CSU2510H Exam 1 (SOLUTION) – Spring 2012

- You may use the usual primitives and expression forms of any of the `class` languages; for everything else, define it.
- You may write `c` $\rightarrow$ `e` as shorthand for `(check-expect c e)`.
- To add a method to an existing class definition, you may write just the method and indicate the appropriate class name rather than re-write the entire class definition.
- We expect data *and* interface definitions.
- If an interface is given to you, you do not need to repeat the contract and purpose statements in your implementations. Likewise, you do not need to repeat any test cases given to you, but you should add tests wherever appropriate.
- The extra credit problem is *all or nothing*; no partial credit will be awarded.
- Unless specifically requested, templates and super classes are *not* required.
- Some basic test taking advice: Before you start answering any problems, read *every* problem, so your brain can think about the harder problems in the background while you knock off the easy ones.

Problem	Points	/out of
1		/ 18
2		/ 10
3		/ 15
4		/ 15
5		/ 18
Extra		/ 5
Total		/ 76+5

Good luck!

Problem 1 Here is a data definition for a class of objects. It intentionally uses meaningless names to test whether you've sufficiently internalized the data-driven development process we've talked about since the Fall. 18 POINTS

```
;; A P is one of:
;; - A                  ;; An A is a (new a% String)
;; - B                  ;; A  B is a (new b% String A)
;; - C                  ;; A  C is a (new c% P)

(define-class a% (fields w))
(define-class b% (fields x y))
(define-class c% (fields z))
```

Design the following methods for `P`s:

1. `size` - the size of an `A` is 5; the size of a `B` is 8, regardless of what's inside; and the size of a `C` contributes 1 to the size of the `P` inside.

2. `replace-x` - produces a `P` like this one, but with any `x` field replaced by the given string.

3. `same-x-and-w?` - given two strings, produce true if this `P` contains a `B` whose `x` is equal to the first given string and whose `y` is an `A` whose `w` is equal to the second given string. Produces false otherwise.

Be sure to define any interfaces you need and say which classes implement which interfaces so that your program treats all values according to their interface.

```
;;> [6 pts -- contract and purpose for each method]
;; A P implements:
;; - size : -> Number
;;   Size of this P.
;; - replace-x : String -> P
;;   Replace x field (if any) of this P with given string.
;; - same-x-and-w? : String String -> Boolean
;;   Is there a B within this P with given x field and w field?

;;> [0 pts -- not required, but write something]
;; An A implements:
;; - w : -> String
;;   Get the w string.

;;> [2 pts]
;; In a%:
(define (size) 5)
(define (replace-x str) this)
(define (same-x-and-w? g c) false)

;;> [3 pts]
;; In b%:
(define (size) 8)
(define (replace-x str)
  (new b% str (this . y)))
(define (same-x-and-w? g c)
  (and (string=? g (this . x))
       (string=? c (this . y . w))))

;;> [4 pts]
;; In c%:
(define (size) (add1 (this . z . size)))
(define (replace-x str)
  (new c% (this . z . replace-x str)))
(define (same-x-and-w? g c)
  (this . z . same-x-and-w? g c))

;; tests
```

```
;;> [3 pts -- 1 per method]
(define a (new a% "a"))
(define b (new b% "b" a))
(define c (new c% b))
(define d (new c% a))
(check-expect (a . size) 5)
(check-expect (b . size) 8)
(check-expect (c . size) 6)
(check-expect (d . size) 9)
(check-expect (a . replace-x "*") a)
(check-expect (b . replace-x "*") (new b% "*" a))
(check-expect (c . replace-x "*") (new c% (new b% "*" a)))
(check-expect (d . replace-x "*") d)
(check-expect (a . same-x-and-w? "b" "a") false)
(check-expect (b . same-x-and-w? "b" "a") true)
(check-expect (c . same-x-and-w? "b" "a") true)
(check-expect (d . same-x-and-w? "b" "a") false)
```

