Brian Cullinan

Principles of Languages

Assignment 3 - Beauty, thy name is OOPS

Bjc76

Project 3 - The objective of this project was to obtain a higher understanding of Objects and implement objects in scheme.

Part 1:

I accomplished objects by compiling and evaluating lists. I first part out each section of the class definition, then I compile the class into a list that can be evaluated.

For my caller function it just calls \*this\* with same arguments.

\*this\* does the only processing because it finds the method with the same name and same number of arguments, then it executes that method.

Part 2:

I accomplished inheritance by copying the methods and variables out of the parent class. The methods are merged with the child class, so if there is a parent of a parent it will still use all the methods.

The parent methods get replaced by the child methods if they have the same name and same number of arguments.

The most important functionality is the **class**, this is where most of the processing happens. It compiles the list together so that it can be evaluated.

**Get-method** searches through a list of method for the specified name and number of arguments.

**Merge-methods** combine two lists of methods skipping the ones that the child class overrides.

**Get-definition** extracts the definition from the class so (parent: ) and (methods: ) can be in any order.

**New** executes the \_\_new function in the class definition.

**Output:**

Here is what my class definition looks like before it is evaluated, I spaced it out and added some more comments just to clarify a few things:

(define pet (letrec ((\*this\* (eval (lambda args ; this creates the main function for a class

(let ((method (get-method \_\_methods (car args) (count (cdr args)))))

(if (null? method) ; get method

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args)))))))

(\_\_methods `((\_\_new 1 ,(lambda (petname) ; the functions that belong to the class definition

(eval (letrec ((weight (random 50))

(name petname)

(friendly (random 2))

(\*this\* (eval (lambda args (let ((method (get-method \_\_methods (car args) (count (cdr args)))))

(if (null? method)

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args)))))))

(\_\_methods `((describe 0 ,(lambda ()

(begin

(display "My name is" )

(display name)

(display "; I'm all of" )

(display weight)

(display " kilos of generic pet protoplasm and I'm permanently" )

(if (eqv? friendly 0) (display "vicious! Grrr!") (display "happy! Let's play"))

(newline))))

(feed 1 ,(lambda (weight)

(begin

(\*this\* 'setweight (+ (\*this\* 'getweight) weight))

(display "gloomf! Food absorbed!")

(newline))))

(setweight 1 ,(lambda (newval) (set! weight newval)))

(getweight 0 ,(lambda () weight)))))

(lambda args (apply \*this\* args)))))) ; pass to \*this\*

(\_\_get-methods 0 #<procedure>) ; returns the methods for this class

(\_\_get-ivars 0 #<procedure>) ; returns the variables for the class

)))

(lambda args (apply \*this\* args)))) ; pass to \*this\*

All the execution outputs exactly what yours does in the examples.

**CODE:**

;; load-file is a simple function that loads the definitions from a file

;; It recursives calls itself, reading one line at a time; on the recursive return, it cons'es

;; all of the lines together in a giant list, which it returns to caller.

(require (lib "include.ss") )

( define load-file

( lambda ( port )

( let ( ( nextrec ( read port ) ) )

( cond

( ( eof-object? nextrec ) '() ) ;; If I've read off the end, return empty list

( else

( let\* ( ( nascent-db ( load-file port ) ) ) ;; Recursive call to finish reading file

;; Now add the line read at this level to growing list

( cons nextrec nascent-db ) ) ) ) ) ) )

; get the specified part of the class definition

(define get-definition

(lambda (args name)

(if (null? args)

'()

(if (equal? (caar args) name)

(apply (eval (caar args)) `(,(cdar args)))

(get-definition (cdr args) name)

))))

; merge methods together, used for merging the parent to the child class

(define merge-methods

(lambda (parent-methods child-methods)

;(display parent-methods)

(if (null? parent-methods)

child-methods

(if (null? (car parent-methods))

(if (null? (cdr parent-methods))

child-methods

(merge-methods (cdr parent-methods) child-methods)

)

(if (null? (get-method child-methods (caar parent-methods) (cadar parent-methods)))

; add the parent method to child methods

(cons (car parent-methods) (merge-methods (cdr parent-methods) child-methods))

(merge-methods (cdr parent-methods) child-methods)

)))))

; merge ivars

(define merge-ivars

(lambda (parent-ivars child-ivars)

