Spring, 1992 Name: ____________________

CS 342 — Principles of Programming Languages

# Smalltalk Test (Problem Solving, Version 2)

Before you begin, please take a moment to look over the entire test so that you can budget your time.
The usual directions for grading, etc. apply.
In your solutions you may use any classes and procedures in the Smalltalk library. These include the classes in the collection hierarchy and the number hierarchy, the class `Association`, and the classes in the simulation example. You may also use any of the variables and procedures defined below. Any other classes or procedures that you wish to use must be written in your solution. If you wish to use functions from your solutions in other parts of the exam, you must clearly state where we can find those function definitions.

```
(define +1 (x) (+ x 1))
(define or (x y) (if x x y))
(define and (x y) (if x y x))
(define not (x) (if x false true))
(define <> (x y) (not (= x y)))
(define <= (x y) (or (< x y) (= x y)))
(define divides (n x) (= (* n (/ x n)) x))
(define mod (n m) (- n (* m (/ n m))))
(define gcd (i j)
   (if (> i j) (gcd j i)
     (if (divides i j) i (gcd (mod j i) i))))
(define abs (x) (if (< x 0) (- 0 x) x))
;
(set false 0)
(set true 1)
(define isNil (x) (= x #nil))
(define notNil (x) (<> x #nil))
;
(define printColl (c)
  (printColl-aux c (first c)))
(define printColl-aux (c temp)
  (begin
    (while (notNil temp)
      (begin
        (print temp)
        (set temp (next c))))))
```

1. (15 points) Define a class `Triple` as a subclass of some existing class in the Smalltalk library. (That is, you may *not* make `Triple` a subclass of `Object`.) Objects of the class `Triple` can be thought of as a 3-tuple of objects. The protocol supported by `Triple` objects should be as follows:

   - **`fst`**, **`snd`**, and **`thrd`**, return the first, second, and third components (respectively) of a `Triple`.
   - **`fst:`**, **`snd:`**, and **`thrd:`**, set the first, second, and third components (respectively) to be the additional argument.

   Other protocol may be supported if you wish, but these are the only required methods. The global function **`mkTriple`** should take 3 arguments, which are (in order) the first, second, and third components of the new object.

   For example, consider the following session, where only the "interesting" results printed by the interpreter are shown. (That is, most outputs of <userval> and undefined outputs are omitted.)

```
-> (set tr (mkTriple 1 2 3))
<userval>
-> (fst tr)
1
-> (snd tr)
2
-> (thrd tr)
3
-> (fst: tr #smalltalk)
-> (fst tr)
smalltalk
-> (snd tr)
2
-> (snd: tr #is)
-> (snd tr)
is
-> (thrd: tr #fun)
-> (thrd tr)
fun
```

2. (20 points) Write a global function (not in any class) `common-elems` that takes two objects of class `Array` and returns a new `Array` object such that `(common-elems a1 a2)` is a Array that contains just the elements that are in both `a1` and `a2`, in some unspecified order. You may choose whatever number you like for the low bound of the resulting `Array`. The elements of `a1` and `a2` should remain unchanged. You may assume that `a1` and `a2` only contain symbols or numbers.

   For example, consider the following session, where only the "interesting" results printed by the interpreter are shown. (That is, outputs of <userval> and undefined outputs are omitted.) Note also that the order of outputs from `printColl` applied to the result of `common-elems` is undefined, so the order of your results may differ slightly in their order.

```
-> (set myArray (mkArray 1 3))
-> (set anArr (mkArray 1 2))
-> (at:put: myArray 1 #x)
-> (at:put: myArray 2 #x)
-> (at:put: myArray 3 #x)
-> (at:put: anArr 1 #x)
-> (at:put: anArr 2 #x)
-> (printColl myArray)
x
x
x
0
-> ;; The last 0 above is the return value of printColl

-> (set res1 (common-elems myArray anArr))
-> (printColl res1)
x
0

-> (at:put: anArr 1 #fun)
-> (set res2 (common-elems myArray anArr))
-> (printColl res2)
x
0

-> (printColl anArr)
fun
x
0

-> (set res3 (common-elems anArr myArray))
-> (printColl res3)
x
0

-> (printColl myArray)
x
x
x
0

-> (at:put: myArray 1 #fun)
-> (at:put: myArray 2 #is)
-> (at:put: myArray 3 #smalltalk)
```

```
-> (at:put: anArr 1 #fun)
-> (at:put: anArr 2 #smalltalk)
-> (set res4 (common-elems myArray anArr))
-> (printColl res4)
smalltalk
fun
0
-> ;; yours may be printed in a different order.
```