Problem 2 Here are data, class, and interface definitions for a list of numbers: 10 POINTS

```
;; An LoN is one of:
;; - (new mt%)
;; - (new cons% Number LoN)
;; and implements:
;; - sum : -> Number
;;   Compute the sum of the numbers in this list.
;; - sqrs : -> LoN
;;   List of squares of each number in this list.

(define-class mt%
  (define (sum) 0)
  (define (sqrs) this))

(define-class cons%
  (fields first rest)
  (define (sum)
    (+ (this . first)
       (this . rest . sum)))
  (define (sqrs)
    (new cons%
         (sqr (this . first)) ; (sqr x) = x^2
         (this . rest . sqrs))))
```

Here are some examples:

```
(check-expect ((new mt%) . sum) 0)
(check-expect ((new cons% 4 (new cons% 5 (new mt%))) . sum)  9)
(check-expect ((new mt%) . sqrs) (new mt%))
(check-expect ((new cons% 4 (new cons% 5 (new mt%))) . sqrs)
              (new cons% 16 (new cons% 25 (new mt%))))
```

Develop the `sum-sqrs` method that computes the sum of the squares of a list of numbers. Here is an example:

```
(check-expect ((new cons% 4 (new cons% 5 (new mt%))) . sum-sqrs)
              (+ 16 25))
```

If you can, define a super class of `mt%` and `cons%` and lift the definition of `sum-sqrs` to this class. (We will assume the appropriate `super` declaration in `mt%` and `cons%`.) If you cannot, define methods in both `mt%` and `cons%` for partial credit.

```
;;> [2 pts% -- class def]
(define-class list%
  ;;> [2 pts -- contract and purpose]
  ;; sum-sqrs : -> Number
  ;; compute the sum of the squares of the numbers in this list
  ;;> [4 pts -- implementation]
  (define (sum-sqrs)
    ((this . sqrs) . sum)))

;;> [2 pts -- tests]
(check-expect ((new mt%) . sum-sqrs) 0)
```

Problem 3 In order to manage class scheduling, the College decided that they needed to create a new implementation of the `Set` data structure. They went with the lowest bidder, which turned out to be a bunch of BU students. Now, having paid for this code, the College needs to test it, and they turn to you. 15 POINTS

Unfortunately, the code the BU students produced is so convoluted that it doesn't bear looking at. All you have to go on is that the `Set` data structure obeys the following interface:

```
;; A Set implements:
;; - insert : String -> Set
;;   adds given string to this set, if it isn't already there
;;   (otherwise produces this set)
;; - remove : String -> Set
;;   removes given string from this set, if it's already there
;;   (otherwise produces this set)
;; - size : -> Number
;;   count the elements of this set
;; - has? : String -> Boolean
;;   is the given String in this set?

;; empty-set is a Set value containing no elements
```

`Sets` should satisfy the following property, called the "insert-size" property, for any element and set:

> If we insert any element into any set, then the resulting set is the same size as the original set *if* the element was already in the set, and the resulting set is one larger than the original set if the element was *not* already in the set.

1. Codify the "insert-size" property as a predicate. Write a test case that should pass (if the sets work right) using the predicate you have defined.

The College wants to be really sure that their sets are correct, and they've come up with another property that they think is true, called the "insert-remove" property:

> If we insert element any element a into any set, and then remove any element b from the resulting set, the result has the same size as if we do the two operations in the reverse order, i.e., remove b, then insert a.

The College is considering adding this to the test suite as well.

2. Codify this property as a predicate, and write a test case that should pass using the predicate.

3. Is the "insert-remove" property true for all sets? If so, give an explanation. If not, write a counter-example (i.e., a failing test case) using your predicate.

```
;;> [1 pt -- contract & purpose]
;; check-insert-size : Set String -> Boolean
;; check the "insert-size" property on s and i.
;;> [4 pts -- code]
(define (check-insert-size s i)
  (= (s . insert i . size)
     (cond [(s . has? i) (add1 (s . size))]
           [else               (s . size)])))

;;> [2 pts -- test case]
(check-expect (check-insert-size empty-set "hi") true)

;;> [1 pt -- contract & purpose]
;; check-remove-insert : Set String -> Boolean
;; check the "remove-insert" property on s, a, and b.
;;> [3 pts -- code]
(define (check-remove-insert s a b)
  (= (s . insert a . remove b . size)
     (s . remove b . insert a . size))

;;> [1 pt -- successful test]
(check-expect (check-remove-insert empty-set "yes" "no") true)
```

```
;; Property does not hold.
;; Counter-example (failing test):
;;> [3 pts -- failing test]
(check-expect (check-remove-insert empty-set "yes" "yes") true)
```

