1 Introduction

In this programming assignment, you will implement a number of classes and methods in \( \mu \)Smalltalk in order to gain familiarity with the language and to practice object-oriented programming.

Download \texttt{prog06.smt}, \texttt{prog06_tests.smt} and \texttt{prog06_tests.soln.out} (or copy from /usr/local/pub/mtf/plc/programming/prog06-smalltalk on the CS Department Linux systems). The first is a template for your submission and also includes a number of supporting classes. The second is a test suite for the assignment and the third is reference solution’s output on the test suite.

2 Description

This assignment investigates writing \( \mu \)Smalltalk classes that represent immutable, space-efficient vectors, which we call “xvectors”. Complete the definitions of the abstract class \texttt{XVector} and its concrete sub-classes \texttt{ArrayXVector}, \texttt{ConcatXVector}, \texttt{RepeatXVector}, \texttt{ReverseXVector}, \texttt{SwizzleXVector}, and \texttt{BlockXVector} to provide the protocols specified in Figures 1, 2, and 3. (Note: These classes represent space-efficient vectors. Hence, they should not unnecessarily allocate new data. The trade-off is that the \texttt{at:} method on xvectors may not be \( O(1) \).)

See Requirements and Submissions for important restrictions.
**XVector instance protocol**

### Display methods

**print**
Print the receiver on standard output; an xvector is printed as `<<`, the list of elements separated by spaces, followed by `>>`.

**debug**
Print a representation of the xvector on standard output; the representation is constructed from the name of the receiver’s class, an open parenthesis, the arguments used to construct the receiver (separated by commas), and a close parenthesis; any xvector arguments used to construct the receiver are printed using `debug`; non-xvector arguments used to construct the receiver are printed using `print`. (subclass responsibility) (10pts)

Note: The initial basis of the µSmalltalk interpreter includes global variables `space`, `newline`, `semicolon`, `quote`, `left-paren`, `right-paren`, `left-square`, `right-square`, `left-curly`, and `right-curly`, which are bound to objects of class `Char` that represent the space character, the new line character, the semicolon character “;”, the quote character “’”, the left parenthesis character “(”, the right parenthesis character “)”, the left square bracket character “[”, the right square bracket character “]”, the left curly brace character “{”, and the right curly brace character “}”. Such characters are useful for printing (send them the `print` message), but cannot be expressed using µSmalltalk’s literal symbol notation.

### Observer methods

**isEmpty**
Answer whether the receiver contains any elements. (like the corresponding `Collection` method)

**size**
Answer how many elements the receiver contains. (like the corresponding `Collection` method) (subclass responsibility) (10pts)

**at: anIndex**
Answer the element at position `anIndex`, or report the error `index-out-of-bounds` if the position `anIndex` is out of bounds. A non-negative position counts forward from the start of the xvector (i.e., `(at: xvector 0)` answers the first element); a negative position counts backward from the end of the xvector (i.e., `(at: xvector -1)` answers the last element).

**at:ifAbsent: anIndex exnBlock**
Answer the element at position `anIndex`, or answer `value exnBlock` if the position `anIndex` is out of bounds. (see `at:` method comments) (10pts)

**includes: anObject**
Answer whether the receiver contains `anObject`; uses `=` to compare `anObject` to elements. (like the corresponding `Collection` method)

**occurrencesOf: anObject**
Answer how many of the receiver’s elements are equal to `anObject`; uses `=` to compare `anObject` to elements. (like the corresponding `Collection` method)

**detect: aBlock**
Answer the first element `x` in the receiver for which `(value aBlock x)` is true, or report the error `no-object-detected` if none. (like the corresponding `Collection` method)

**detect:ifNone: aBlock exnBlock**
Answer the first element `x` in the receiver for which `(value aBlock x)` is true, or answer `value exnBlock` if none. (like the corresponding `Collection` method)

**sum**
Answer the sum of the elements in the receiver; assumes all elements are members of the same `Number` subclass and answers an `Integer` if the receiver is empty. (5pts)

