面向对象的程序设计语言

信息科学技术学院 王迪（学号：1300012802）

2014 年 6 月 7 日

Contents

1 Tutorial exercise 2

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

(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)))))

(define-method * ((v1 <vector>) (v2 <vector>))
  (+ (* (get-slot v1 'xcor)
        (get-slot v2 'xcor))
     (* (get-slot v1 'ycor)
        (get-slot v2 'ycor))))

(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 (* n
                          (get-slot v 'xcor)))
                 (ycor (* n
                          (get-slot v 'ycor)))))

(define-generic-function length)
(define-method length ((o <object>))
  (sqrt (* o o)))
```

2 Tutorial exercise 3

`paramlist-element-class`应该调用`tool-eval`，因为类名不一定以常量符号即形如`<object>`给出，可能是一个合法表达式，需要对其求值。这样一来，我们在`define-method`时便获得了更大的灵活性。

3 Tutorial exercise 4

首先，解释器发现`say`是一个 generic function，于是通过`generic-function-methods`获取了该 function 的所有 methods，一共有 3 个。然后因为`fluffy`是`<house-cat>`而非`<show-cat>`，所以会过滤掉 1 个，传给排序的 methods 其实只有 2 个：

1. `say ((cat <cat>) (stuff <object>))`
2. `say ((cat <cat>) (stuff <number>))`

按照`method-more-specific?`谓词排序后，第 2 个 method 获得了较高的优先级，所以就调用了它。

4 Tutorial exercise 5

```
(define-method print ((v <vector>))
  (print (cons
          (get-slot v 'xcor)
          (get-slot v 'ycor))))
```

5 Lab exercise 6

在为`<vector>`定义`print`之前：

```
TOOL==> (define v (make <vector> (xcor 1) (ycor 5)))
*undefined*

TOOL==> v
(instance of <vector>)
```

定义了`print`后：

```
TOOL==> (define v (make <vector> (xcor 1) (ycor 5)))
*undefined*

TOOL==> v
(1 . 5)
```

6 Lab exercise 7

我认为新的 generic function 应该限制在当前的 eval 环境中，而不是放进全局框架里。

- 第一，从代码规范上来讲，如果一个 generic function 在全局范围内有作用，那么它应该显式地在全局进行定义，而不是在某个过程中被`define-method`隐式定义；
- 第二，从作用域上来讲，局部定义的 generic function 只在局部起作用，不仅合乎逻辑，也防止了局部的 function 名称污染全局环境；
- 第三，从效率上来讲，这样做提高了局部 method 寻找的效率，某种程度上也方便垃圾回收（一般来说，过程完成后，局部框架会回收，而因为加入的 generic function 与其他环境框架无关，所以也可以被回收）。

下面是一个例子：

```
(define-method test ()
  (define-method method-in-test ((n <number>))
    (+ n 1)))

(test)
(method-in-test 1)
```

在我的修改版本中，最后一行调用会引发一个变量未约束的错误，而若是将 generic function 定义在了全局范围，最后一行调用则能成功，且返回值为 2。

我在过程`eval-define-method`中添加了如下代码：

```
(let ((var (method-definition-generic-function exp)))
  (if (variable? var)
    (let ((b (binding-in-env var env)))
      (if (or
            (not (found-binding? b))
            (not (generic-function? (binding-value b))))
        (let ((val (make-generic-function var)))
          (define-variable! var val env))))))
```

下面是一些测试：

```
TOOL==> (define-method inc ((n <number>)) (+ n 1))
(added method to generic function: inc)

TOOL==> (inc 5)
6

(define-method inc ((l <list>)) (cons 1 l))
(added method to generic function: inc)

TOOL==> (inc '(1 2 3))
(1 1 2 3)
```

7 Lab exercise 8

直接调用`tool-eval`实现，且基于了上一题的结果。在`eval-define-class`最后返回值前加入了如下代码：

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

代码第 4 行最左端是一个反引号。

下面是一些测试：

```
TOOL==> (define-class <person> <object> name sex)
(defined class: <person>)

TOOL==> (define me (make <person> (name 'wayne) (sex 'male)))
*undefined*

TOOL==> (name me)
wayne

TOOL==> (sex me)
male
```

8 Lab exercise 9

首先是一些关于`<vector>`的例子：

```
TOOL==> (define-class <vector> <object> xcor ycor)
(defined class: <vector>)

TOOL==> (define-method print ((v <vector>))
          (print (cons (xcor v) (ycor v))))
(added method to generic function: print)

TOOL==> (define-method + ((v1 <vector>) (v2 <vector>))
          (make <vector>
                (xcor (+ (xcor v1) (xcor v2)))
                (ycor (+ (ycor v1) (ycor v2)))))
(added method to generic function: +)

TOOL==> (define v1
          (make <vector>
                (xcor (make <vector> (xcor 1) (ycor 5)))
                (ycor 4)))
*undefined*
TOOL==> (define v2
          (make <vector>
                (xcor (make <vector> (xcor -2) (ycor 2)))
                (ycor -1)))
*undefined*

TOOL==> (+ v1 v2)
((instance (class <vector> ((class <object> () ())) (xcor ycor)) (-1 7)) . 3)

TOOL==> (ycor v2)
-1

TOOL==> (xcor v1)
(1 . 5)
```

然后从<vector>类派生了<3d-vector>类：

```
TOOL==> (define-class <3d-vector> <vector> zcor)
(defined class: <3d-vector>)

TOOL==> (define v3
          (make <3d-vector>
                (xcor (make <vector> (xcor -1) (ycor 3)))
                (ycor 2)
                (zcor -3)))
*undefined*