Problem 4 Here are data and class definitions for representing the files and directories of a computer: 15 POINTS

```
;; An Elem is one of
;; - (new file% String Number)
;; - (new dir% String LoElem)
(define-class file% (fields name size))
(define-class dir% (fields name elems))

;; A LoElem is one of
;; - (new mt%)
;; - (new cons% Elem LoElem)
(define-class mt%)
(define-class cons% (fields first rest))
```

Here is an example of a directory tree and its representation as an `Elem`:

```
#|
 User
   +-- A
   |   +-- a.txt
   |   +-- B
   |       +-- b.rkt
   +-- C
       +-- c.c
|#

(define mt (new mt%))
(define a (new file% "a.txt" 5))
(define b (new file% "b.rkt" 10))
(define c (new file% "c.c" 1000))

(define C/ (new dir% "C" (new cons% c mt)))
(define B/ (new dir% "B" (new cons% b mt)))
(define A/ (new dir% "A" (new cons% a (new cons% B/ mt))))
(define User/ (new dir% "/" (new cons% A/ (new cons% C/ mt))))
```

Your job is to implement one of the main tasks of the common Unix utility `find`, which gives a list of all the files within an element. To do so, add a `list-files` method of `Elem` which produces a list of files. So for example, we expect:

```
(check-expect (User/ . list-files)
              (new cons% a (new cons% b (new cons% c mt))))
```

Hint: it may come in handy to be able to concatenate two `LoElem` together.

```
;;> [1pts -- interfaces, contracts, purpose]
;; An Elem implements
;; - list-files : -> LoElem
;;   List all files within this element.

;;> [2pts -- interfaces, contracts, purpose]
;; A LoElem implements
;; - list-files : -> LoElem
;;   List all files within this list of elements.
;; - append : LoElem -> LoElem
;;   Append this list of elements to given list.

;;> [1pt -- code]
;; in dir%
(define (list-files)
  (this . elems . list-files))

;;> [1pt -- code]
;; in file%
(define (list-files)
  (new cons% this (new mt%)))

;;> [2pt -- code]
;; in mt%
(define (append loe) loe)
(define (list-files) this)

;;> [4pt -- code]
;; in cons%
(define (append loe)
```

```
  (new cons% (this . first) (this . rest . append loe)))
(define (list-files)
  (this . first . list-files . append (this . rest . list-files)))

;;> [1pt -- sanity]

;;> [3pts -- tests]
;; tests
(check-expect (mt . append (new cons% a mt)) (new cons% a mt))
(check-expect ((new cons% a mt) . append (new cons% b mt))
              (new cons% a (new cons% b mt)))

(check-expect (mt . list-files) mt)
(check-expect ((new cons% a (new cons% b mt)) . list-files)
              (new cons% a (new cons% b mt)))

(check-expect (a . list-files) (new cons% a mt))
```

Problem 5 Last night, Asumu was up late improving the `class` languages. Unfortunately, he was too tired to test his work, and his last commit broke the numeric system. This puts us in the dire position of no longer being able to calculate the factorial of large numbers, and we turn to you for help. 18 POINTS

This is the definition of a class of functional objects with a `fact` method:

```
;; A Fact is (new fact%) and implements
;; - fact : Natural -> Natural
;;   Compute n!.
(define-class fact%
  (define (fact n)
    (cond [(zero? n) 1]
          [else (* n (this . fact (sub1 n)))])))

(check-expect ((new fact%) . fact 0) 1)
(check-expect ((new fact%) . fact 5) 120)
```

Since we can no longer rely on built in numbers to represent `Naturals`, we will represent `Naturals` with objects and then rewrite the `fact` method to operate over this new kind of data.

