CSU2510H Exam 1 (SOLUTION) – Spring 2013

- 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.

Good luck!

Problem	Points	/out of
1		/ 20
2		/ 19
Extra		/ 5
Total		/ 39+5

Problem 1 JSON is a lightweight data-interchange format. It is widely used on the web and plays exactly the role of S-Expressions in universe programs, but for JavaScript programs. It's used by folks like Google and there's tons of data on the web that's available in the JSON format. So as to not be left out of the fun, your co-op employer has asked you to design an object representation of JSON to incorporate into their 2 million lines of `class/2` code. To start things off simply, here are some examples of what we'll call "JSON, Jr.":

20 POINTS

```
"this is JSON"
[ "this is JSON", "Jr." ]
[ [], [ "So", "is" ], "this" ]
```

The first example is just a string. The second is an "array" of two JSON elements, which are both strings. The third is another array, but of three elements: the first is an array of zero elements, the second is an array of two strings, and the third is a string.

1. Design an object-based data representation for JSON, Jr.

```
;; A JSON is one of:  [5pt]
;; - (new s% String)
;; - Array
;;
;; An Array is one of
;; - (new a0%)
;; - (new a+% JSON Array)

(define-class s%
  (fields val))

(define-class a0%)

(define-class a+%
  (fields hd tl))
```

2. Design a method for counting the number of strings in a JSON, Jr. object.

```
;; JSON implements [1pts]
;; - Count the number of strings in this object
;;   count-strings : -> Natural

;; in s% [1pt]
(define (count-strings) 1)

;; in a0% [1pt]
(define (count-strings) 0)

;; in a+% [1pt]
(define (count-strings)
  (+ (this . hd . count-strings)
     (this . tl . count-strings)))

;; [1pts]
(define eg1 (new s% "This is JSON"))
(define eg2 (new a+% eg1 (new a+% (new s% "Jr.") (new a0%))))
(define eg3 (new a+% (new a0%)
                 (new a+% (new a+% (new s% "So")
                                   (new a+% (new s% "is") (new a0%)))
                          (new a+% (new s% "this")
                               (new a0%)))))

;; [1pts]
(check-expect (eg1 . count-strings) 1)
(check-expect (eg2 . count-strings) 2)
(check-expect (eg3 . count-strings) 3)
```

3. Among the 2M lines of code your employer maintains are 30,000 lines of code devoted to implementations of the `StringPred` interface, which is:

```
;; A StringPred implements
;; - apply : String -> Boolean
```

These objects represent predicates on strings. Now that your JSON, Jr. library is in place, they've decided to put their predicates to use in searching a large corpus of JSON data. To accomodate this, design the following method for JSON, Jr. objects:

```
;; Find the first string in this JSON Jr object satisfying
;; given predicate, or #f if there's no such string.
;; find : StringPred -> String or #f

;; in s%  [2pts]
(define (find p)
  (if (p . apply (this . val))
      (this . val)
      #f))

;; in a0% [1pts]
(define (find p) false)

;; in a+% [3pts]
(define (find p)
  (local [(define r (this . hd . find p))]
    (if (false? r)
        (this . tl . find p)
        r)))

;; [2pts]
;; A (new eq% String) implements StringPred
(define-class eq%
  (fields x)
  (define (apply y)
    (string=? (this . x) y)))
```

```
;; [1pts]
(check-expect (eg1 . find (new eq% "This is JSON")) "This is JSON")
(check-expect (eg1 . find (new eq% "this")) false)
(check-expect (eg2 . find (new eq% "This is JSON")) "This is JSON")
(check-expect (eg2 . find (new eq% "this")) false)
(check-expect (eg3 . find (new eq% "This is JSON")) false)
(check-expect (eg3 . find (new eq% "this")) "this")
```

Problem 2 In order to manage the distribution of ID cards, the University needs a data structure for managing sets of numbers. Van Horn started on the implementation, resulting in what you see below. 19 POINTS

```
;; A Set is one of:
;; - (new empty%)
;; - (new singelton% Number)
;; - (new union% Set Set)
;; Interpretation: a set that contains all the *unique* numbers in the data
;; and implements
;; - sum : -> Number
;;   Produce the sum of all the unique elements of this set
;; - sum/acc : Set -> Number
;;   Produce the sum of all the unique elements of this set that are
;;   *not* in the given set
;; - contains : Number -> Boolean
;;   Does this set contain the given number?
;; - union : Set -> Set
;;   Combine this set with the given set

(define-class empty%
  (define (sum) 0)
  (define (sum/acc s) 0))

(define-class singleton%
  (fields n)
  (define (sum) (this . n))
  (define (sum/acc s) (cond [(s . contains (this . n)) 0]
                            [else (this . n)])))

(define-class union%
  (fields l r)
  (define (sum) (this . sum/acc (new empty%)))
  (define (sum/acc s) (+ (this . l . sum/acc s)
                         (this . r . sum/acc (s . union (this . l))))))
```

