Midterm Solution, Cmput 325, Section A2
October, 1994

INSTRUCTIONS: The time for this test is 80 minutes. No references should be used. Write all answers in these pages. The mark total for this exam is 100.

1. (25 points)

1.1 Reduce the following λ-expressions to their normal forms. Show all the reduction steps.

(a) $(\lambda xy \mid y(xx))\ (\lambda x \mid x)\ (\lambda z \mid z)$

$$\begin{aligned} &\rightarrow (\lambda y \mid y((\lambda x \mid x)(\lambda x \mid x)))(\lambda z \mid z) \\ &\rightarrow (\lambda y \mid y(\lambda x \mid x))(\lambda z \mid z) \\ &\rightarrow (\lambda z \mid z)(\lambda x \mid x) \\ &\rightarrow (\lambda x \mid x) \end{aligned}$$

(b) $(\lambda xy \mid xb)\ (\lambda zw \mid z)\ a$

$$\begin{aligned} &\rightarrow (\lambda y \mid (\lambda zw \mid z)b)\ a \\ &\rightarrow (\lambda zw \mid z)b \\ &\rightarrow (\lambda w \mid b) \end{aligned}$$

(c) $(\lambda xy \mid xy)\ (\lambda z \mid zy)\ a\ \ b$

$$\begin{aligned} &\rightarrow (\lambda w \mid (\lambda z \mid zy)w)\ a\ b \\ &\rightarrow (\lambda z \mid zy)\ a\ b \\ &\rightarrow ayb \end{aligned}$$

1.2 Consider reducing the following λ-expression

$$(\lambda xy \mid (\lambda y \mid xyy)y)\ (\lambda x \mid xx)\ a$$

to its normal form. Show (a) the sequence of normal order reductions and (b) the sequence of applicative order reductions.

(a)

$$\begin{aligned} &\rightarrow (\lambda y \mid (\lambda y \mid (\lambda x \mid xx)yy)y)\ a \\ &\rightarrow (\lambda y \mid (\lambda x \mid xx)yy)a \\ &\rightarrow (\lambda x \mid xx)aa \\ &\rightarrow aaa \end{aligned}$$

(b)

$$\begin{aligned} &\rightarrow (\lambda xy \mid xyy)\ (\lambda x \mid xx)\ a \\ &\rightarrow (\lambda y \mid (\lambda x \mid xx)yy)\ a \\ &\rightarrow (\lambda y \mid yyy)a \\ &\rightarrow aaa \end{aligned}$$

NOTE: It is wrong to reduce $(\lambda xy \mid xyy)\,(\lambda x \mid xx)\; a$ to $(\lambda xy \mid xyy)\; a\; a$. This is of the pattern ABC which equals $(AB)C$ but not $A(BC)$.

2. (12 points) Consider the following Lisp definitions:

```
(defun f (L)
       (cond ((null L) nil)
             (t (cons (cons (cadar L) (caar L)) (f (cdr L))))
       )
)

(defun g (L)
       (cond ((null L) nil)
             (t (append (g (cdr L))
                        (cons (cons (cadar L)
                                    (cons (caar L) nil))
                              nil
                        )
                )
             )
       )
)
```

What will be returned after executing the following expressions?

```
          (f '((a 1) (b 2) (c 3)))
answer: ((1 . a) (2 . b) (3 . c))

          (g '((a 1) (b 2) (c 3)))
answer: ((3 c) (2 b) (1 a))

          (f (g '((a 1) (b 2) (c 3))))
answer: ((c . 3) (b . 2) (a . 1))
```

3. (22 points) In solving the following problems, you may use any Lisp primitive or built-in functions that have been allowed in the assignments.

