CS61A FALL 2008 — FINAL REVIEW SESSION

THE STAFF

1. Summary of Topics

The following topics will comprise the bulk of the final:

(1) Functions and Recursion
(2) `word, sentence, first, butfirst ...`
(3) Higher-Order Functions
(4) Order of Growth
(5) Recursive vs. Iterative *Process*
(6) Basics on Pairs and Lists
(7) Data Abstraction
(8) Data-directed Programming, Message-Passing Programming
(9) Object-Oriented Programming, above and below the line
(10) Environmental Diagrams and states
(11) List mutation
(12) Vector mutation
(13) Concurrency and Serialization
(14) Streams
(15) Mapreduce
(16) Metacircular Evaluator Implementation
(17) Lazy Evaluator Concept
(18) Analyzing Evaluator Concept
(19) Non-deterministic Programming
(20) Logic Programming

2. Problems

Exercise 1. Write a higher-order function, `all-pairs`, that takes two lists and return a list of all possible pairs of elements from the two argument lists. DO NOT use recursion. For example:

```
> (all-pairs '(1 2) '(2 3 4 5))
 ((1 2) (1 3) (1 4) (1 5) (2 2) (2 3) (2 4) (2 5))
```

Solutions:

```
(define (all-pairs ls1 ls2)
  (accumulate append (map (lambda (x) (map (lambda (y) (list x y)) ls1)) ls 2)))
```

Exercise 2. What is the running time of `all-pairs` in term of length of the first list m and length of the second list n? True or False, it is possible to implement `all-pairs` in linear time if we allow recursion

*Solutions:*Runtime is $\Theta(mn)$. It cannot be done in linear time because output is a list of size mn.

Exercise 3. What will Scheme print?

```
# 1
> (accumulate cons '(1 2 3) '(4 5 6))

(4 5 6 1 2 3)

# 2
> (equal? ((lambda (x) (x x x)) 7) '(7 7 7))
```

Date: December 17, 2008.

```
ERROR

# 3
> (define ls (append! (cons 1 (list 1 2)) nil))
> ls

(1 1 2)

# 4
> (and (or '(#f 1 2)) (or '(#f #f)))

(#f #f)

#5

> (define x (cons 1 2))
> (set-cdr! x x)
> (define y x)
> (set! x 1)
> y

infinite loop
```

Exercise 4. Short Answer:

(1) Suppose `x` has already been defined to be 0. What is the difference between calling `(define x 10)` and `(set! x 10)`
(2) T/F, applicative order will *always* run faster than normal order
(3) T/F, we can use `append!` to implement `set-cdr!`
(4) Which is faster? Finding the last element of a list or of a vector?
(5) Which is faster? Reversing a list or reversing a vector?
(6) What is the minimum number of mutexes needed to create deadlock?
(7) What is the minimum number of serializers needed to create deadlock?
(8) T/F, if we perform purely functional programming, there is no need for serializers and mutexes
(9) Explain how analyzing an expression differs from evaluating an expression

Solutions:

(1) define creates new binding in current environment if it finds no bindings, set! only changes existing bindings
(2) False
(3) False
(4) Vector is faster
(5) Same
(6) 2
(7) 2
(8) True
(9) Analyzing an expression doesn't give return value, it gives an expression procedure. To evaluate an expression after analyzing, we call the returned expression-procedure with the environment

Exercise 5. Define the stream of even powers of 2, without defining procedures (no lambdas). You may use the stream `ones` as a building block.

Solutions:

```
(define powers
   (cons-stream 1 (stream-map * fours powers)))

(define fours
```

```
  (stream-add ones ones ones ones))
```

Exercise 6. Draw environment diagrams resulting from evaluating the following:

```
(define foo
  (let ((x 3))
    (lambda ()
      (if (= x 1)
  x
  (* x (begin (set! x (- x 1)) (foo)))))))

(foo) ;;return value: _____

(foo) ;;return value: _____
```

*Solutions:*Use `envdraw`

Exercise 7. We would like to implement part of glookup using mapreduce. Suppose that the format of the data is a stream of key-value pairs. Each key is a login (such as `bh`), and each value is a list of the form `(name grades)`, in which `grades` is a list of grades, and each grade is a number. Write the mapper function, the reducer function, and the base case for a mapreduce call to find out which student (name and login) has the highest total score, assuming that the grades are nonnegative integers and that there are no ties.

Solutions:

```
(define (use-grade-as-key input-pair)
  (list (make-kv-pair (accumulate + 0 (cadr (kv-value input-pair)))
                      (list (kv-key input-pair) (car (kv-value input-pair))))))

(kv-value (stream-car (mapreduce use-grade-as-key (lambda (x y) (cons x y)) "/grade-info")))
```

Exercise 8. Give two examples where analyzing evaluator would perform faster than the MC-evaluator.
Consider the following code:

```
> ((lambda (x) (+ (* x x) (* x x) (* x x)) 3)
```

Will the analyzing evaluator perform faster than the MC-evaluator?

Solutions:

Exercise 9. Assert all necessary rules such that assertions like `((1 2) is sublist of (1 2 3))` is satisfied and assertions like `((1 2) is sublist of (1))` is not. Remember that a sublist cannot be a deep element, so `(1 2)` is not a sublist of `((1 2))`.

