# Languages for object-oriented programming

**李昀烛 1300012798 信息科学技术学院**

## 目录



## Tutorial exercise 1

Louis 的错误在于他将语句的顺序调整之后，若有一个表达式需要检测，那么无论它是否是一个过程调用，他都会被当成一个过程调用来执行。如书上给的例子：

```
(define x 3)
```

eval 会认为 define 是一个过程，并在环境中寻找名为 define 的过程，试图去赋值，就会引发错误。

对于 Louis 的要求，我们只需要改变对 application 的判断函数：

```
(define (application? exp)
(tagged-list? exp ‘call))
```

和 operator、operands 的抓取函数：

```
(define (operator exp) (cadr exp))
(define (operands exp) (cddr exp))
```

便可以正常工作了。

## Tutorial exercise 2

```
; definition of the class <vector>
```

```
(define-class <vector> <object> xcor ycor)

; add a new method to generic function: *
(define-method * ((v1 <vector>) (v2 <vector>))
  (+ (* (get-slot v1 'xcor)
        (get-slot v2 'xcor))
     (* (get-slot v1 'ycor)
        (get-slot v2 'ycor))))

; add a new method to generic function: +
(define-method + ((v1 <vector>) (v2 <vector>))
  (make <vector>
        (xcor (+ (get-slot v1 'xcor)
                          (get-slot v2 'xcor)))
        (ycor (+ (get-slot v1 'ycor)
                 (get-slot v2 'ycor)))))

; add a new method to generic function: *
(define-method * ((v <vector>) (n <number>))
  (make <vector>
        (xcor (* (get-slot v 'xcor) n))
        (ycor (* (get-slot v 'ycor) n))))
(define-method * ((n <number>) (v <vector>))
  (make <vector>
        (xcor (* (get-slot v 'xcor) n))
        (ycor (* (get-slot v 'ycor) n))))

; define generic functions: square abs length
(define-generic-function square)
(define-generic-function abs)
(define-generic-function length)

; add a new method to generic function: square
(define-method square ((n <number>)) (* n n))

; add a new method to generic function: abs
(define-method abs ((n <number>))
  (cond ((< n 0) (- 0 n))
        (else n)))

; add a new method to generic function: length
(define-method length ((v <vector>))
  (sqrt (+ (square (get-slot v 'xcor))
           (square (get-slot v 'ycor)))))

(define-method length ((n <number>)) (abs n))

; several tests
(define v1 (make <vector> (xcor 2) (ycor 3)))
(define v2 (make <vector> (xcor 1) (ycor 2)))
(* v1 v2)                          ; value: 8
(get-slot (+ v1 v2) 'xcor)         ; value: 3
(get-slot (* v1 2) 'xcor)          ; value: 4
(get-slot (* 2 v1) 'ycor)          ; value: 6
(length v1)                        ; value: 3.605551275463989
(length -2)                        ; value: 2
```

# Tutorial exercise 3

paramlist-element-class 应该去调用 tool-eval，因为如果输入的是一个表达式的话，直接把它当作类型名称是不合适的，所以调用 tool-eval 之后，可以将其值计算出来，这样在定义的过程中就不用拘泥于具体的形式，而有灵活多样的定义方式。

# Tutorial exercise 4

在实现的过程中，首先是查看哪些是可以匹配的 method。（我们说"可以匹配"，是指实际应用的参数的类型，都是这个 method 初始设定类型的子类）

之后对这些可以匹配的 method 进行排序，如果 method_a 的所有参数的类都是 method_b 的子类，那么，我们认为 method_a 比 method_b 更加 specific，于是 method_a 就排在 method_b 之前，这也是给出代码中 method-more-specific?中定义的比较方式。

但是，这就有一个问题，对于多参数的 method，有可能其中的参数设置上可能出现：method_a 中的一个参数是 method_b 的子类，method_a 中的另一个参数则是 method_b 的父类，而这两个 method 都可以匹配，那么应该选哪一个呢？具体可能出问题的代码：

```
(define-class <a> <object>)
(define-class <b> <a>)
(define-generic-method check)
(define-method check ((a <a>) (b <b>)) 'check1)
(define-method check ((b <b>) (a <a>)) 'check2)
(define b (make <b>))
```

那么在运行(check b b)的时候，上面定义的两个方法都可以匹配，那么到底是应该输出 check1 还是 check2 呢？

经过实验，发现输出的是后者，交换定义的顺序之后，发现输出的还是后定义的 method，所以我发现与定义的顺序有关系。再经过查看代码，发现这种情况的产生主要还是在 sort 函数上面，在我重写了 sort 函数之后，发现完全可以通过控制比较的方法来控制匹配的 method。

# Tutorial exercise 5

```
; add a new method to generic function: print
(define-method print ((v <vector>))
  (print (cons (get-slot v 'xcor)
               (get-slot v 'ycor))))

; several tests
v1
; value: (2 . 3)
```

# Lab exercise 6

在我定义之前，TOOL 认为 v1 是一个从<vector>生成出来的实例：

```
v1 ; value: (instance of <vector>)
```