Unfortunately, as you can see, Van Horn fell asleep before finishing, and the deadline is today. Help him out by implementing the `union` and `contains` methods for all `Sets`.

```
;;> [1 pt -- union method]
;;> [3 pts -- contains method]
;; in empty%
(define (union s) s)
(define (contains n) #f)
;; also ok:
(define (union s) (new union% this s))
;; in singleton%
(define (union s) (new union% this s))
(define (contains n) (= n (this . n)))
;; in union%
(define (union s) (new union% this s))
(define (contains n) (or (this . l . contains n)
                         (this . r . contains n)))
(define e (new empty%))
;;> [2 pts -- tests]
(check-expect (e . contains 5) #f)
(check-expect ((new singleton% 5) . contains 5) #t)
(check-expect ((new singleton% 6) . contains 5) #f)
(check-expect ((new singleton% 6) . union  e . contains 5) #f)
(check-expect (e . union (new singleton% 5) . contains 5) #t)
(check-expect ((new singleton% 5) . union (new singleton% 5) . contains 5) #t)
```

`Sets` should satisfy the following property, called the "union-contains" property, for any number n and any two sets s_1 and s_2:

> Given two sets s_1 and s_2, if n is contained in the union of s_1 and s_2, then n is either contained in s_1 or it is contained in s_2.

1. Codify the "union-contains" property as a predicate. Write a test case that should pass (if your implementations work right) using the predicate you have defined.

```
;;> [1 pt -- contract & purpose]
;; union-contains : Set Set Number -> Boolean
;;> [3 pts -- code]
(define (union-contains s1 s2 n)
  (check-expect (s1 . union s2 . contains n)
                (or (s1 . contains n) (s2 . contains n))))
;;> [1 pt -- successful test]
(union-contains e (new singleton% 17) 17)
```

The University doesn't really trust Van Horn, and they want to make sure that `Sets` satisfy another property, called the "union-sum" property:

> Given two sets s_1 and s_2, computing the sum of s_1 and the sum of s_2 and then adding the two results is equal to the sum of the union of s_1 and s_2.

The University 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 "union-sum" 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]
;; union-sum : Set Set -> Boolean
;;> [3 pts -- code]
(define (union-sum s1 s2 n)
  (check-expect (s1 . union s2 . sum)
                (+ (s1 . sum) (s2 . sum))))

;;> [1 pt -- successful test]
;; successful test
(union-sum (new singleton% 17) e)
;;> [3 pts -- failing test]
;; It's false, counterexample:
(union-sum (new singleton% 17) (new singleton% 17))
```

Problem 3 Extra Credit 5 POINTS

Labich was unhappy to discover that the class languages don't have his favorite programming language feature—pattern matching. After thinking hard about this problem, he came up with a solution, which he calls *pattern objects*.

Consider the following data defintion:

```
;; A BTree is one of:
;; - (new leaf% Number)
;; - (new node% BTree Number BTree)
;; and implements
;; - match : [Pattern X] -> X
;;   applies the given pattern object to this tree

;; A [Pattern X] implements:
;; - leaf : Number -> X
;; - node : BTree Number BTree -> X

(define-class leaf%
  (fields n)
  ;; apply the appropriate method from the given pattern object
  (define (match p) (p . leaf (this . n))))

(define-class node%
  (fields left n right)
  ;; apply the appropriate method from the given pattern object
  (define (match p)
    (p . node (this . left) (this . n) (this . right))))
```

A pattern object is like the implementation of two different functions at once: one that implements a function for leaves from a binary tree, and one function that works on nodes.

Write a pattern object that computes the product of all of the numbers in a binary tree.

```
;;> [5 pts -- get it right]
;; product% implements [Pattern Number]
(define-class product%
  ;; the product of a leaf
  (define (leaf n) n)
  ;; the product of a node
  (define (node l n r) (* n (l . match this) (r . match this))))

(define p (new product%))
(check-expect ((new leaf% 5) . match p) 5)
(check-expect ((new node%  (new leaf% 2) 5 (new leaf% 7)) . match p) 70)
```