Chunyi Lyu
Principles of Programming Languages
James Marshall
December 14, 2015

Object-Oriented PASTA

My conference project is extending the language "PASTA" to be an object-oriented language. It adds additional language features, which enable users to define classes, methods as well as create instances. While developing this project, I have used instructions of a suggested assignment from the book "Structure and Interpretation of Computer Programs", which can be found in here: https://mitpress.mit.edu/sicp/psets/ps7oop/readme.html

1. Define Class

A 'define-class' special expression takes in the name of the class, a list of superclass which the class inherits from, and zero or more names for slots (the instance variables). Every class has to have at least one super class. There is a predefined class 'object', and any other class is its subclass. Classes are stored in the top level of the environment. As a result of that, name of each class has to be unique. A class can have multiple superclasses. Instances belong to a class can be applied methods that belong to that class's superclasses. When defining a new class, its superclass's slots will also be carried over.

A 'define-class' statement looks like this:

```
==> (define-class cat (object) age breed)
(defined class: cat)

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

Here we first define a class name "cat", which is a subclass of object. Then we define a subclass of "cat", named "house-cat". Classes will be stored as lists in the environment. Class 'house-cat' will look like this:

```
==> house-cat
(class house-cat ((class cat ((class object () ())) (age breed)) (class object () ())) (address age breed))
```

This is a length-four list. The first element is the symbol ‘class’, which could be used for type-checking. Second thing is the name of the class. The third element is a list of the class’s superclasses. The forth element is a list of all of the slots’ names of the class.

2. Define Method

A ‘define-method’ special expression takes in the name of the method, name of the class that this method can be applied to, and an lambda-exp. Names of methods are stored the same as names of classes, so their have to be unique, and classes and methods could not share the same name either. You could not define a method that could be applied to an non-existing class. The lambda-exp can take in extra arguments, just like our normal lambda expression. Note that lambda expression has to have ‘self’ as the first parameter.
A ‘define-method’ statement looks like this:

```
==> (define-method breed? cat (lambda (self) breed))
(defined method: breed? for class (var-exp cat))
```

Methods are also stored as lists in environment as classes. The stored list of method ‘breed?’ is :

```
==> breed?
(define-method breed? (var-exp cat) (lambda-exp (self) (var-exp breed)))
```

This is a length-four list. The first element is the symbol ‘define-method’. The second is the name of the method. The third is the class that this method belongs to. Then it follows by the lambda expression.

3. Create Instance

Users can create instances by the ‘make’ expression. The expression takes name of a class, and then slots’ names and values pairs. For example, if we would like to create an instance belongs to class ‘cat’, we can do:

```
==> (define neko (make cat (breed "persian") (age 7)))
```

Here, ’make’ expression will create a list. Value of ‘neko’ will be:

```
==> neko
(instance (class cat ((class object () ())) (age breed)) (7 persian))
```

The list has ‘instance’ as the first element, then follows by the class of the instance, as well as a list of slots’ values, which corresponds to the order of slots’ names in the definition of the class.

4. Apply method

When applying a method to an instance, we need to use the 'tell' expression. It takes the instance's name, name of the method, and zero or more extra arguments. If we want to know what breed that the cat 'neko' is:

```
==> (tell neko breed?)
persian
```

When evaluating a 'tell' expression, the interpreter will check whether or not the instance belongs to the class that the method is bonded to. In this case, the method 'breed' is a method defined on class 'cat'. Since the class of 'neko' is 'house-cat' and 'house-cat' is a subclass of 'cat', we can apply 'breed?' to the instance 'neko'.

4. Set Expression

Set expression needs to take three arguments. It takes the name of the instance, name of the slot, and the new value you want to assign to the slot. This expression will always return the updated instance value.

```
==> (set neko age 10)
(update-instance instance (class cat ((class object () ())) (age breed)) (10 persian))
```

Set expression can be useful when we define a class that needs to be updated often. Imagine that we need to define a 'bank-account' class, which contains a variable 'balance'. In order to simulate the action of 'withdraw' and 'deposit', we can define our method this way:

```
==> (define-class bank-account (object) balance)
(defined class: bank-account)

==> (define-method withdraw bank-account
  (lambda (self x)
    (if (x > balance)
      then "Insufficient funds"
      else (set self balance (balance - x)))))
(defined method: withdraw for class (var-exp bank-account))

==> (define-method deposit bank-account
  (lambda (self x)
    (set self balance (balance + x))))
(defined method: deposit for class (var-exp bank-account))
```

This is also the main reason for us to always pass in 'self' as the first parameter. Since the lambda expression could contain a set expression procedure, we need the value of instance to be passed in. Notice that when evaluating tell-exp, the interpreter checks whether the procedure

returns a list, with ‘update-instance’ as the first element. If it does, it indicates that the procedure is a set expression, and the interpreter will update the value of the instance by the cdr of returned value (not including the notation ‘update-instance’).

5. ToDos and Current Problems

Since methods and classes are all stored together in the top level. Methods could not have same names, even if they are for different classes. I think it will be more intuitive if the interpreter could handle methods with the same name but belong to different class. In addition to that, in the current version, subclasses can not override methods from their superclasses. It will be great if the language have both ‘override’ and ‘overload’(methods that have same name, but take in different parameters) functions.