在我对 print 进行扩充之后，成功的打印出了 v1 的值，见 Tutorial exercise 5

# Lab exercise 7

我认为 generic function 应该被限制到 the global environment 中，因为：

1. **符合逻辑：**我们既然可以使用一个以他为名的 method，那么在全局中就应该有这样一个 function，不然如果全局中没有，凭空多出来的 method 实在让人不能接受。
2. **符合题目要求：**题目中遇到的问题在于希望方便的定义 method，而我们之所以会认为这件事变方便了，是因为我们跳过了步骤，而这个步骤在之前的操作过程中是限制在全局的，所以在补齐这个 function 的时候，当然应该也把它限制在全局。

在 eval-define-method 函数的本来的函数体前面加入：

```
(let ((name (method-definition-generic-function exp)))
     (if (variable? name)
         (let ((b (binding-in-env name env)))
           (if (or (not (found-binding? b))
                   (not (generic-function? (binding-value b))))
               (let ((val (make-generic-function name)))
                 (define-variable! name val env)
                 (display  (list  'defined  'generic  'function:
name))
                 (newline))))))

; several tests
(define-method check ((n <number>)) (+ n 1))
; value: (defined generic function: check)
;         (add method to generic function: check)
(check 4) ; value: 5
```

但如果改成限制在局部的话，第一次定义的输出和全局版本是一样的，但在之后每次给 check 定义新的 method，都会在它的局部环境中重新限制一个 check 的 function，就会多次输出(defined generic function: check)，如下：

```
; global version
(define-method check ((n <number>)) (+ n 2))
; note: this is the second definition
; value: (add method to generic function: check)

; local version
(define-method check ((n <number>)) (+ n 2))
; note: this is the second definition
; value: (defined generic function: check)
;         (add method to generic function: check)
```

# Lab exercise 8

使用 `for-each` 函数对每一个 `slot` 都进行操作，这里运用了一个 scheme 字符串构造的小 trick，反引号可以构造字符串模板，使得对命令的表达清晰简洁。并且添加解释性的文字，使得输出信息更加全面：

```
(define (eval-define-class exp env)
    (let ((superclass (tool-eval
                          (class-definition-superclass exp)
                          env)))
    (if (not (class? superclass))
      (error "Unrecognized superclass -- MAKE-CLASS >> "
             (class-definition-superclass exp))
      (let ((name (class-definition-name exp))
            (all-slots (collect-slots
                         (class-definition-slot-names exp)
                         superclass)))
        (let ((new-class
                (make-class name superclass all-slots)))
          (define-variable! name new-class env)
          (display (list 'defined 'class: name)) (newline)

          (for-each
               (lambda (slot-name)
                    (tool-eval
                       `(define-method ,slot-name ((obj ,name))
                    (get-slot obj ',slot-name)) env))
               all-slots)
          )))))

; several tests
(define-class <v> <object> x y)
; value: (defined-class: <v>)
;         (defined generic function: x)
;         (defined generic function: y)
(define v (make <v> (x 1) (y 2)))
(x v) ; value: 1
(y v) ; value: 2
```

