Smalltalk-in-Scheme: Adding Multiple Inheritance

In this exercise we will be adding support for multiple inheritance to the Smalltalk-like OO language implementation discussed in class. With this change the language should support class definitions like the following:

```
(define ClassA
  (CLASS (Root)
    (VAR varA 0)
    (METHOD getVarA ()
      (? varA))))

(define ClassB
  (CLASS (Root)
    (VAR varB 10)
    (METHOD getVarB ()
      (? varB))))

(define ClassC
  (CLASS (ClassA ClassB)))
```

Use the implementation of Smalltalk-in-Scheme with inheritance but without the Meta-Object Protocol (slides 44-49 of lecture 2) as the starting implementation.

1. Change the OO language so that it accepts a list of superclasses instead of just one superclass in a class specification: the language should still only support single inheritance though, just use the first class of the list as the superclass and ignore the others. Verify that with this change the language accepts the example class specifications given above.

(a) What did you have to change?

(b) Which messages can now be sent to an instance of ClassC?

(c) How many variables does an instance of ClassC have?
2. Now change the language so that a new class inherits the variable specifications of all the superclasses.
   (a) What did you have to change in the definition of the CLASS macro?

   (b) What other changes if any did you have to make?

   (c) What happens when a class inherits from two classes that both define a variable named $X$?

3. Change the language implementation so that in the example given above instances of ClassC will understand both the messages implemented in ClassA and those in ClassB.
   (a) What changes did you have to apply?
(b) What happens when you add a method to ClassC that does a SUPER send?

(c) How do you think should SUPER sending be changed to take multiple inheritance into account?

4. Consider the class hierarchies depicted in the diagram below.
(a) In your implementation, which methodX gets executed when a message methodX is sent to an instance of ClassG?

(b) In your implementation, which methodY gets executed when a message methodY is sent to an instance of ClassG?

(c) In your implementation, which methodZ gets executed when a message methodZ is sent to an instance of ClassG?

(d) What search strategy does your implementation use for its method lookup?

(e) What other strategies could you use that would change the answer to question (a), (b) or (c)?

(f) What are the advantages/disadvantages of the different strategies? (Hint: think of a more concrete class hierarchy with "printOn:" methods and what impact the method lookup has on how the objects would get printed)
For these questions you should use the OO language implementation which uses lexical addressing for variable access and method lookup. (slides 76-91 of lecture 2) Now consider the following class specifications:

(define ClassA
  (CLASS (Root)
    (VAR a 36)
    (VAR b 6)
    (METHOD sum ()
      (+ (? a) (? b)))))

(define ClassB
  (CLASS (Root)
    (VAR c 58)
    (VAR d 16)
    (METHOD difference ()
      (- (? c) (? d)))))

(define ClassC
  (CLASS (ClassA ClassB)))

(a) Think of ClassA and ClassB in the single inheritance inheritance case still for a moment. What offsets will be used for their variables a, b, c, and d respectively? Make a drawing of the memory layout of an instance of ClassA and an instance of ClassB.  
(Note: the memory layout of the instance = the Scheme vector that represents the instance)

(b) Now in the multiple inheritance case, what memory layout would you use for an instance of ClassC? Make a drawing again.
(c) How would you have to change the implementation of the «REPOSITORY»? (Give an explanation of the necessary and most important changes, you don't have to implement this)

(d) If you leave the other code unchanged, what will happen when you send a message *sum* to an instance of ClassC? What about a message *difference*?

(e) How will you change the variable access and method evaluation to work deal with multiple inheritance? (Again only give an explanation of the necessary changes, you don't have to implement this)

If you have time left, try actually implementing the multiple inheritance and lexical addressing combination.