University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2003 Instructor: Kurt Meinz 2003-07-25

CS 61A Midterm #2

Personal Information

First and Last Name	
Your Login	cs61a-__ __
Lab Section Time & Location ***you attend***	
Discussion Section Time & Location ***you attend***	
"All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61a who have not taken it yet."	***(please sign)***

Instructions

- Partial credit may be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- Feel free to use any Scheme function that was described in lecture or sections of the textbook we have read without defining it yourself. Do not use functions or constructs that we have not yet covered. Unless specifically prohibited, you are allowed to use helper functions on any problem.
- Please use "true" instead of #t, and "false" instead of #f. We have found that handwritten #t and #f look too much alike.
- Please write legibly! If we can't read it, we won't grade it!
- Feel free to draw funny pictures/messages on the exam.

Grading Results

Question	*Max. Points*	*Points Earned*
1	**10**	
2	**10**	
3	**10**	
4	**10**	
Total	**40**	

Name: _____ Login: _____

Question 1a: Deep List Recursion

The substitution model of evaluation says that, before a procedure body can be evaluated, any occurrences of the formal parameters in the body must be replaced by the actual argument values in the invocation. We would like to model this behavior by writing a 'substitute' procedure that takes two arguments, a list of parameter-name and actual-argument-value pairs (represented as 2-element lists) and an input list that looks like a Scheme expression. You procedure should replace every occurrence of a parameter name with its respective value in the target list. Examples:

```
> (substitute '((x 3) (y 4)) '(+ (* x y) (- y z)))
(+ (* 3 4) (- 4 z))
> (substitute '((foo (lambda (x) (* x x))) '(foo 4))
((lambda (x) (* x x)) 4)
> (substitute '((z 17) (x (+ y z)) (y 12)) '(* x z))
(* (+ y z) 17)
> (substitute '((func (lambda (x) (* x x))) (lst (list 1 2 3)))
              '(cons (func (car lst)) (map func (cdr lst))))
(cons ((lambda (x) (* x x)) (car (list 1 2 3)))) (map (lambda (x) (* x x))
(cdr  (list 1 2 3)))))
```

Hints:

1. We suggest using two recursive processes. One for each pair, and one to go down the list of pairs.
2. Use "equal?".

```
(define (substitute replace-lst target-lst)
```

Name: _____ **Login:**_____

Question 1b: Substitute, continued.

The naïve substitution procedure that you wrote in Part A would be cooler if it could do repeated substitutions. Please write a new procedure named 'supersub' that takes the same arguments as the usual sub, but, this time, the parameter-value list may contain values that contain more parameters. Your 'supersub' proc should repeatedly substitute values for parameters until no more substitutions can be made. You may assume that there will be no cycles.

```
> (supersub '((z 17) (x (+ y z)) (y 12)) '(* x z))
(* (+ 12 17) 17)
> (supersub '((e 14) (d e) (b c) (c d) (a b)) '(list a b c d e))
(list 14 14 14 14 14)
```

Hint: If you didn't get Part A, here's you chance to redeem yourself. Assume you have a working version of the 'substitute' proc from Part A. Do not define any helper functions, and do not modify the 'substitute' procedure.

```
(define (supersub replace-lst target-lst)
```

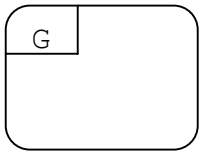
Name: _____ **Login:**_____

Question 2: Alex and Carolen's Question

Please draw an environment diagram for the evaluation of the following expressions. Your diagrams should include all googly pairs created (even unnamed ones) and all frames created should be labeled in creation order. You need not include the function body in the googly pair, but please do include the parameter list.

```
(let ((+ (let ((- (lambda (x y) (+ x y 1))))
           (lambda (a b) (+ a (- 2 b))))))
  (+ 3 (- 4 (+ 2 5))))
```

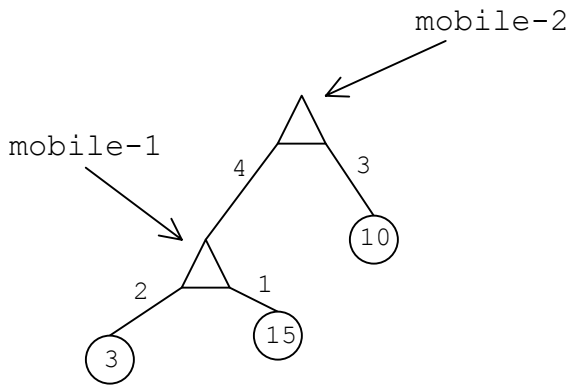
Hint: Translate the lets to lambdas first!