TOOL==> (zcor v3)
-3

TOOL==> (xcor v3)
(-1 . 3)
```

对 generic function 的调用进行了测试：

```
TOOL==> (+ v1 v3)
((instance (class <vector> ((class <object> () ())) (xcor ycor)) (0 8)) . 6)

TOOL==> (define-method + ((v1 <vector>) (v2 <3d-vector>))
          (make <3d-vector>
                (xcor (+ (xcor v1) (xcor v2)))
                (ycor (+ (ycor v1) (ycor v2)))
                (zcor (+ (zcor v2) 100))))
(added method to generic function: +)

TOOL==> (define-method print ((v <3d-vector>))
          (print (cons (xcor v) (cons (ycor v) (zcor v)))))
(added method to generic function: print)

TOOL==> (+ v1 v3)
((instance (class <vector> ((class <object> () ())) (xcor ycor)) (0 8)) 6 . 97)
```

可以看到7、8两个练习中的修改都工作得很好。

接下来是对<cat>类的测试。

```
TOOL==> (define-class <cat> <object> size breed)
(defined class: <cat>)

TOOL==> (define garfield (make <cat> (size 6) (breed 'weird)))
*undefined*

TOOL==> (breed garfield)
weird

TOOL==> (define-method 4-legged? ((x <cat>)) true)
(added method to generic function: 4-legged?)

TOOL==> (define-method 4-legged? ((x <object>)) 'Who-knows?)
(added method to generic function: 4-legged?)

TOOL==> (4-legged? garfield)
#t

TOOL==> (4-legged? 'Hal)
who-knows?
```

看到4-legged?谓词根据对象类型进行了正确的选择。

```
TOOL==> (define-method say ((cat <cat>) (stuff <object>))
  (print ’meow)
  (print stuff))
(added method to generic function: say)

TOOL==> (define-method say ((cat <cat>) (number <number>))
  (print ’i-do-not-recognize-numbers)
  (print ’ooooooooooooooooooooooooops))
(added method to generic function: say)

TOOL==> (say garfield ’(feed me))
meow
(feed me)
#!unspecific

TOOL==> (say garfield 563)
i-do-not-recognize-numbers
ooooooooooooooooooooooooops
#!unspecific
```

say方法也根据参数类型选择了最为匹配的过程执行。

```
TOOL==> (define-class <house-cat> <cat> address)
(defined class: <house-cat>)

TOOL==> (define fluffy (make <house-cat> (size ’tiny) (address ’America)))
*undefined*

TOOL==> (breed fluffy)
*undefined*

TOOL==> (size fluffy)
tiny

TOOL==> (address fluffy)
america

TOOL==> (say fluffy ’(feed fluffy))
meow
(feed fluffy)
#!unspecific
```

可以看到<house-cat>类正确地继承了<cat>类的属性和方法。

```
TOOL==> (define-method say ((cat <house-cat>) (stuff <object>))
  (print 'i-am-a-house-cat)
  (print stuff)
  (print 'i-am-specail!))
(added method to generic function: say)

TOOL==> (say garfield 'pardon?)
meow
pardon?
#!unspecific

TOOL==> (say fluffy 'pardon?)
i-am-a-house-cat
pardon?
i-am-specail!
#!unspecific
```

对`<house-cat>`类重写了`say`方法，发现改变了`<house-cat>`实例的行为，而`<cat>`的实例的行为不变。

```
TOOL==> (say fluffy 443)
i-am-a-house-cat
443
i-am-specail!
#!unspecific
```

这是测试TOOL在`say <cat> <number>`与`say <house-cat> <object>`之间的取舍。行为跟预期相符：选择了后者，因为考察第一个参数的类型，`<house-cat>`要比`<cat>`更加匹配一点。和下面的测试做对比：

```
TOOL==> (define-method shout ((stuff <object>) (cat <house-cat>))
  (print 'shout!)
  (print stuff))
(added method to generic function: shout)

TOOL==> (define-method shout ((number <number>) (cat <cat>))
  (print 'shout-number!)
  (print number))
(added method to generic function: shout)

TOOL==> (shout 43 fluffy)
shout-number!
43
#!unspecific

TOOL==> (shout 'you-are-great! fluffy)
shout!
you-are-great!
#!unspecific
```

可见在方法调用时排在前面的参数类型的确要重要一些。

以上测试代码均在`test.scm`文件中。

9 总结

9.1 完成工作

按照项目说明文件一步一步做下来的，所做的工作在前面的部分已经详细说明了。

9.2 基本设计

考虑 TOOL 语言本身，它是数据导向的，通过良好设置的抽象屏障，使得对 TOOL 进行功能扩充非常方便，不用深入底层数据结构。

但是在有些时候还是需要去查看底层数据结构，比如我做 Lab exercise 7 时我查看了 binding 的实现方式，这是因为底层 API 并不完备，缺少一个询问变量是否存在的谓词。这可以通过补充 API 很好的解决。

9.3 所遇问题

基本上没遇到什么大问题，项目说明很清楚。

9.4 如何改进

Lab 10 是一个关于多重继承的语言设计，但我还没有一个完整的想法，故并未写在作业中。多继承在现代编程语言中似乎并不多见，所以我在考虑能否像 Java 或者 Ruby 那样，添加接口或者模块的功能。另外我对 generic-function 的想法有一些疑惑，总觉得一个类的实例的方法应该只属于这个类的实例，在全局出现一个类似索引的东西，感觉不是特别优美。总而言之，还是有很多可以改进的地方，需要更深入的思考。