CSE 341 Midterm

(Open Book, Open Notes – 40 points total)

Winter 2001

Name: ______________________________

Section: ______________________________

1. (5 points) Write a Scheme function `how-many` that takes a list and an element, `e`, and returns the number of occurrences of `e` in the list. For example:

```
(how-many '(a b a) 'a) => 2
```

```
(define (how-many lst elt)
   (cond ((null? lst) 0)
         ((equal? (car lst) elt) (+ 1 (how-many (cdr lst) elt)))
         (else (how-many (cdr lst) elt))
   )
)
```

or

```
(define (how-many lst elt)
   (sum (map (lambda (x) (if (equal? x elt) 1 0)) lst))
)
```

2. (5 points) Write `how_many` in Miranda.

```
how_many [] elt = 0
how_many (elt:xs) elt = 1 + how_many xs elt
how_many (x:xs) elt = how_many xs elt
```

or

```
how_many list elt = sum map same list
                    where
                       same x = 1, if x = elt
                              = 0, otherwise
```

3. (10 points) Given the following Miranda code:

```
maybe * ::= Nothing | Something *

my_sqrt x = Something (sqrt x), if x >= 0
          = Nothing, otherwise

shuffle (x:xs) (y:ys) = x : y : shuffle ys xs
shuffle xs ys = xs ++ ys

things [] = []
things (Nothing:rest) = things rest
things (Something x:rest) = x : things rest

mystery x [] = x
mystery 1 (y:ys) = y
mystery x (y:ys) = mystery (x-1) ys
```

What are the results (type or value) of the following expressions. If there is an error, indicate what the error is (2 points each):

(a) `my_sqrt ::`
`num->maybe num`

(b) `shuffle ::`
`[*]->[*]->[*]`

(c) `map shuffle ::`
`[[*]]->[[*]->[*]]`

(d) `things ::`
`[maybe *]->[*]`

(e) `shuffle [Nothing] ::`
`[maybe *]->[maybe *]`

(f) `shuffle [1,2] [3,4]`
`[1,3,4,2]`

(g) `map my_sqrt ::`
`[num]->[maybe num]`

(h) `map my_sqrt [0,1,-1,4,-4]`
`[Something 0.0,Something 1.0,Nothing,Something 2.0,Nothing]`

(i) `things (map my_sqrt [0,1,-1,4,-4])`
`[0.0,1.0,2.0]`

(j) `mystery 4 "abc"`
`type error in expression, cannot unify [char] with [num]`

4. (5 points) We implemented `foldl` in Miranda and in Scheme:

```
foldl op id [] = id
foldl op id (x:xs) = foldl op (op id x) xs

(define (foldl op id lst)
   (if (null? lst)
       id
       (foldl op (op id (car lst)) (cdr lst))
   )
)
```

The Scheme implementation is tail-recursive and takes constant space to run. The Miranda implementation looks tail recursive it actually takes linear space to run. Because of this, the `foldl` in the standard environment is implemented differently (so as to be tail-recursive and take only constant space to run). Why does our Miranda implementation above take linear space? Justify your answer.

Why? The Miranda `foldl` implementation takes linear space because Miranda is lazily evaluated.

Justification: Each recursive call is passed `(op id x)` as its second argument. Because Miranda is lazy, this does not get evaluated at the time of the call. Instead a delayed computation is passed. Each time we call `foldl` we make a new delayed computation (which references the old delayed computation). None of these are actually evaluated until we reach the base case and return `id`. It then will evaluate all of these nested delayed computations. There are a linear number of delayed computations stored. Thus, it takes linear space.

5. (5 points) Given the following code with `foldl` defined as usual (See Question 4).

```
(define (add x y) (+ x y))

(let ((+ *))
     (list (+ 2 3)
           (foldl add 10 '(1 2 3))
     )
)
```

   (a) If this code is evaluated in Scheme, what is the result?
   `(6 16)`

   (b) If this code is evaluated in a language that is identical to Scheme except that it is dynamically scoped, what is the result?
   `(6 60)`

   (c) Which of the following languages use dynamic scoping for variables: C++, Scheme, Miranda?
   None of them are dynamically scoped.

6. (5 points) We defined a function `fib` in Miranda to return the ith Fibonacci number as follows:

```
add_adjacent (x:y:xs) = (x + y) : add_adjacent (y:xs)

fibs = 0 : 1 : add_adjacent fibs

fib = (!) fibs
```

   (a) What is the value of `"hello world" !  7`?
   `'o'`

   (b) How many times is `+` called when evaluating `fib 4`?
   3

   (c) How many times is `+` called when evaluating `fib k` for some integer `k` (Do not use Big-Oh notation)?
   $k - 1$

7. (5 points) The `eval` and `apply` functions for the meta-circular interpreter were defined in class to be:

```
1    (define (eval expr env)
2       (cond ((symbol? expr) (lookup env expr))
3             ((literal? expr) expr)
4             ((special-form? expr) (eval-special-form expr env))
5             ((pair? expr)
6                 (let ((exp-list (map (lambda (arg) (eval arg env)) expr)))
7                    (apply (car exp-list) (cdr exp-list))
8                 )
9             )
10      )
11   )
12
13   (define (apply proc args)
14      (let ((new-frame (map list (proc-params proc) args)))
15           (eval (proc-body proc) (cons new-frame (proc-env proc)))
16      )
17   )
```

Here `proc-body`, `proc-params`, `proc-env` are selectors that give the body (source code), the formal parameter list, and the defining environment of the procedure, respectively.

Modify this above code so that our interpreter uses dynamic scoping. You may either neatly make your changes directly to the code above or indicate below which lines you would change and what you would change them to.

```
7         (apply (car exp-list) (cdr exp-list) env)
13        (define (apply proc args env)
15        (eval (proc-body proc) (cons new-grame env))
```