CSE 341 - Programming Languages
Midterm - Autumn 2008 - Sample Solution

90 points total

1. (10 points) Suppose that we have the following definition of the `member` function in Haskell:

```
member x [] = False
member x (y:ys) | x==y      = True
                | otherwise = member x ys
```

The following are correct type declarations for `member`:

```
member :: (Ord a) => a -> [a] -> Bool

member :: (Eq a) => a -> [a] -> Bool

member :: (Eq a) => [a] -> [[a]] -> Bool

member :: Bool -> [Bool] -> Bool
```

(Note that this type:
`member :: (Integer -> Integer) ->  [Integer -> Integer] -> Bool`
isn't a correct type for `member`, since function types aren't in the `Eq` class (i.e. you can't test whether two functions are equal.)

This is the most general type for `member`:

```
member :: (Eq a) => a -> [a] -> Bool
```

2. (20 points) Suppose the following Haskell program has been read in.

```
my_const c x = c

append [] ys = ys
append (x:xs) ys = x : append xs ys

map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys

the_great_appender = do
    x <- readLn
    return (append x ["squid", "octopus"])
```

What is the *value* of each of the following expressions? (Some may give a type error; if so say that.)

(a) `my_const 3 "squid"  =>  3`

(b) `append [1,2,3] [True]  =>  type error`

(c) `append "bull" "dog"  =>  "bulldog"`

(d)
```
map2 my_const [1, 2, 3] ["squid", "clam", "octopus"]
     =>  [1, 2, 3]
```

What is the *type* of each of the following expressions? Some of them may give type errors — if so, say that. (Hint for 2d: use the most general type for +, which is `(Num a) => a -> a -> a`.)

(a) `:t append []  => [a] -> [a]`

(b) `:t map2  =>  (a->b->c) -> [a] -> [b] -> [c]`

(c) `:t map2 my_const  => [a] -> [b] -> [a]`

(d) `:t map2 (+)  =>  (Num t) => [t] -> [t] -> [t]`

(e) `:t the_great_appender  => IO [[Char]]`

(f) `:t the_great_appender >>= \n -> putStrLn (show n)  =>  IO ()`

3. (5 points) What are the first 5 elements in the following list?

```
mystery = 1 : map (\n -> 2*n+1) mystery
```

`[1,3,7,15,31]`

4. (9 points) Consider the following example in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k*2;
    print(k);
    end;
n := 4;
p(n);
print(n);
end;
```

(a) What is the output when k is passed by value? **8 5**

(b) What is the output when k is passed by value result? **8 8**

(c) What is the output when k is passed by reference? **10 10**

5. (5 points) What is the value of the following Scheme expression?

```
(let ((x 1)
      (y 5))
  (let ((x (+ y 2))
        (y (* x 3))
        (s (lambda (z) (* x y z))))
    (+ x y (s 10))))

=>  60
```

6. (15 points) One of the extensions to the Scheme metacircular interpreter was to include `and` in the language. This was implemented as a special form. As the exercise noted, an alternative way to implement this is as a derived expression.

   (a) Write a Scheme function `and->combination` that takes a list representing an `and` expression and that returns another list representing an equivalent expression (i.e., one that evaluates to the same thing), but that either doesn't use `and` or that has a simpler `and` expression (i.e., a recursive case). You can assume the input expression is actually an `and` expression and is syntactically correct.

   There are several approaches. The most elegant is to reduce `and` to an `if` (and another `and` if there is more than one expression in the `and`).

```
(define (and->combination exp)
  (cond ((null? (cdr exp)) #t)
        ((null? (cddr exp)) (cadr exp))
        (else (list 'if (cadr exp)
                    (cons 'and (cddr exp))
                    #f))))
```

   Another approach is to reduce it immediately to nested if statements:

```
(define (and->combination exp)
  (cond ((null? (cdr exp)) #t)
        ((null? (cddr exp)) (cadr exp))
        (else (list 'if (cadr exp)
                    (and->combination (cons 'and (cddr exp)))
                    #f))))
```

(b) Using your definition, what is the value of the following expressions?

Using the first solution:

- `(and->combination '(and)) => #t`
- `(and->combination '(and (mystery x))) => (mystery x)`
- `(and->combination '(and (= x 5) (< y 10) (member x xs)))`
  `=> (if (= x 5) (and (< y 10) (member x xs)) #f)`

Using the second solution:

- `(and->combination '(and)) => #t`
- `(and->combination '(and (mystery x))) => (mystery x)`
- `(and->combination '(and (= x 5) (< y 10) (member x xs)))`
  `=> (if (= x 5) (if (< y 10) (member x xs) #f) #f)`

Hints: The function `and->combination` is a way to implement `and` as a derived expression, just like `let->combination` was a way to implement `let` as a derived expression.

Your equivalent expression should probably use `if` or `cond`.

Finally, recall the exact definition of `and`:

> The expressions are evaluated from left to right. If any expression evaluates to `#f`, `#f` is returned; any remaining expressions are not evaluated. If all the expressions evaluate to true values (i.e., anything other than `#f`), the value of the last expression is returned. If there are no expressions then true is returned.

7. (10 points) Suppose that the Glasgow Haskell Compiler people release a new version of Haskell, Haskell Flat, that passes parameters using call-by-value. (If Microsoft can have C#, Scotland can have Haskell Flat.)

   (a) Are there any Haskell programs that used to type check correctly that would no longer type check in Haskell Flat? If so give an example.

   No - changing to call-by-value doesn't affect types or type checking.

   (b) Are there any Haskell programs that used to run, but that in Haskell Flat would fail due to a runtime error of some sort? If so give an example.

   Yes - a function call with a parameter that was never evaluated in Haskell but that gives an error in Haskell Flat would stop running. Example:

   ```
   const 3 (tail [])
   ```

   (c) Are there any Haskell programs that run and terminate successfully both in the old Haskell and in Haskell Flat, but that give a different answer? If so give an example.

   No. The only difference is that some programs that run in Haskell either terminate with an error in Haskell Flat or never terminate; but if they terminate successfully they give the same answer.

8. (6 points) Tacky but easy-to-grade true/false questions!

   (a) Because Scheme is type safe, it is also statically type checked. False.

   (b) A Haskell program is statically typed if the programmer includes a type declaration for all functions; otherwise it is dynamically typed. False

   (c) Any recursive function `f` can be rewritten as a tail-recursive version that uses a constant amount of space no matter how many recursive calls there are of `f`. False.

9. (10 points) Consider the bank account example that illustrates simulating objects in Scheme. Suppose that we run this in the metacircular interpreter, and add a `debug-info` command, as follows:

