1 Introduction
You will explore some aspects of and write some simple classes and methods in the μSmalltalk language.

2 Description
- (Adapted from Exercise 12 of Chapter 10 from "Programming Languages: Build, Prove, and Compare" (p. 804).) Implement a class `Rand` having the following protocol.
  - Class method `fromSeed:` creates a new random-number generator in which the argument is the seed (an integer).
  - Class method `new` creates a new random-number generator with a default (but unspecified) seed.
  - Instance method `next` answers the next random number and updates the seed. By default, the sequence of random numbers should be generated as follows:
    \[ s_{i+1} = 9 \times s_i + 5 \mod 1024 \]
    where \( s_0 \) is the initial seed.
  - (extra) Class method `fromSeed:withAPRNG:` creates a new random-generator in which the first argument is the seed (an integer) and the second argument is an applicative pseudorandom number generator (a block of one argument), which determines the sequence of random numbers to be generated.

- (Adapted from Exercise 15 of Chapter 10 from "Programming Languages: Build, Prove, and Compare" (p. 805).) Add to class `Array` a class method `from: aCollection` that makes an array out of the elements of another collection.
  (Note: Simply give the implementation of the class method; there should be no need to otherwise change the implementation of class `Array`.)

The important methods from the class protocol of `Array` are:

| new: anInteger | Create and answer an array of size `anInteger` in which each element is `nil`. |

and the important methods from instance protocols of the `Collection` hierarchy are:

| at:put: key value | Modify the receiver by associating `key` with `value`. May add a new value of replace an existing value. Answer the receiver. (From the instance protocol for `KeyedCollection`.) |
| do: aBlock | For each element `x` of the collection, evaluate `(value aBlock x)`. (From the instance protocol for `Collection`.) |
| size | Answer how many element the receiver contains. (From the instance protocol for `Collection`.) |
• Recall the implementations of the instance methods size and addAll: of the class Collection:

```
(method size () (locals cnt)
  (set cnt 0)
  (do: self (block (x) (set cnt (+ cnt 1))))
  cnt)
(method addAll: (aCollection)
  (do: aCollection (block (x) (add: self x)))
  self)
```

- Implement the instance method size without using a block literal.
  (Hint: Although one typically uses a block literal to construct the argument of do:, do: simply requires that its argument is an object that “walks like a block and swims like a block and quacks like a block”)

- Implement the instance method addAll: aCollection without using a block literal.

- Comment on the feasibility and/or utility of abolishing block literals from μSmalltalk, from the perspective of a language designer/implementer and from the perspective of a language user.

• (Adapted from Exercise 18.5 of Chapter 10 from Programming Languages: Build, Prove, and Compare (p. 806).)

Recall the implementation of the instance method detect:ifNone: of the class Collection:

```
(method detect:ifNone: (aBlock exnBlock) (locals answer searching)
  (set searching true)
  (do: self (block (x)
    (ifTrue: (and: searching {((value aBlock x)})
      {{(set searching false)
        (set answer x)})})
    (if searching exnBlock {answer}))

Implement the instance method detect:ifNone: so as to avoid the final if.

3 Requirements and Submission

You may use the reference interpreter (see Appendix[A]), but there may only be one active laptop in each group.

At the end of class, submit the group’s solutions as hard-copy; be sure to include the names of all group members in the submission.

A Interpreter

A reference μSmalltalk interpreter is available on the CS Department Linux systems (e.g., glados.cs.rit.edu and queeg.cs.rit.edu and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/usmalltalk
```

Use the reference interpreter to check your code.
Solutions

- **class Rand**

  ```
  (class Rand Object
    (seed)
    (class-method new ()
      (withSeed: self 42))
    (class-method withSeed: (seed)
      (withSeed: (new super) seed))
    (method withSeed: (thatSeed)
      (set seed thatSeed) self)
    (method next ()
      (set seed (mod: (+ (* 9 seed) 5) 1024)))
  )

  (class Rand Object
    (seed aprng)
    (class-method new ()
      (withSeed: self 42))
    (class-method withSeed: (seed)
      (withSeed:withAPRNG: self seed (block (s) (mod: (+ (* 9 s) 5) 1024))))
    (class-method withSeed:withAPRNG: (seed aprng)
      (withSeed:withAPRNG: (new super) seed aprng))
    (method withSeed:withAPRNG: (thatSeed thatAPRNG)
      (set seed thatSeed) (set aprng thatAPRNG) self)
    (method next ()
      (set seed (value aprng seed)))
  )
  ```

  Note the use of (new super), rather that (new Rand), to create the new instance of the Rand class. This avoids an infinite loop, as the Rand class has a class-method new.

