CS 61A Struct. and Interp. of Computer Programs
Summer 2009 Min Xu MT 1 Soln

PRINT your name: Turing, (last) Alan (first)

SIGN your name: _____

PRINT your Inst account login: _____

Your section time (e.g., Tue 3pm): _____

Full name of the person sitting to your left: _____

Full name of the person sitting to your right: _____

You may consult any books, notes, or other paper-based inanimate objects available to you. Calculators and computers are not permitted. Please write your answers in the spaces provided in the test; in particular, we will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there.

The exam is untimed. There are 7 questions, of varying credit (100 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

Do not turn this page until your instructor tells you to do so.

Problem 1	/ 12
Problem 2	/ 14
Problem 3	/ 12
Problem 4	/ 14
Problem 5	/ 18
Problem 6	/ 12
Problem 7	/ 18
Total	/ 100

Problem 1. Scheme Basics (12 pts)

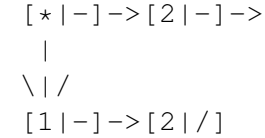
Part A. What will Scheme print if you type the following expressions into the interpreter? Some may result in error, if so, you may just state "ERROR" without what type of error it is.

`((lambda (x) (x x x)) 7)`	ERROR
`(if '#f 'yes 'no)`	no
`((compose butfirst first) '(penny lane))`	enny
`((lambda (f) (f 2 3)) (first '(+)))`	ERROR

Part B. What will Scheme print if the type the following expressions into the interpreter? If the result does not produce an error, also **draw the box-and-pointer diagram of value returned by the expression**

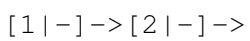
```
(cons (list 1 2) (list 2 4))
```

```
((1 2) 2 4)
```



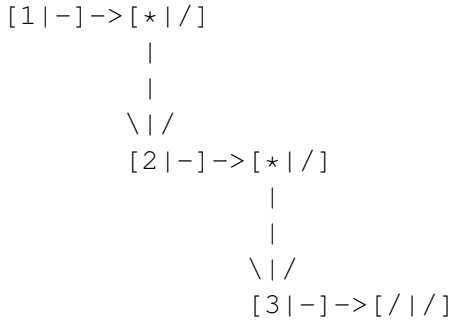

```
(append (cons 1 (list 2)) (cons nil nil))
```

```
(1 2 ())
```




```
(list 1 (list 2 (list 3 nil)))
```

```
(1 (2 (3 ())))
```



Problem 2. Asymptotic Runtime + Recursive/Iterative Process (14 pts)

Part A. Circle all the accurate statements among the following:

1. On the same computer, a $\Theta(n)$ program always runs faster than a $\Theta(n^2)$ program.
2. On the same computer and given large enough input, two programs that are both $\Theta(n)$ will take about the same amount of time to finish.
3. On the same computer and given large enough input, a $\Theta(n)$ program will run faster than a $\Theta(n^2)$ program.
4. If we are clever, we can make all programs run in $\Theta(n)$ time or faster.

Part B. The `first-streak` program from homework is presented below along with a *new version* of `best-streak`. In the recursive call of the new `best-streak`, we skip the first streak completely as opposed to just skipping the first element.

```
(define (first-streak series)
  (cond ((empty? series) 0)
        ((empty? (bf series)) 1)
        ((equal? (first series) (first (bf series)))
                (+ 1 (first-streak (bf series))))
        (else 1)))

(define (best-streak series)
  (if (empty? series) 0
      (max (first-streak series)
           (best-streak
                 ((repeat bf (first-streak series)) series)))))
```

Let n be the length of `series`.
What is the big-Theta runtime of `first-streak` ? $\Theta(n)$

What is the big-Theta runtime of `best-streak` ? $\Theta(n)$

Does `first-streak` use an iterative process or a recursive process? Recursive

Does `best-streak` use an iterative process or a recursive process? Recursive

Problem 3. Recursion (12 pts)

It is a little-known fact that Matloob is a notorious international art thief. One day, Matloob broke into the Louvre Museum and found that he brought too small of a knapsack to steal everything; Matloob can't put more than a maximum pounds of weight in his knapsack. Matloob also has, with him, a list describing how much each item in the Louvre weigh and how much dollars they are worth. Matloob would like select the items to put into his knapsack such that his loot is worth the most amount of money.

We will represent each item as a pair whose car is the weight and whose cdr is the dollar values. The item list will then just be a list of items.

Help Matloob out by writing a procedure `(knapsack ls max-wt)` that takes an item list, a maximum weight, and determine what is the most amount of dollars Matloob can loot.

For example:

```
> (knapsack '((11 . 5000) (9 . 2600) (9 . 2600)) 18)
 5200
```

Because in this case, even though the 11 pound 5000 dollar item is the most expensive, it is better for Matloob to pack the two 9 pound 2600 dollar items since his bag can take 18 pounds of weight.

Luckily, Charles has written out some parts of the program for you but Charles does not know how to finish. Charles goes to Chris for help and although Chris does not know how to write the Scheme code, Chris observes "we can either try to bring the first item with us, see what is the most money we can steal or we can just decide to not bring the first item, and see how much money we can steal."