**product**
Answer the product of the elements in the receiver; assumes all elements are members of the same `Number` subclass and answers an `Integer` if the receiver is empty. (5pts)

**min**
Answer the minimum element in the receiver, or report the error `min-of-empty` if the receiver is empty; assumes all elements answer messages of the `Magnitude` instance protocol. (5pts)

**max**
Answer the maximum element in the receiver, or report the error `max-of-empty` if the receiver is empty; assumes all elements answer messages of the `Magnitude` instance protocol. (5pts)

---

Figure 1: XVector instance protocol
XVector instance protocol (continued)

**iterator methods**
- `do: aBlock`
  For each element `x` in the receiver (in order of increasing position), evaluate `(value aBlock x)`. (like the corresponding Collection method) (10pts)

- `inject: into: thisValue binaryBlock`
  Evaluates `binaryBlock` once for each element in the receiver. The first argument of the block is an element from the receiver; the second argument is the result of the previous evaluation of the block, starting with `thisValue`. Answer the final value of the block. (like the corresponding Collection method)

**comparison methods**
- `similar anObject`
  Answers whether the receiver is similar to `anObject`; an xvector is not similar to an object that is not an instance of XVector and two xvectors are similar if they have the same size and elements of corresponding positions are similar. (like the corresponding Collection method) (10pts)

- `< anXVector`
  Answers whether the receiver is less than `anXVector`; xvectors are compared via lexicographic order; assumes all elements answer messages of the Magnitude instance protocol. (like the corresponding Magnitude method) (10pts)

- `> anXVector`
  Answers whether the receiver is greater than `anXVector`. (see < method comments; like the corresponding Magnitude method)

- `<= anXVector`
  Answers whether the receiver is no greater than `anXVector`. (see < method comments; like the corresponding Magnitude method)

- `>= anXVector`
  Answers whether the receiver is no less than `anXVector`. (see < method comments; like the corresponding Magnitude method)

- `min: anXVector`
  Answer the lesser of the receiver and `anXVector`. (see < method comments; like the corresponding Magnitude method)

- `max: anXVector`
  Answer the greater of the receiver and `anXVector`. (see < method comments; like the corresponding Magnitude method)

**producer methods**
- `* anXVector`
  Answer an xvector that represents the concatenation of the receiver and `anXVector`.

- `* anInteger`
  If `anInteger` is non-negative, answer an xvector that represents `anInteger` concatenations of the receiver. If `anInteger` is negative, report the error `negative-repeat`. (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. Note: Remember that these classes represent space-efficient vectors. An overriding implementation should not allocate more data than the abstract superclass implementation and should make the answered xvector more efficient for (some) operations than the xvector answered by the abstract superclass implementation. (bonus 3pts))

- `reverse`
  Answer an xvector that represents the reversal of the receiver. (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. (see * method comments) (bonus 3pts))

- `fromIndex:toIndex: aStartIndex anEndIndex`
  Answer an xvector that represents the elements of the receiver from position `aStartIndex` to position `anEndIndex` (inclusive). If position `aStartIndex` comes after position `anEndIndex` in the receiver, then the answered xvector has elements from the end of the receiver followed by elements from the start of the receiver (i.e., the slice “wraps around”). If either position `aStartIndex` or position `anEndIndex` are out of bounds, then report the error report the error `index-out-of-bounds`. (10pts)
  (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. (see * method comments) (bonus 3pts))

**private methods (internal to XVector classes)**
- `elem: anIndex`
  Answer the element at position `anIndex`; assumes that the position `anIndex` is positive and within bounds. (subclass responsibility) (10pts)

---

Figure 2: XVector instance protocol (continued)
### ArrayXVector class protocol

| withArr: anArray | Create and answer an xvector that represents the elements of anArray; since an xvector is immutable, the elements of anArray must be copied at the time of construction. |

### ConcatXVector class protocol

| withXV1:withXV2: anXVector1 anXVector2 | Create and answer an xvector that represents the concatenation of anXVector1 and anXVector2. (2pts) |