Solutions:

```
(as! (rule (sublist () (?car . ?cdr))))
(as! (rule (sublist () ())))
(as! (rule (sublist (?x . ?cdr1) (?x . ?cdr2))
           (append ?cdr1 ?dontcare ?cdr2)))
(as! (rule (sublist (?x . ?cdr1) (?y . ?cdr2))
           (sublist (?x . ?cdr1) ?cdr2)))
```

Exercise 10. Write program `all-sublists-of-length` that takes a list and a length as argument for the non-deterministic evaluator that successively returns all the sublists of an input list of a given length. For example:

```
> (all-sublists '(1 2 3) 2)
 (1 2)
> try-again
 (2 3)
> try-again
 no more values
```

Solutions:

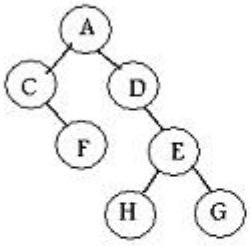
```
(define (first-n-elements ls n)
  (if (= n 0) '()
      (cons (car ls) (first-n-elements (cdr ls) (- n 1)))))

(define (all-sublists-length ls n)
  (if (null? ls) (amb)
      (amb (first-n-elements ls n) (all-sublists-length (cdr ls) n))))
```

Exercise 11. Suppose we would like to modify the metacircular evaluator to support an additional special form: `delete!`. `delet!` will take a symbol and remove it from the environment entirely. Consult the code for the metacircular evaluator and describe what changes you would make to implement `delete!`

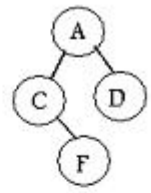
*Solutions:*Very similiar to `set!` consult that.

Exercise 12. Write a procedure `(sub-btree tree1 tree2)` that takes input 2 binary-trees and returns `#t` if tree1 is a *subtree* of tree2 and `#f` if not. We say that a tree B is a subtree of tree A if all the data (plural of datum) in tree B are also in tree A AND have the same parent-child relationship. So if in tree B, a node with datum 1 is the parent of a node with datum 2, then in tree A, there must also be a node with datum 1 who is the parent of a node with datum 2. In another words, we can fit tree B into tree A.

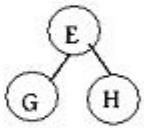
For example, in the following diagram, Tree 2, Tree 3 are both subtree of Tree 1; But Tree 4 is NOT a subtree of Tree 1 because A is not a parent of E in Tree 1.



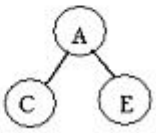
Tree 1.



Tree 2.



Tree 3.



Tree 4.

IMPORTANT Note: input to `sub-btree` will be binary trees so use the selectors `entry, left-child, right-child`.

Answer:

Exercise 13. A *run* in a word is a sequence of one letter repeated. For example, the word `gaattaccca` has a run of one `g`, a run of two `a`, a run of two `t`, a run of one `a`, a run of three `c`, and a run of one `a`.

Create a procedure `longest-run` that takes as argument a word and returns the longest-run in the word. If multiple runs of different letters have the same longest length, you may return any one of them. For example:

```
STk> (longest-run 'gaattaccca)
  ccc
STk> (longest-run 'cs61a)
  s
STk> (longest-run 'brrrr)
  rrrr
```

HINT: You may find it helpful to define a helper procedure `first-run` that takes as arguments a word and returns the first run in the word.

Answer:

Exercise 14. After many Berkeley students complained that the Berkeley buses are frequently late, AC Transit hired you to revise their bus scheduling program. The previous program uses OOP and has two main classes: `bus-line` and `bus`. The code for their scheduling program is as followed:

```
1. (define-class (bus-line num)
2.   (class-vars (buses '()) (num-of-stops 0))
3.   (method (num-of-bus?) (length buses))
4.   (method (num-of-stops?) num-of-stops)
     ;; ADD EXTRA METHODS AS NEEDED



      )

5. (define-class (bus line-num)
6.  (parent (bus-line line-num))
7.   (class-vars (all-buses '())
8.   (instance-vars (next-stop ""))
9.   (initialize (set! all-buses (cons self all-buses)))
10.  (method (get-line) line-num)
11.  (method (all-bus-from-line num)

12.     ________________________________________)
13.  (method (all-next-stops-from-line num)

14.     ________________________________________)
      ;; ADD EXTRA METHODS AS NEEDED



     )
```

(1) In line 1-9, write down which lines of code, if any, demonstrate incorrect object-oriented programming. Also state how you would fix these lines.

(2) `(all-bus-from-line num)` should returns a list of all buses of a certain bus-line. Fill in the blank to implement this method; add extra methods to either the class `bus` or `bus-line` as needed. Assume all your corrections from part 1. has been implemented.

(3) `(all-next-stops-from-line num)` should return a list of stops that buses of a certain bus-line will stop at next. Fill in the blank to implement this method; add extra methods to either the class `bus` or `bus-line` as needed. Assume all your corrections from part 1. has been implemented.

Answer:

Exercise 15. Suppose we would like to implement `set-datum!` that takes a Tree and a value as arguments and sets the datum of the input Tree to the input value. Consider the following proposed way of implementing `set-datum!`, state whether they are correct or not. If not, state why.

(1) `(define (set-datum! tree val) (set! (datum tree) val))`
(2) `(define (set-datum! tree val) (set-car! tree val))`

Is it possible to implement `set-datum!`? How might you do this?

Solutions:

```
(define (set-datum! input-tree val)
   (define temp datum)
   (set! datum
      (lambda (x) (if (eq? input-tree x) val (temp x)))))
```