Integer of int
   | Boolean of bool
   | Nil
   | Symbol of Identifier
   | Pair of Value * Value
   | PrimFun of Value list -> Value
   | UserFun of {formals:Identifier list, body:Value,
                 env:Identifier->Value}
withtype Identifier = string
```

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

CoreML strives for simplicity and regularity/orthogonality, to highlight the primitive notions underlying ML-like languages, and to make reasoning about properties of the language simpler. Some features are available in CoreML that aren't available directly in ML, such as unions, recursion, and branding separately from each other.

ML is targeted at users who want to get work done, so it builds in some conveniences and common idioms that aren't strictly necessary.

ML also includes features we haven't gotten to yet in the incremental design of core ML, like polymorphism and modules. But we'll get there....

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]
```

Answer:

```
datatype Tree = Empty | Leaf of int | Node of Tree*Tree
```

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

The constructor values and functions.

Implicit fold/unfold & brand/unbrand expressions and patterns.

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}
```

Answer:

```
type Bar =
   rec t =
      branded Bar
         [Frob:unit, Qux:t*int*t, Quibble:{n:string, b:t}]
```

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.

In real ML:

```
datatype intList = Nil | Pair of int * intList
```

```
fun reduce f b Nil = b
  | reduce f b (Pair(x,l)) = f(x, reduce f b l)

fun map f = reduce (fn(x,r) => Pair(f x, r)) Nil
```

In CoreML:

```
type intList = (rec t = [Nil:unit, Pair:int * t])

val vNil:intList = fold [Nil=()] as intList

val fPair:int*intList->intList =
  lambda(x:int, l:intList). fold [Pair=(x,l)] as intList

val reduce:(int*intList->intList)->intList->intList
           ->intList =
  rec reduce:(int*intList->intList)->intList->intList
             ->intList =
    lambda(f:int*intList->intList).
      lambda(b:intList).
        lambda(l:intList).
          case (unfold l) of
            [Nil=()] => b
          | [Pair=(x:int,l':intList)] =>
              f(x, reduce f b l')

val map:(int->int)->intList->intList =
  lambda(f:int->int).
    reduce (lambda(x:int,r:intList). fPair(f x, r)) vNil
```

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?

I'd include records, since tuples can be viewed as syntactic sugar for records with integer labels. I only include one, to keep the core language small.

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

I'd include both, since they're incomparable in expressiveness. Languages need both notions.

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.

```
tau ::= ... | <> tau
e   ::= ... | <e, ..., e> | <e#e> | e!e := e | e!e | #e
p   ::= ... | <p, ..., p>
```

An array type is written `<> tau`, where `tau` is the type of its elements. An array can be created (1) with the `<e1, ..., eN>` form, which creates an `N`-long array whose elements are the results of evaluating expressions `e1` through `eN`, or (2) with the `<e1#e2>` form, which creates an array whose length is `e1` and all of whose elements are initialized to `e2`, or (3) with the `e1!e2 := e3` form, which takes an array `e1`, an index `e2`, and a new element value `e3`, and returns a new array whose elements are the same as `e1` except that the `e2`'th element is updated to be `e3` (or fails with a run-time error if `e2` is out of bounds). Arrays can be accessed with the `e1!e2` form, where `e1` evaluates to an array and `e2` to an integer, and returns the `e2`'th element in `e1` (or fails with a run-time error if `e2` is out of bounds). The `#e` form returns the length of the array `e`. The pattern `<p1,...,pN>` matches arrays of length `N` each of whose element values matches the corresponding patterns `p1..pN`.

These features seem complete to manipulate arrays w/o using side-effects. There are more constructs than are strictly required, but each of the constructs seems pretty useful. Additional operations to build arrays whose elements are determined via some sort of map-like interface (i.e., an argument function is called to compute the value of each array position) would be nice, too, as would operations to construct arrays by concatenating arrays, or by selecting out subranges of existing arrays; in real langauges, these are typically provided by libraries rather than as built-in expression forms. I don't know of a way to have pattern forms that correspond to the other introduction forms; this aspect of this design seems asymmetrical.

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)))
```

Answer:

```
(define (raise exn) ((top-handler) exn))

(define (try body catch-block)
  (call-with-current-continuation
    (lambda(exit)
      ;; invoke exit to break out of the try block...

      ;; push a new catch-handler on the stack
      (push-handler!
        (lambda(exn)
          ;; catch block was invoked...

          ;; pop this catch-block, in case there are
          ;; any raises in the catch block
          (pop-handler!)

          ;; run the catch-block, and exit the try block
          (exit (catch-block exn))))

      ;; evaluate body in the context of this catch handler
      (let ((result (body)))
        ;; body returned normally...
```

```
;; pop the handler
(pop-handler!)

;; return from the try block
result))))
```

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.

Using any:

```
raise: any->none

try: (unit->any) * (any->any) -> any
```

Using polymorphic types:

```
raise: 'a->none

try: (unit->'a) * ('b->'a) -> 'a
```