(if (null? parent-ivars)

child-ivars

(if (null? (car parent-ivars))

(if (null? (cdr parent-ivars))

child-ivars

(merge-ivars (cdr parent-ivars) child-ivars)

)

(if (null? (get-ivar child-ivars (caar parent-ivars)))

; add the parent method to child methods

(cons (car parent-ivars) (merge-ivars (cdr parent-ivars) child-ivars))

(merge-ivars (cdr parent-ivars) child-ivars)

)))))

; merge the second list onto the end of the first list

(define merge

(lambda (first second)

;(display second)

(if (null? first)

second

(cons (car first) (merge (cdr first) second))

)))

; this is our class creator

(define class

(lambda args

; call all the parts that define a class

(let ((return-val '())

(class-name (car args))

(parent (get-definition (cdr args) 'parent:))

(constructor-args (get-definition (cdr args) 'constructor\_args:))

(ivars (get-definition (cdr args) 'ivars:))

(methods (get-definition (cdr args) 'methods:))

(static (get-definition (cdr args) 'static:))

(constructor '())

(get-methods '())

(get-ivars '())

(this `(eval (lambda args ,\_\_call)))

(this-static `(eval (lambda args ,\_\_call-static)))

)

; merge the methods if there is a parent

(if (null? parent)

'()

(let ()

(set! methods (merge-methods (apply (eval (car parent)) '(\_\_get-methods)) methods))

(set! ivars (merge-ivars (apply (eval (car parent)) '(\_\_get-ivars)) ivars))

)

)

; set the get-methods to return the methods for an instantiated class

(set! get-methods `(\_\_get-methods 0 ,(eval (lambda () methods))))

(set! get-ivars `(\_\_get-ivars 0 ,(eval (lambda () ivars))))

; get the constructor method for creating new instances

(set! constructor `(\_\_new ,(count constructor-args) ,(cons 'unquote `(

(lambda ,constructor-args

(eval (letrec ,(merge ivars (list

`(\*this\* ,this)

`(\_\_methods ,(list 'quasiquote methods))

))

(lambda args (apply \*this\* args))

)))))))

; add constructor and \_\_get-methods to static methods

(set! static (list constructor get-methods get-ivars static))

;(display constructor)

; THIS ALLOWS FOR A STATIC SECTION

(set! return-val (list 'define class-name `(letrec ,(list `(\*this\* ,this-static)

`(\_\_methods ,(list 'quasiquote static))

)

;(display (apply \*this\* '(\_\_get-methods)))

(lambda args (apply \*this\* args))

)))

;(display return-val)

(newline)

(eval return-val)

)))

; this is the dispatcher for static objects

(define \_\_call-static

; make this a list so the class creator can import this code into the class

'(let ( ; get called method from methods

(method (get-method \_\_methods (car args) (count (cdr args))))

)

; if the get-method returns '() then no method was found

;(display args)

(if (null? method)

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args))

)

))

; this is our function for distributing function calls

(define \_\_call

; make this a list so the class creator can import this code into the class

'(let (

(method (get-method \_\_methods (car args) (count (cdr args))))

)

;(display (cdr args))

(if (null? method)

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args))

)

;(display args)

))

; this is out object factory that creates local objects

(define new

(lambda args

; rebuild the object with a constructor method and replace dispatcher

(apply (car args) (cons '\_\_new (cdr args)))

))

; get an ivar by name

(define get-ivar

(lambda (ivars name)

(if (null? ivars)

'()

(if (null? (car ivars))

(if (null? (cdr ivars))

'()

(get-ivar (cdr ivars) name)

)

(if (equal? (caar ivars) name)

(car ivars)

(get-ivar (cdr ivars) name)

)))))

; recursevly loop through methods and find the one that matches

(define get-method

; take a methods list of all the methods, the name of the method being searched for, and the count for how many arguments the method has

(lambda (methods name count-args)

;(display methods)

(if (null? methods)

'()

(if (null? (car methods))

(if (null? (cdr methods))

'()

(get-method (cdr methods) name count-args)

)

(if (list? (cadar methods))

(if (and (equal? (caar methods) name) (equal? (count (cadar methods)) count-args))

(car methods)

(if (null? (cdr methods))

'()

(get-method (cadr methods) name count-args)

))