Here is the `Natural` interface:

```
;; A Natural implements:
;; - zero? : -> Boolean
;;   is this Natural the zero value?
;; - sub1 : -> Natural
;;   produce a natural one less than this natural
;;   (or zero if this natural is zero)
;; - times : Natural -> Natural
;;   multiply this natural by the given natural
```

Write a new definition of the `fact` method using the `Natural` interface definition given above, and *not* using the built-in numbers of the class system or DrRacket. You can assume there is a defined constant `one`, which is a `Natural` object representing 1.

```
(check-expect ((new fact%) . fact one) one)
(check-expect ((new fact%) . fact (one . sub1)) one)
```

```
;;> [2 pts]
(define-class fact%
  (define (fact n)
    (cond [(n . zero?) one]
          [else (n . times (this . fact (n . sub1)))])))
```

Now, design data and class defintions that implement `Natural`, and supports the methods `zero?`, `sub1`, and `times`. Define the constant `one`. (*Hint*: remember that the natural numbers can be seen as a recursive union; a natural number is either zero, or one more than a natural number.)

It will be helpful to define the helper methods `add1` and `plus`. Also, remember that $n + m = (n - 1) + m + 1$ and that $n \cdot m = (n - 1) \cdot m + m$. Don't overlook the slightly weird specification for `sub1`: $0 - 1 = 0$; this weirdness is common amongst mathematicians when working with natural numbers.

```
;;> [2 pts -- data defintion]
;; A Natural is one of
;; - (new zero%)
;; - (new add1% Natural)
(define-class zero%
  ;;> [4 pts -- methods in zero%]
  (define (sub1) this)
  (define (zero?) true)
  (define (add1) (new add1% this))
  (define (plus n) n)
  (define (times n) this))

(define-class add1%
  (fields sub1)
  ;;> [7 pts -- methods in add1%]
  (define (zero?) false)
  (define (add1) (new add1% this))
  (define (plus n) (this . sub1 . plus n . add1))
  (define (times n) (this . sub1 . times n . plus n)))

(define zero (new zero%))
;;> [1 pt -- define one]
(define one (new add1% (new zero%)))

;;> [3 pts -- tests]
(check-expect (one . zero?) false)
(check-expect (one . sub1) zero)
(check-expect ((new add1% one) . sub1) one)
```

```
(check-expect (zero . zero?) true)
(check-expect (zero . sub1) zero)
```

Problem 6 Extra Credit 5 POINTS

After Van Horn discovered the problems with numbers, he attempted to fix them. Unfortunately, not only did he not fix them, he broke booleans as well. Fortunately, we have an idea about how you can help, expressed in the following interface definitions.

```
;; A Bool implements:
;; - branch : [Action X] [Action X] -> X
;;   run the first action if this bool is true,
;;   otherwise run the second action

;; An [Action X] implements:
;; - run : -> X
;; run this action
```

Following these interface definitions, design data and class defintions for implementations of the `Bool` interface, and reimplement the `fact` method to use them. You will need to provide new defintions for `true` and `false`, since they were broken by this catastrophe as well.

```
;;> [5 pts -- get it right]
(define-class fact%
  (define (fact n)
    (n . zero? . branch (new const% one) (new recur% n))))

(define-class const%
  (fields run))

(define-class recur%
  (fields n)
  (define (run)
    ((new fact%) . fact (this . n . sub1) . times (this . n))))

(define-class t%
  (define (branch t e) (t . run)))
(define-class f%
```

```
  (define (branch t e) (e . run)))

(define true (new t%))
(define false (new f%))

(check-expect ((new fact%) . fact zero) one)                    ; 0! = 1
(check-expect ((new fact%) . fact one) one)                     ; 1! = 1
(check-expect ((new fact%) . fact (one . add1)) (one . add1)) ; 2! = 2
(check-expect ((new fact%) . fact (one . add1 . add1))        ; 3! = 6
              (one . add1 . add1 . add1 . add1 . add1))
```