# Lab exercise 9

我做的实验是：

1 定义复数类

  1.1 定义复数的加减乘除，以及取模运算

2 把每个复数当做负平面上的向量，从它派生出负平面上的线段，定义线段类

  2.1 定义点类，从它出发实现向量的生成

  2.2 定义线段取模运算

  2.3 定义叉积

  2.4 判断线段是否相交

```
;;; Lab exercise 9
```

```
;; define the complex class
(define-class <complex> <object> real imag)

; two instances
(define c1 (make <complex> (real 1) (imag 2)))
(define c2 (make <complex> (real 2) (imag 3)))

; basic operations
(define-method square ((n <number>)) (* n n))

(define-method * ((c1 <complex>) (c2 <complex>))
  (make <complex>
        (real (- (* (real c1) (real c2))
                         (* (imag c1) (imag c2))))
        (imag (+ (* (real c1) (imag c2))
                 (* (imag c1) (real c2))))))

(define-method + ((c1 <complex>) (c2 <complex>))
  (make <complex>
        (real (+ (real c1)
                         (real c2)))
        (imag (+ (imag c1)
                 (imag c2)))))

(define-method - ((c1 <complex>) (c2 <complex>))
  (make <complex>
        (real (- (real c1)
                         (real c2)))
        (imag (- (imag c1)
                 (imag c2)))))

(define-method length ((c <complex>))
  (sqrt (+ (square (real c))
           (square (imag c)))))

(define-method / ((c <complex>) (n <number>))
  (make <complex>
        (real (/ (real c) n))
        (imag (/ (imag c) n))))

(define-method / ((c1 <complex>) (c2 <complex>))
  (/ (* (make <complex>
              (real (real c2))
              (imag (- 0 (imag c2))))
        c1)
     (+ (square (real c2))
        (square (imag c2)))))

; print module
(define-method print ((c <complex>))
  (print (cons (real c) (imag c))))

; several tests
(* c1 c2)   ; value: (-4 . 7)
(+ c1 c2)   ; value: (3 . 5)
(- c1 c2)   ; value: (-1 . -1)
(length c1) ; value: 2.23606797749979
(/ c1 c2)   ; value: (8/13 . 1/13)
```

```
;; define the segment class
(define-class &lt;segment&gt; <complex> xcor ycor)

;; define the dot class
(define-class <dot> <object> xcor ycor)

; length
(define-method length ((s &lt;segment&gt;))
  (length (make <complex>
                (real (real s))
                (imag (imag s)))))

; several tests
(define s1 (make &lt;segment&gt; (real 1) (imag 2) (xcor 1) (ycor 1)))
(length s1) ; value: 2.23606797749979

; print module
(define-method print ((d <dot>))
  (print (cons (xcor d) (ycor d))))
(define-method print ((s &lt;segment&gt;))
  (print (cons 'complex (cons (real s) (imag s))))
  (print (cons 'dot (cons (xcor s) (ycor s)))))

; basic operation
(define-method - ((d1 <dot>) (d2 <dot>))
  (make <complex>
        (real (- (xcor d1)
                          (xcor d2)))
        (imag (- (ycor d1)
                 (ycor d2)))))

(define-method cross-product ((c1 <complex>) (c2 <complex>))
  (- (* (real c1) (imag c2))
     (* (imag c1) (real c2))))

(define-method  seperate-side?  ((s  &lt;segment&gt;)  (d1  <dot>)  (d2
<dot>))
  (define s1 (make <dot> (xcor (xcor s)) (ycor (ycor s))))
  (define s2 (make <dot> (xcor (+ (xcor s) (real s)))
                   (ycor (+ (ycor s) (imag s)))))
  (define result (* (cross-product
                     (- d1 s1)
                     (- s2 s1))
                    (cross-product
                     (- d2 s1)
                     (- s2 s1))))
  (cond ((< result 0) #t)
        ((= result 0) #t)
        (else #f)))

(define-method seg-cross? ((s1 &lt;segment&gt;) (s2 &lt;segment&gt;))
  (define d11 (make <dot>
                    (xcor (xcor s1)) (ycor (ycor s1))))
  (define d12 (make <dot>
                    (xcor (+ (xcor s1) (real s1)))
                    (ycor (+ (ycor s1) (imag s1)))))
```

```
  (define d21 (make <dot>
                    (xcor (xcor s2)) (ycor (ycor s2))))
  (define d22 (make <dot>
                    (xcor (+ (xcor s2) (real s2)))
                    (ycor (+ (ycor s2) (imag s2)))))
  (define result1 (seperate-side? s1 d21 d22))
  (define result2 (seperate-side? s2 d11 d12))
  (cond (result1
         (cond (result2 #t)
               (else #f)))
        (else #f)))

; several tests
(define s (make &lt;segment&gt; (real 1) (imag 2) (xcor 1) (ycor 1)))
(define s1 (make &lt;segment&gt; (real 1) (imag 0) (xcor 0) (ycor 0)))
(define s2 (make &lt;segment&gt; (real 1) (imag -1) (xcor 1) (ycor 2)))
(define s3 (make &lt;segment&gt; (real 0) (imag 1) (xcor 2) (ycor 0)))
(length s)         ; value: 2.23606797749979
(seg-cross? s s2)  ; value: #t
(seg-cross? s s1)  ; value: #f
(seg-cross? s2 s3) ; value: #t
```

## Post-lab exercise 10

这道题我没有具体的去写代码，这里仅仅只是谈一谈一些小的想法：

我们可以看见，虽然在如 c++一类的面向对象的语言中确实存在着多重继承，但是在时下流行的语言，如 Python、ruby 中却不支持，究其原因，无非两点：没有必要和容易产生歧义性。

我们说 C 从 A 和 B 继承过来，那么自然的可以说“C 是 A，并且 C 也是 B”，当我们回归到现实，发现这种情况很容易出现在职业和社会角色上面，我们可以说一个人“既是一个医生，又是一个母亲”，这两种身份并没有互相包含的意味，但却可以同时在一个人身上得到体现，但是如果在面对生活中的种种事件的时候，两种身份所要求的选择很有可能是不一样的，那么究竟应该采取何种行动就是多重继承的歧义性所在。

为了解决这个问题，我所想到的解决方法就是给这些“身份”定一个优先级，谁是你的主要身份，谁是次要身份，这样就可以在面对歧义性的时候给出应该有的行为。当然，在某些情况下，可能产生主要身份的变更，这些情况就需要进一步的讨论了。

## Notification

1. 上述代码中的定义和测试部分都可以在 test.scm 文件中找到，本地使用的环境是 mit-scheme，所有代码均已全部通过测试。
2. 对于 Lab exercise 7 和 Lab exercise 8 的任务，已经在提交的 mod.scm 中进行修改，也已通过本地测试。