Complete the program. (**Note:** Do not fight the problem and write your own procedure, we will NOT grade any extraneous code)

```
(define (knapsack ls max-wt)

  (cond ((null? ls) 0 )

        ((<= max-wt 0) 0)

        ((> (caar ls)  max-wt) (knapsack (cdr ls) max-wt))

        (else (max (+ (cdar ls) (knapsack (cdr ls) (- max-wt (caar ls))))

                   (knapsack (cdr ls) max-wt)))))
```

Problem 4. Higher-order Procedures (14 pts)

Part A You are collecting statistics about the political make-up of Berkeley and you get a raw count like:

Democrat	11230
Republican	781
Independent	4213
Green Party	123

We can take this data and use it to find the percentages of democrats, republicans, etc. by dividing all the numbers by the total. This process is called *normalization*.

Define a procedure `(normalize sent)` that takes a sentence of *p*ositive numbers and normalizes the sentence by dividing all the numbers by the total. You may assume that at least one number is greater than zero. **Use only higher-order procedures every, keep, collapse, repeat, compose, (you may not need all 5) DO NOT USE recursion**

For example:

```
> (normalize '(2 2 4))
  (0.25 0.25 0.5)
```

```
(define (normalize sent)
   (every (lambda (x) (/ x (collapse + 0 sent))) sent))
```

Part B Define a procedure `(apply-all ls)` that takes a list of *functions of one argument*, and returns a function of one argument. Applying the output function on an argument should be equivalent to applying all the functions in the input-list on the argument with the right-most function applied first. **Use only higher-order procedures map, filter, accumulate, repeat, compose (you may not need all 5), DO NOT USE recursion.**

Map, filter, accumulate are basically every, keep, collapse that work on lists. Their definitions are provided on the next page.

For example:

```
> ((apply-all (list even? square 1+)) 3)
  #t
```

Because `(1+ 3)` is 4, `(square 4)` is 16, and `(even? 16)` is `#t`. Note that here, we are calling the `list` procedure to make our actual procedure-list so the `list` procedure is not among our input-list.

```
(define (apply-all ls)
  (accumulate compose (lambda (x) x) ls))
```

Definitions for every, keep, collapse

```
(define (every fn sent)
  (if (empty? sent) '()
      (se (fn (first sent))
          (every fn (bf sent)))))

(define (keep pred sent)
  (cond ((empty? sent) '())
        ((pred (first sent)) (se (first sent)
                                 (keep pred (bf sent))))
        (else (keep pred (bf sent)))))

(define (collapse fn base sent)
   (if (empty? sent) base
       (fn (first sent)
           (collapse fn base (bf sent)))))
```

Definitions for map, filter, accumulate

```
(define (map fn ls)
  (if (null? ls) '()
      (cons (fn (car ls))
            (map fn (cdr ls)))))

(define (filter pred ls)
  (cond ((null? ls) '())
        ((pred ls) (cons (car ls)
                         (filter pred (cdr ls))))
        (else (filter pred (cdr ls)))))

(define (accumulate fn base ls)
  (if (null? ls) base
      (fn (car ls)
          (accumulate fn base (cdr ls)))))
```

Definitions for compose, repeat

```
(define (compose f g)
  (lambda (arg) (f (g arg))))

(define (repeat fn n)
  (if (= n 0) (lambda (x) x)
      (compose fn (repeat fn (- n 1)))))
```

This page is intentionally filled with jokes

1. Hannibal Lecter is out shopping for some brains at the butcher shop. He begins to question the butcher about the cost of these brains.

   “How much does it cost for a mathematician brain?”

   “Three dollars an ounce.”

   “How much does it cost for a programmer brain?”

   “Four dollars an ounce.”

   “How much for a lawyer brain?”

   “$1,000 an ounce.”

   “Why is lawyer brain so much more?”

   “Do you know how many lawyers we had to get through to get one ounce of brain?”

2. The devil visited a lawyer’s office and made him an offer. ”I can arrange some things for you, ” the devil said. ”I’ll increase your income five-fold. Your partners will love you; your clients will respect you; you’ll have four months of vacation each year and live to be a hundred. All I require in return is that your wife’s soul, your children’s souls, and their children’s souls rot in hell for eternity.”

   The lawyer thought for a moment and asked, ”What’s the catch?”

3. Patient to optometrist: Im very worried about the outcome of this operation, doctor. What are the chances?

   Optometrist to patient: Dont worry, you wont be able to see the difference.

4. Various Proofs that All Odd Numbers are Prime:

   Physicist: 3 is prime, 5 is prime, 7 is prime, 9 is an experimental error, 11 is prime ...

   Psychologist: 3 is a prime, 5 is a prime, 7 is a prime, 9 is a prime but tries to suppress it...

   Computer Scientist: 10 is prime, 11 is prime, 101 is prime...

   Redneck: 3 is prime, 5 is prime, 7 is prime, 9 is prime, 11 is prime ...