Name: _____ **Login:**_____

Question 3: OOP Mobile

Remember the mobiles from homework 3.1? Well, here's your second chance at figuring out whether a mobile is balanced or not. This time, however, we will represent the mobiles in OOP rather than as nested lists. To build up this mobile



You'd say:

```
(define weight-5  (instantiate weight 5))
(define weight-10 (instantiate weight 10))
(define weight-20 (instantiate weight 20))

(define mobile-1 (instantiate mobile
                       (instantiate branch 2 weight-5)
                       (instantiate branch 1 weight-10)))
(define mobile-2 (instantiate mobile
                       (instantiate branch 4 mobile-1)
                       (instantiate branch 3 weight-20)))
```

Please give definitions for these classes such that (ask <amobile> 'balanced?) will return #t if the mobile is balanced (as described in the homework) and #f otherwise and (ask <amobile> 'total-weight) will return the sum of all the weights in that mobile.

You may NOT use 'if' or 'cond' in your solution!

Hints:

1. Look at the solution for the homework version of this problem – it is attached to the back of the exam.
2. Each object (mobile, branch, and weight) can determine both its weight and whether it is balanced or not at instantiation time.

You may put your answers on the next page.

Name: _____ **Login:**_____

Question 3: OOP Mobiles, cont

```
(define-class (weight value)
```

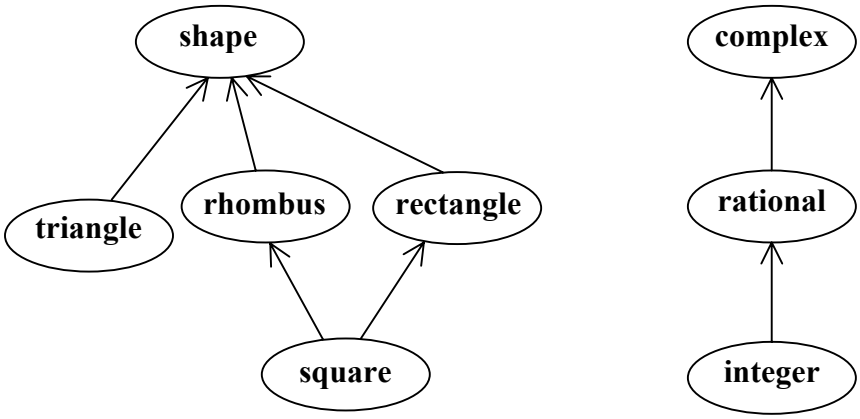
```
(define-class (branch branch-length branch-struct)
```

```
(define-class (mobile left-branch right-branch)
```

Name: _____ **Login:**_____

Question 4a: Hierarchy of Types

We want to use OOP to model the tree-like structure of relations among different types. For example, here are two type hierarchies that you have seen (Section 2.5):



The arrows denote super-types. Looking from the bottom up, the lower nodes are specializations of the types to which they point (e.g. an **integer** is a kind of **rational**, which is a kind of **complex**; a **square** is both a **rectangle** and a **rhombus**). Hence, the types at the top of the hierarchy—**shape** and **complex**—will be the most general and the ones at the bottom will be the least.

We would like you to represent this type hierarchy in OOP. Each type will be represented by an instance of a Typeobj class, and will keep a list of its immediate super-types (e.g. **square** should have a list containing the **rhombus** and **rectangle** objects). Notice that in this system, a type can have any number of direct super-types.

Here are the requirements for the system:

1. Each Typeobj must be constructed with exactly one instantiation variable: 'type-name'
2. Each object must have an 'add-super-type' method that takes another Typeobj and records it as a super-type of itself; the return value is up to you.
3. Each object must have an 'isa?' method that takes a Typeobj and returns true if it is a specialization of the argument object.

Here is some sample usage that will construct part of the shape hierarchy shown above. Make sure your system will work for these calls.

```
(define shape-object     (instantiate Typeobj 'shape))
(define rhombus-object   (instantiate Typeobj 'rhombus))
(define rectangle-object (instantiate Typeobj 'rectangle))
(define square-object    (instantiate Typeobj 'square))
(ask rhombus-object    'add-super-type shape-object)
(ask rectangle-object  'add-super-type shape-object)
(ask square-object   'isa? shape-object)                ➔ #f
(ask square-object     'add-super-type rhombus-object)
(ask square-object     'add-super-type rectangle-object)
(ask square-object   'isa? rhombus-object)              ➔ #t
(ask rhombus-object   'isa? square-object)              ➔ #f
(ask square-object   'isa? shape-object)                ➔ #t
```

Name: _____ **Login:**_____

```
(ask shape-object   'isa? shape-object)                ➔ #t
```

Part A:

Please provide a class definition for Typeobj that satisfies all of the requirements.

```
(define-class (Typeobj type-name)
```

Name: _____ **Login:**_____

Question 4b: Hierarchy of Types, continued

The Typeobj class you created in the previous question is not all that useful by itself. It would be more useful if it would serve as the basis for an extension of the type-tagging system presented in Section 2.4. Currently, tagging an object serves only to tell us it's type; we'd like to extend this mechanism to also tell us about the hierarchical relationships among our tagged objects. Specifically, we'd like you to change the type-tag interface so that it supports the 'isa?' relation among tagged objects.