(if (and (equal? (caar methods) name) (equal? (cadar methods) count-args))

(car methods)

(if (null? (cdr methods))

'()

(get-method (cdr methods) name count-args)

)))))))

; count the number of top level elements

(define count

(lambda (list)

(if (null? list)

0

(+ 1 (count (cdr list)))

)))

; this gets the parent class for another class to use

(define parent:

(lambda (args)

args

))

; this gets the constructor arguments in a list form

(define constructor\_args:

(lambda (args)

args

))

; this is the let list of local variables for an object

(define ivars:

(lambda (args)

args

))

; returns the list of methods

(define methods:

(lambda (args)

(if (null? args)

'()

(cons (list (caar args) (count (cadar args)) (list 'unquote `(lambda ,(cadar args) ,(caddar args))) ) (methods: (cdr args)))

)))

; loads a file of class definitions into a list

(define load-classes

(lambda (file)

(load-class (load-file (open-input-file file)))

(display "")

)

)

; loads a single class at a time recursively

(define load-class

(lambda (classes)

(if (null? classes)

'()

(if (equal? (caar classes) 'define)

(eval (car classes))

(let ()

(apply (eval (caar classes)) (cdar classes))

(display "Defined class:")

(display (cadar classes))

(display "\n")

(load-class (cdr classes))

)))))

#|

(load-classes "CLASS1.TXT")

(define x (new point 3 4))

(define y (new point 2 5))

(x 'display)

(x 'getx)

(x 'getz)

(x 'polar)

(x 'dist y)

(x 'setx 67)

(x 'display)

(load-classes "inherit.txt")

(define foo (new point 12 45))

(define fum (new 3dpoint 5 7 9))

(foo 'show)

(fum 'show)

(foo 'getx)

(foo 'getz)

(fum 'getx)

(fum 'getz)

(foo 'show 'Hello!)

(fum 'show 'Howdy)

(load-classes "car\_world.txt")

(define car1 (new xcar 'mercedes 250 null form1))

(define dude (new person 40 'eck))

(define punk (new person 17 'bubba))

(define geez (new person 78 'oldtimer))

(car1 'status)

(car1 'drive)

(car1 'fillup 15)

(car1 'status)

(car1 'drive)

(car1 'new\_driver dude)

(car1 'status)

(car1 'drive)

(car1 'status)

(car1 'insurance)

(car1 'new\_driver punk)

(car1 'status)

(car1 'drive)

(car1 'status)

(car1 'fillup 20)

(car1 'insurance)

(car1 'new\_driver geez)

(car1 'status)

(car1 'drive)

(car1 'status)

(car1 'insurance)

|#

(load-classes "inheritance2.txt")

(define test (new dog 'fluffy 200))

(test 'describe)

(define pet (letrec ((\*this\* (eval (lambda args ; this creates the main function for a class

(let ((method (get-method \_\_methods (car args) (count (cdr args)))))

(if (null? method) ; get method

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args)))))))

(\_\_methods `((\_\_new 1 ,(lambda (petname) ; the functions that belong to the class definition

(eval (letrec ((weight (random 50))

(name petname)

(friendly (random 2))

(\*this\* (eval (lambda args (let ((method (get-method \_\_methods (car args) (count (cdr args)))))

(if (null? method)

(let ()

(display "Error: No method with signature:")

(display args)

(newline))

(apply (caddr method) (cdr args)))))))

(\_\_methods `((describe 0 ,(lambda ()

(begin

(display "My name is" )

(display name)

(display "; I'm all of" )

(display weight)

(display " kilos of generic pet protoplasm and I'm permanently" )

(if (eqv? friendly 0) (display "vicious! Grrr!") (display "happy! Let's play"))

(newline))))

(feed 1 ,(lambda (weight)

(begin

(\*this\* 'setweight (+ (\*this\* 'getweight) weight))

(display "gloomf! Food absorbed!")

(newline))))

(setweight 1 ,(lambda (newval) (set! weight newval)))

(getweight 0 ,(lambda () weight)))))

(lambda args (apply \*this\* args)))))) ; pass to \*this\*

(\_\_get-methods 0 #<procedure>) ; returns the methods for this class

(\_\_get-ivars 0 #<procedure>) ; returns the variables for the class

)))

(lambda args (apply \*this\* args)))) ; pass to \*this\*