- **class method Array.from: aCollection**

  ```
  (class Array Array
    ()
    (class-method from: (aCollection) (locals a i)
      (set a (new: self (size aCollection)))
      (set i 0)
      (do: aCollection (block (x)
        (at:put: a i x)
        (set i (+ i 1)))
      a)
  )
  ```
• living without block literals
  – instance method Collection.size
    (class CntBlock Block
      (cnt)
      (class-method new () (locals cnt)
        (withCnt: (new super) 0))
      (method withCnt: (thatCnt)
        (set cnt thatCnt) self)
      (method value (_) (set cnt (+ cnt 1)))
      (method get () cnt))
    )
    (class Collection Collection
      ()
      (method size () (locals cnt)
        (set cnt (new CntBlock))
        (do: self cnt)
        (get cnt))
      )
  – instance method Collection.addAll: aCollection
    (class AddAllBlock Block
      (coll)
      (class-method withColl: (coll)
        (withColl: (new super) coll))
      (method withColl: (thatColl)
        (set coll thatColl) self)
      (method value (x) (add: coll x))
    )
    (class Collection Collection
      ()
      (method addAll: (aCollection)
        (do: aCollection (withColl: AddAllBlock self))
        self)
      )
  – For the language designer/implementer, the \(\mu\)Smalltalk semantics and implementation would be slightly simplified if block literals were abolished. It would simplify the semantics, by eliminating the CLOSURE primitive representation for blocks, the rule for evaluating a block literal, and the special-case rules for sending the message value to an object. (Indeed, the operational semantics of \(\mu\)Smalltalk as presented in (the current draft of) Programming Languages: Build, Prove, Compare does not admit the above implementations, since the operational semantics does not allow the value message to be answered by a user-defined method, though the interpreter does.) It would similarly simplify the implementation, by eliminating the parsing of block literals and various cases in the evaluator; however, the value message would remain a special case when checking the arity of method definitions and send expressions.

For the language user, programming in \(\mu\)Smalltalk would be awkward and cumbersome if block literals were abolished. The above implementations demonstrate a general pattern: for each eliminated block literal, introduce a new class with instance variables that correspond to the free variables of the block literal and initialize with the free variables of the block literal. Of course, there are some subtleties. Note that the last use of cnt in the original implementation of size is translated to a get cnt; after the execution of an eliminated block literal, the values of any instance variables set by the value method of the proxy object should be reflected back to the corresponding free variables via updates. The major difficulty with automating this elimination is that where and when a block object is executed is distinct from where and when a block object is created (cf. control-flow analysis in functional programming languages and points-to analysis is object-oriented programming languages); hence, it may be difficult or impossible to know where to insert such updates. (The implementations above “work” because we know
that the created block object is executed by the do: method and is not stored away to be executed later; note, however, that this required knowing something about the do: method in addition to the knowing the block literal to be eliminated.) The nuclear option would be to modify all uses and sets of any variable captured by a block literal to go through a wrapper object that introduces a level of indirection.

In any case, manually building closures is awkward and cumbersome. Witness the addition of anonymous classes, inner classes, and lambdas to Java to simplify the definition and use of such one-off classes.

In addition to the issues described above, there are two additional features of Smalltalk that complicate the elimination of block literals. In the µSmalltalk and Smalltalk-80 languages, it is not clear that the use of the super receiver in a block literal can be (easily) handled. Similarly, in the Smalltalk-80 language, the use of a non-local return in a block literal can not be (easily) handled.

- detect:ifNone: without if

```lisp
(method detect:ifNone: (aBlock exnBlock) (locals answer searching)
  (set searching true)
  (do: self (block (x)
    (ifTrue: (and: searching {((value aBlock x)})
      {(set searching false)
        (set exnBlock {x})})
      )
    (value exnBlock)))
```