Let's say we've got ways to create various shapes (the implementation details are unimportant) and we've made a couple of these shapes and tagged them:

```
(attach-tag 'square    (make-square  . . .))
(attach-tag 'shape     (make-shape   . . .))
(attach-tag 'rhombus   (make-rhombus  . . .))
(attach-tag 'rectangle (make-rectangle  . . .))
```

After tagging an object, the type-tag can be used in two functions that you will write:

```
(subtype? 'square 'shape)     ➔ #f
(subtype? 'rhombus 'rhombus)  ➔ #t

(register-as-super-type 'rhombus 'shape)        ;; rhombus is a kind of shape
(register-as-super-type 'rectangle 'shape)      ;; so is rectangle
(register-as-super-type 'square 'rhombus)       ;; square is a type of rhombus
(register-as-super-type 'square 'rectangle)     ;; it is also a rectangle

(subtype? 'square 'rhombus)    ➔ #t
(subtype? 'square 'rectangle)  ➔ #t
(subtype? 'square 'shape)      ➔ #t
```

The functionality is the same as in Part A: a way to test if one type is a specialization of another, and a way to assert such a relationship. What's different is that the Typeobj instances that implement this hierarchy are hidden from the user of the type-tagging system; **'subtype?' and 'register-as-super-type' are regular Scheme functions**—not methods—and they take in words, not instances.

Your job is to use the Typeobj class from Part A to implement this mechanism. The new interface should have these procedures:

```
1. (attach-tag type-tag thing)          : returns a tagged object
2. (contents tagged-object)             : returns the thing
3. (type-tag tagged-object)             : returns the type-tag
4. (register-as-super-type type1 type2) : declares type1 a subtype of type2
5. (subtype? type1 type2) : returns true if type1 is a specialization of type2
```

Hints:

1. You'll need some way to get from a type-name to the Typeobj that represents it. One easy way to do this is to use the global table ('get' and 'put').
2. You may assume that before a type can be an argument to 'subtype?' or 'register-as-super-type,' it will be used in 'attach-tag.' Therefore, any bookkeeping you need to do to enter a type into this system can be done in 'attach-tag.'
3. The 'subtype?' function should make use of the 'isa?' method, and the 'register-as-super-type' function should use the 'add-super-type' method.

```
(define (attach-tag type-tag thing)




(define (type-tag tagged-object)


(define (contents tagged-object)


(define (subtype? type1 type2)




(define (register-as-super-type type1 type2)
```

Name: _____ **Login:**_____

SOLUTION FOR HW3-1 QUESTION 3 (mobiles)

**** Nothing on this or the next page will be graded. It is not part of the exam. ****

```
;; It would have been better to create MOBILE? and BRANCH? predicates, to
;; avoid duplicating code in the selectors:

;; Part A

(define (mobile? thing)
  (and (list? thing) (equal? (car thing) 'mobile)))

(define (branch? thing)
  (and (list? thing) (equal? (car thing) 'branch)))

(define (left-branch mobile)
  (if (mobile? mobile)
      (cadr mobile)
      (error "Not a mobile -- LEFT-BRANCH: " mobile)))

(define (right-branch mobile)
  (if (mobile? mobile)
      (caddr mobile)
      (error "Not a mobile -- RIGHT-BRANCH: " mobile)))

(define (branch-structure branch)
  (if (branch? branch)
      (caddr branch)
      (error "Not a branch -- BRANCH-STRUCTURE: " branch)))

(define (branch-length branch)
  (if (branch? branch)
      (cadr branch)
      (error "Not a branch -- BRANCH-LENGTH: " branch)))

;; Part B

;; All TOTAL-WEIGHT does is sum the "leaves" of a mobile -- the weights.
;; How can you tell if you've reached a weight? It's just a number!

(define (total-weight mobile)
  (if (number? mobile)
      mobile
      (+ (total-weight (branch-structure (left-branch mobile)))
 (total-weight (branch-structure (right-branch mobile))))))

;; Something like...
;;
;;  (+ (total-weight (left-branch mobile))
;;     (total-weight (right-branch mobile)))
;;
;; ... should not work because LEFT-BRANCH gives you a branch, but
;; TOTAL-WEIGHT takes a mobile (or a number, too). The other way to solve this
;; problem was with cases; you'd need four of them:
;;
;; 1. Is the structure on the left and right a number? (base case)
;; 2. Is the structure on the left a number? (one recursive call)
;; 3. Is the structure on the right a number? (one recursive call)
```

Name: _____ **Login:**_____

```
;; 4. None of the above. (two recursive calls)

;; Part C

;; It looks like we're going to need a function to compute the
;; torque of a branch:

(define (torque branch)
  (* (branch-length branch)
     (total-weight (branch-structure branch))))

;; For a mobile to be balanced, three things must be true:
;;
;; 1. The torque applied by its left branch must equal that applied
;;    by its right branch.
;; 2. The mobile on the left must be balanced.
;; 3. The mobile on the right must be balanced.

(define (balanced? mobile)
  (if (number? mobile)
      #t
      (and (= (torque (left-branch mobile))
      (torque (right-branch mobile)))
   (balanced? (branch-structure (left-branch mobile)))
   (balanced? (branch-structure (right-branch mobile))))))
```