```
(define (make-account)
   (let ((my-balance 0))

      (define (balance)
         my-balance)

     (define (withdraw amount)
        (if (>= my-balance amount)
           (begin (set! my-balance (- my-balance amount))
                  my-balance)
           "Insufficient funds"))

     (define (deposit amount)
        (debug-info)   ;; NOTE ADDED EXPRESSION HERE
        (set! my-balance (+ my-balance amount))
        my-balance)

     ;; the dispatching function -- decide what to do with the request
     (define (dispatch m)
        (cond ((eq? m 'balance) balance)
              ((eq? m 'withdraw) withdraw)
              ((eq? m 'deposit) deposit)
              (else (error "Unknown request -- MAKE-ACCOUNT"  m))))

      dispatch))
```

   Suppose that we define a new account: `(define acct1 (make-account))`.

   (a) What does `((acct1 'deposit) 100)` print? (Hint: this should print the `debug-info` information, and also the result of the deposit.)

```
((acct1 'deposit) 100) =>

debug-info here.  Current environment is as follows:
***** new frame *****
amount: 100
***** new frame *****
dispatch: (compound-procedure (m) ((cond ((eq? m 'balance)...)))
           <procedure-env>)
deposit: (compound-procedure (amount) ((debug-info) (set! ...))
           <procedure-env>)
withdraw: (compound-procedure (amount) ((if ...)) <procedure-env>)
balance: (compound-procedure () (my-balance) <procedure-env>)
my-balance: 0
***** new frame *****
***** new frame *****
acct1: (compound-procedure (m) ((cond ((eq? m 'balance) ...)))
        <procedure-env>)
make-account: (compound-procedure () ((let ((my-balance 0)) ...
               dispatch))
         <procedure-env>)
car: (primitive #<primitive:car>)
cdr: (primitive #<primitive:cdr>)
cons: (primitive #<primitive:cons>)
null?: (primitive #<primitive:null?>)
+: (primitive #<primitive:+>)
-: (primitive #<primitive:->)
*: (primitive #<primitive:*>)
=: (primitive #<primitive:=>)
eq?: (primitive #<primitive:eq?>)
inspect-it: (primitive #<procedure:inspect-it>)
***** display environment done *****

100
```

(b) What does `(inspect-it (acct1 'balance))` print? (Hint: look closely at how much is evauated here. This doesn't return the balance of 100!)

```
(compound-procedure () (my-balance))
function environment:
***** new frame *****
dispatch: (compound-procedure (m) ((cond ((eq? m 'balance) balance)...
          <procedure-env>)
deposit: (compound-procedure (amount) ((debug-info) (set! ...))
```

```
          <procedure-env>)
withdraw: (compound-procedure (amount) ((if ...)) <procedure-env>)
balance: (compound-procedure () (my-balance) <procedure-env>)
my-balance: 100
***** new frame *****
***** new frame *****
acct1: (compound-procedure (m) ((cond ((eq? m 'balance) ...)))
        <procedure-env>)
make-account: (compound-procedure () ((let ((my-balance 0)) ...
               dispatch))
         <procedure-env>)
car: (primitive #<primitive:car>)
cdr: (primitive #<primitive:cdr>)
cons: (primitive #<primitive:cons>)
null?: (primitive #<primitive:null?>)
+: (primitive #<primitive:+>)
-: (primitive #<primitive:->)
*: (primitive #<primitive:*>)
=: (primitive #<primitive:=>)
eq?: (primitive #<primitive:eq?>)
inspect-it: (primitive #<procedure:inspect-it>)
***** display environment done *****
```

You don't need to get the exact format correct for `debug-info` and `inspect-it`, but be sure it includes the different frames in the environment, clearly separated. For the global environment just give a few of the variables, including any new variables bound by this program. For other frames include all the variables. Write your answers on the back of this page.