Smalltalk-in-Scheme:
Adding Multiple Inheritance
(revised solution 2007-12-21)

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?
The super argument of the CLASS macro is now a list of names of variables that hold classes. To reduce this to single inheritance by ignoring all but the first of the class names we simply have to change the definition of the «SUPER» variable to take the car of the list in super:

```
(«SUPER» ,(car super))
```

(b) Which messages can now be sent to an instance of ClassC?
Only the message getVarA can be sent, not the message getVarB since the inheritance from ClassB for ClassC is currently ignored.

(c) How many variables does an instance of ClassC have?
One, not two, again because ClassC does not yet actually inherit from ClassB.
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?

We need to add a «SUPERS» variable to a Class to hold the list of all superclasses and change the initialization of «VARS» to be a merge of the variable tables of all the superclasses:

```
(...
(«SUPER» , (car super))
(«SUPERS» (list ,@super))
(«METHODS» («TABLE»))
(«VARS» (let ((merged («TABLE»)))
  (for-each (lambda (super)
      (super '«COPYINTO» merged))
    «SUPERS»)
  merged))
...)
```

The «COPYINTO» message can simply be implemented as:

```
((«COPYINTO»)
 («VARS» 'copyInto (car ARGS)))
```

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

An additional case for handling the copyInto operation on a «TABLE» needs to be added:

```
(copyInto)
  (let ((table (car rest)))
    (for-each
      (lambda (elt)
        (table 'put (car elt) (cdr elt)))
      tab))
```

The Root class needs to support the «COPYINTO» message as well which can be implemented as:

```
((«COPYINTO»)
 (car ARGS))
```

In the CLASS macro the following small replacement should also be done:

```
(context 'replace '«SELF» self) -> (context 'put '«SELF» self)
```

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

In this case the merging of the variable tables will fail, resulting in a "duplicate name" error.

3c 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.
What changes did you have to apply?
For this it best to separate the method lookup from the actual application of the method, thus we add the implementation for a message "GETMETHOD" to the CLASS macro:

```lisp
(("GETMETHOD")
 (let* ((msg (car ARGS))
        (entry ("METHODS" 'get msg)))
  (if entry
    entry
    (find (lambda (super)
          (super "GETMETHOD" msg))
         "SUPERS")))))
```

Where `find` is a function defined as follows:

```lisp
(define (find condition list)
  (if (null? list)
      #f
      (let ((result (condition (car list))))
        (if result
            result
            (find condition (cdr list))))))
```

The implementation of the "EVAL" message can then be changed to:

```lisp
(("EVAL")
 (let* ((context (car ARGS))
        (msg (cadr ARGS))
        (args (caddr ARGS))
        (entry ("CLASS" "GETMETHOD" msg)))
  (if entry
      (apply entry (cons context args))
      (error "method not found " msg))))
```

These messages also need to be provided in the Root class, with the following definitions:

```lisp
((?EVAL?)
 (let* ((msg (cadr ARGS))
        (error "method not found " msg)))
```
(b) What happens when you add a method to ClassC that does a SUPER send? If the «SUPER» variable has not yet been removed from the CLASS macro, the existing SUPER macro will still produce working code. But the resulting super message send will always start its method lookup in ClassA and will ignore ClassB entirely, there is therefore no way to override the getVarB method in ClassC while still allowing the overridden method to use the original getVarB method from ClassB.

(c) How do you think should SUPER sending be changed to take multiple inheritance into account? To solve the problem described in (b) the implementation of the SUPER macro can be changed so that it performs a method lookup that takes all the superclasses of a class into account. However this can also lead to problems when both ClassA and ClassB implement a method with the same name that we would like to override in ClassC. It is therefore most common in languages with multiple inheritance for super sends to be explicitly qualified with the name of the class where lookup has to start (e.g. C++), so that in our OO language a super send would look like:

(SUPER ClassB someMessage)

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

The one in Class\(D\).

(b) In your implementation, which method\(Y\) gets executed when a message method\(Y\) is sent to an instance of Class\(G\)?

The one in Class\(C\).

(c) In your implementation, which method\(Z\) gets executed when a message method\(Z\) is sent to an instance of Class\(G\)?

The one in Class\(D\).

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

Depth-first, left-to-right (preorder).

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

A breadth-first, left-to-right strategy would change the answer for (b) to the method method\(Y\) in class Class\(F\).

(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)

Usually, the greater the distance in the class hierarchy between a class and one of its superclasses the more abstract and less specialized the superclass is. A breadth-first-search-based method lookup minimizes the distance between the class of an object receiving a certain message and the superclass that actually has a method for that message. One can assume that that method will be more specialized and hence more 'appropriate' for handling that message for that object than a method that is farther removed.

5. 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:

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

(define ClassB
  (CLASS (Root)
    (VAR c 58)
    (VAR d 16))
```
(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)

Offset of variable a: 0, variable b: 1, variable c: 0, variable d: 1.

(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)

The declare operation should be changed so that the var-list that is used for a new class-entry consists of the variable lists of all the super classes appended together.

(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?

The method sum will evaluate correctly. A problem arises with the difference method however. When this method is created inside of the ClassB class specification, the ? macro will lookup the offset of variables varC and varD with respect to ClassB, having respectively offsets 0 and 1. When this method is however evaluated in the «CONTEXT» of an instance of ClassC it will incorrectly use the ClassC object's values of varA and varB!

(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)
One solution is to let subclasses have their own copy of the methods from their superclasses with the correct variable offsets, but this is memory intensive.

Another solution is to add another parameter like «CONTEXT» to the lambdas created for methods (see the METHOD macro): «VARBASEOFFSET». The method lookup and evaluation needs to also compute this base offset (this can be done at 'macro expansion time' of course) so that when a method of ClassB is evaluated with an instance of ClassB as «CONTEXT» the base offset is 0, and when the «CONTEXT» is an instance of ClassC the base offset is 2. The variable accessing macro's should take this base offset into account by adding it to the offset of the variable itself.