(a) (7 points) Define a function MakeSet such that given a list L of arbitrary s-expressions, (MakeSet L) removes the duplicates, i.e., it returns a list that contains each element in L exactly once. E.g., (MakeSet '(a b a (a) (d) e (d))) $\Rightarrow$ any list that contains exactly five elements, a, b, (a), (d), and e, in any order.

```
(defun MakeSet (S)
    (cond ((null S) nil)
          ((xmember (car S) (cdr S)) (MakeSet (cdr S)))
          (t (cons (car S) (MakeSet (cdr S))))
    )
)

where xmember is defined in assignment 1.
```

(b) (3 points) Can we define the function MakeSet above using the higher order function MAPCAR? If yes, define it. Otherwise explain why.

No. MAPCAR applies to a unary function to each element of a given list.

(c) (12 points) Define a function SetEqual such that given two lists S1 and S2 (of arbitrary s-expressions), (SetEqual S1 S2) returns T if S1 and S2 represent the same set, otherwise it returns NIL. E.g., (SetEqual '(a (b)) '((b) a)) $\Rightarrow$ T. Assume that S1 and S2 do not have repeated elements.

```
(defun SetEqual (S1 S2)
    (and (subset S1 S2) (subset S2 S1))
)

(defun subset (S1 S2)
    (cond ((null S1) T)
          ((xmember (car S1) S2) (subset (cdr S1) S2))
          (t nil)
    )
)
```

4. (16 points)
(a) (4 points) Give an s-expression that is represented internally as follows.

```
          -----    -----      -----
       -> | | | -> | | | ->   | | | -> nil
          -----    -----      -----
            |        |          |
            V        V          V
            a                   d
                   -----
                   | | | -> c
                   -----
                     |
                     V
                     b
```

```
answer: (a (b . c) d), or equivalently, any s-expression
in  part (b).
```

(b) (4 points) Give two additional s-expressions that are also represented internally by the same structure above.

```
(a . ((b . c) d))
(a . ((b . c) d . nil))
(a . ((b . c) . (d . nil)))
```

(c) (8 points) Show how the following s-expressions are represented internally.

```
(i)     (a b (c d))
(ii)    ((a (b c)) . d)
```

Almost all of the students did correctly.

4. (25 points) This question is about the context-based interpreter. You will be asked about what is the context when an expression is evaluated. To answer these questions, sometimes you need to go through a number of evaluation steps mentally. However, you don't need to show these steps.

For the purpose of this question, you only need to show a context using the notation `{n1 -> v1, ..., nk -> vk}`. If any `vi` is a closure, use the notation `[fn, CT]` where `fn` should be a lambda function and `CT` a context. The starting context is always assumed empty.

(a) (5 points) Consider

```
(let  (x . 1)  (y . 2)  ((lambda (z) (+ x (+ y z))) 3))
```

Show the context when `(+ x (+ y z))` is evaluated.

```
in context { z->3, y->2, x->1}
```

(b) (8 points) Consider

```
((lambda (x) (+ x 1))  ((lambda (y) (+ y 2))  3))
```

Show the context associated with each eval.

```
(i) eval (+ y 2)  in {y->3}

(ii) eval (+ x 1) in {x->5}
```

(c) (12 points) Consider

```
((lambda (x y) (y x))  (lambda (z) (+ z 1)) (lambda (w) (w 2)))
```

Show the context associated with each eval.

```
(i) eval (y x)
   CT0 = empty
   eval (y x) in CT1 = {x->[(lambda (z) (+ z 1)), CT0],
                        y->[(lambda (w) (w 2)), CT0]} U CT0

(ii) eval (w 2)
 /* the steps leading to eval (w 2):
      eval (y x) in CT1
   => eval y in CT1 and eval x in CT1
   => get closures of x and y, bind formal parameter w
      with actual, extend the current context (the one
      in which the body of the function should be evaluated,
      and continue eval the body */

     eval (w 2) in CT2 = {w->[(lambda (z) (+ z 1)), CT0]} U CT0

(iii) eval (+ z 1)
   /* eval eval (w 2) in CT2
   => eval w in CT2 gets the closure, formal is bound to actual,
      extend the context (in which the body should be eval'd,
      and eval the body next */

      eval (+ z 1) in {z->2} U CT0
```