5. A wealthy man was having an affair with an Italian woman for several years. One night, during one of their rendezvous, she confided in him that she was pregnant. Not wanting to ruin his reputation or his marriage, he paid her a large sum of money, if she would go to Italy to secretly have the child. If she stayed in Italy to raise the child, he would also provide child support until the child turned 18.

   She agreed, but asked how he would know when the baby was born. To keep it discrete, he told her to simply mail him a post card and write ”Spaghetti” on the back. He would then arrange for child support payments to begin.

   One day, about 9 months later, he came home to his confused wife.

   ”Honey,” she said. ”You received a very strange post card today.”

   ”Oh, just give it to me and I’ll explain it,” he said.

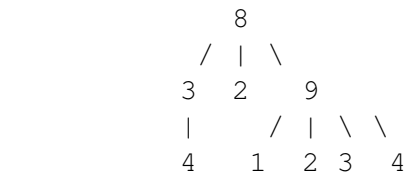
   The wife obeyed and watched as her husband read the card, turned white and fainted.

   On the card was written: ”Spaghetti, Spaghetti, Spaghetti Spaghetti. Two with sausage and meatballs, two without.”

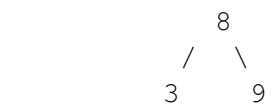
Problem 5. Tree Recursion (18 pts)

This question concerns the ADT Tree with constructor `make-tree` and selectors `datum` and `children`.

Write a procedure `(trim tree)` that takes a Tree and returns a new Tree with all the leaves (Trees with no children) removes. For instance, if we apply `trim` on the following Tree:



We will get back a Tree that looks like:



Solution1:

```
(define (trim tree)
   (make-tree (datum tree) (trim-forest (children tree))))

(define (trim-forest ls)
   (cond ((null? ls) '())
         ((null? (children (car ls))) (trim-forest (cdr ls)))
         (else (cons (trim (car ls)) (trim-forest (cdr ls))))))
```

Solution 2:

```
(define (trim tree)
   (make-tree (datum tree)
      (map trim
          (filter (lambda (t) (not (null? (children t)))) (children tree)))))
```

Problem 6. Data Abstraction (12 pts)

We would like to represent an adventure game character by a list with three elements:

1. First element is a list of healing potions that this character owns (a single potion is just represented as the word `potion`)
2. Second element is a word: this character's name
3. Third element is a number from 0 to 100: this character's health

The following procedure `heal` checks to see if the input character has any healing potions. If so, it returns a healed version of the character: the new character the `heal` procedure returns has the same name as the input character; its health is 100, and it has one less potion than the input character. If the input character has no potion, `heal` does nothing.

```
(define (heal char)

  (if   (null?   (GET-POTIONS   char))   char

      (MAKE-CHAR   (cdr   (GET-POTIONS   char))

                (GET-NAME   char)

                100 )))
```

Part A. We would like to define the adventure game character as an Abstract Data Type instead, fill in the following constructors and selectors.

```
(define (make-char potions name health)
(list potions name health))

(define (get-potions char)
(car char))

(define (get-name char)
(cadr char))

(define (get-health char)
(caddr char))
```

Part B. Go back to the `heal` procedure and cross out all Data Abstraction violations in the code and correct the violations in the space underneath the code.

Problem 7. Deep-list (18 pts)

Often times, such as representing multi-dimensional tables, it is advantageous to enforce that all list in a deep-list must either contain ALL lists or ALL atoms. Let us call a deep-list that fits that criterion *pure*.

For example, `(1 (2))` is not pure because the first level list contains both `1`, an atom, and `(2)`, a list. However, `((1) (2))` is a pure list. `((1 (2)) (3))` is not pure because the second level list `(1 (2))` is not pure.

So, we can recursively define a pure-deep-list to be a list that either contains all atoms or all pure-deep-lists.

Write a procedure `(pure? ls)` that takes a deep-list and returns `#t` if and only if the input deep-list is pure. For example:

```
> (pure? '((1 (2)) 3))
 #f
> (pure? '((1 2) (3)))
 #t
> (pure? '(((1) (2)) (3)))
 #t
```

Solution1:

```
(define (pure? ls)
  (cond ((null? ls) #t)
        ((atom? ls) #t)
        ((null? (cdr ls)) #t)
        ((and (atom? (car ls)) (atom? (cadr ls)))
           (pure? (cdr ls)))
        ((and (list? (car ls)) (list? (cadr ls)))
           (and (pure? (car ls)) (pure? (cdr ls))))
        (else #f)))
```

Solution2:

```
(define (pure? ls)
  (or (accumulate (lambda (x y) (and (atom? x) y)) #t ls)
      (accumulate (lambda (x y)
                    (and (list? x) (pure? x) y)) #t ls)))
```

This page is intentionally left blank as scratch paper

Anything written on this page **will not be graded** unless you specifically tell us to.

This page is intentionally left blank as scratch paper

Anything written on this page **will not be graded** unless you specifically tell us to.