### RepeatXVector class protocol

| withXV:withN: anXVector anInteger | If anInteger is non-negative, create and answer an xvector that represents anInteger concatenations of anXVector. If anInteger is negative, report the error negative-repeat-count. (2pts) |

### ReverseXVector class protocol

| withXV: anXVector | Create and answer an xvector that represents the reversal of anXVector. (2pts) |

### SwizzleXVector class protocol

| withXV1:withXV2: anXVector1 anXVector2 | Create and answer an xvector that represents the swizzle of anXVector1 and anXVector2: the first element of the swizzle is the first element of anXVector1, the second element of the swizzle is the first element of anXVector2, the third element of the swizzle is the second element of anXVector1, the fourth element of the swizzle is the second element of anXVector2, and so on. If anXVector1 and anXVector2 are of unequal lengths, then the swizzle concludes with the excess elements from the rest of the longer one. (2pts) |

### BlockXVector class protocol

| withN:withBlock: anInteger aBlock | If anInteger is non-negative, create and answer an xvector that is of size anInteger and the element at position $i$ is obtained by (value aBlock $i$). aBlock may assume that it will only be evaluated with indices $i$ such that $0 \leq i < \text{anInteger}$. If anInteger is negative, report the error negative-block-size. (2pts) |

Figure 3: XVector sub-classes class protocols
3 Requirements and Submission

Your submission must be a valid $\mu$Smalltalk program. In particular, it must pass the following test:

```bash
$ cat prog06.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -q > /dev/null
```

without any error messages. If your submission produces error messages (e.g., syntax errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit `prog06.smt` to the Programming 06 Dropbox on MyCourses by the due date.

4 Hints

- You may (and should) add instance variables to the concrete sub-classes.
- You may define additional (private) helper methods.
- You may define additional classes.
A Interpreter

A reference $\mu$Smalltalk 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.

Source code for the interpreter is available on the CS Department file system at:

```
/usr/local/pub/mtf/plc/src/bare/usmalltalk
```

B Test Suite

Executing

```
$ cat prog06.smt prog06_tests.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -q > prog06_tests.out
```

will run the interpreter on the contents of the files prog06.smt and prog06_tests.smt (all tests) without prompts printed and save the output to the file prog06_tests.out; then executing

```
$ diff prog06_tests.soln.out prog06_tests.out
```

will compare the files prog06_tests.soln.out and prog06_tests.out and print any differences.

Similarly, executing

```
$ cat prog06.smt util.smt test-A-at:ifAbsent:.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -q > test-A-at:ifAbsent:.out
```

will run the interpreter on the contents of the files prog06.smt, util.smt, and test-A-at:ifAbsent:.smt (an individual test file) without prompts printed and save the output to the file test-A-at:ifAbsent:.smt.out; then executing

```
$ diff test-A-at:ifAbsent:.soln.out test-A-at:ifAbsent:.out
```

will compare the files test-A-at:ifAbsent:.soln.out and test-A-at:ifAbsent:.soln.out and print any differences. (For individual tests like test-La-size-ConcatXVector.smt, you will need to run the interpreter on prog06.smt, util.smt, test-Ja-init-ConcatXVector.smt, and test-La-size-ConcatXVector.smt; test-Ja-init-ConcatXVector.smt will create instances of the ConcatXVector class for testing with test-La-size-ConcatXVector.smt.)

Note: Due to the interdependencies between the classes and methods of the assignment, it is not easy to test individual pieces of functionality in isolation. You will probably find the test suite most helpful after you have a mostly completed assignment, when you can use the test suite to discover and diagnose any minor errors or missing corner cases. You will probably not find it helpful to use the test suite as the guiding force for completing the assignment.

The best suggestion is to use the system interactively to debug one method at a time. Note that the test output tries to print both the expression that it is evaluating and the expected answer. Thus, it might be most effective to use both your prog06.smt and the util.smt files and then copy-paste individual tests.