EECS 345: Programming Language Concepts

Interpreter Project, Part 4

Due Monday, April 25

In this homework, you will expand on the interpreter of part 3 by adding classes and objects (instances of classes)

An example program is as follows:

```
class A {
  var x = 6;
  var y = 7;

  function prod() {
    return this.x * this.y;
  }

  function set2(a, b) {
    x = a;
    y = b;
  }
}

class B extends A {
  function set1(a) {
    set2(a, a);
  }

  static function main () {
    var b = new B();
    b.set1(10);
    return b.prod();
  }
}
```

Your interpreter should now take two parameters, a *file* and a *classname*. For example, `(interpret "MyProgram.j" "B")`, where *file* is the name of the file to be interpreted, and *classname* is the name of the class whose main method you are to run. The function should call `parser` on the file *file*, and then lookup `(string->symbol classname)` in the environment to get the desired class, and then lookup the `main` method of this class. The final value returned by your interpreter should be whatever is returned by `main`.

Details

1. Note that we now allow the object type in our language. So, objects can be assigned to variables, passed as parameters, and returned from functions.
2. All mathematical and comparison operators should only be implemented for integers, and logical operators should only be implemented for booleans.
3. You are *not* required to implement the == operator for objects, but you can if you wish.
4. The only operator that is required to work on objects is the `dot` operator.
5. The `new` operator will return an object of the desired class.
6. The `new` operator can only be used in expressions, not as the start of a statement.
7. Variables and methods can now be static (class) or non-static (instance).
8. The `main` method should be static.

9. The language supports use of `this` and `super` object references.
10. The top level of the program is only class definitions.
11. Each class definition consists of assignment statements and method definitions (just like the top level of part 3 of the interpreter).
12. Nested uses of the `dot` operator are allowed.

Please Note: You should be able to create objects (using a generic constructor), set values, call methods, and use values `this` and `super`. You do *not* have to support user defined constructors. You do *not* have to support static fields or methods (other than the main method) and you do *not* have to support abstract methods.

Parser Constructs

```
class A {                    =>   (class A () body)
  body

class B extends A {          =>   (class B (extends A)  body)
  body

static var x = 5;            =>   (static-var x 5)
var y = true;                =>   (var y true)

static function main() {     =>   (static-function main () body)
  body

function f() {               =>   (function f () body)
  body

function g();                =>   (abstract-function g ())


new A()                      =>   (new A)

a.x                          =>   (dot a x)

new A().f(3,5)               =>   (funcall (dot (new A) f) 3 5)
```

Tasks

Here is a suggested order to attack the project.

First, get the basic class structure into your interpreter.

1. Create helper functions to create a new class and instance and to access the portions of a class and instance. The class must store the parent class, the list of instance fields, the list of methods/closures, and (optionally) a list of class fields/values and a list of constructors. Use your state/environment structure for each of these lists. The instance must store the instance's class (i.e. the run-time type or the true type) and a list of instance field values.
2. Change the top level interpreter code that you used in part 3 to return a class instead of returning a state.
3. Create a new global level for the interpreter that reads a list of class definitions, and stores each class with its definition in the state.
4. Create a new `interpret` function that looks up the main method in the appropriate class and calls it. See if you can interpret an example like:

```
class A {
  static function main() {
    return 5;
```

```
  }
}
```

or like this

```
class B {
  static function main() {
    var b = new B();
    return b;
  }
}
```

Next, get the dot operator working.

5. All M_state and M_value functions will need to pass a parameter for the class-type (i.e. the compile-time type or current type) and (optionally) the instance (to avoid having to continuously look up "this" in the state).
6. Create a pair of functions (or a single function that returns a pair) that takes the left hand side of a dot expression and returns the class-type (i.e. the compile-time type or current type or class) and the instance of the left hand side of the dot.
7. Add a fourth value to the function closure: a function that looks up the function's class in the environment/state.
8. Update the code that evaluates a function call to deal with objects and classes. (Follow the denotational semantics sketched in lecture.)
9. Add code to the function lookup to handle the dot operator. See if you can interpret an example like:

```
class A {
  function f() {
    return 5;
  }

  static function main() {
    var a = new A();
    return a.f();
  }
}
```

10. Create helper functions that successfully declare, lookup, and update non-static fields. The functions will need to deal with the fact the the field names part of the state structure is in the class and the field values part of the state structure is in the instance.
11. Add code to the places where you do variable lookups so that it can handle the dot operator.
12. Change your code for a variable to first lookup the variable in the local environment and if that fails to look in the non-static fields.
13. Update the code that interprets an assignment statement so that it looks for the variables with dots in the instance fields and for variables without dots it first looks in the local environment and then in the instance fields.
14. Now test on the first 6 sample programs.

Finally, get polymorphism working.

15. If your state consists of separate lists for the names and their values, change the state so that the values are now stored in reverse order, and you use the "index" of the variable name to look up the value.
16. Make sure the functions that create the new classes and instances correctly append their lists onto the lists from the parent class.

Other language features

Everything we did previously in the interpreter is still allowed: functions inside funtions, call-by-reference (if you chose to implement it). A function that is inside a static function will not have the `static` modifier, but it will be static simply by the nature of the containing function.

For Some Additional Challenge:

Add static (class) methods and fields. For static methods, the only change is that the method will not get the "extra" parameter *this*. For static fields, you will need to change the places that do field lookup and assign so that the method looks in three different environments: the local environment, the class fields, and the instance fields.

Add abstract methods. The interpreter will only support non-static abstract methods. The change you must make is to give an abstract method an appropriate value in the body portion of the closure to indicate that the body does not exist. When an instance is created, you should verify that any abstract methods have been overridden. If any have not, give an appropraite error.

Add user-defined constructors. In the language, the constructor will look like a method that has the same name as the class name, but is not preceded with`function`, and in the parse tree it will be identified by `constructor`.

```
class A {
  A(x) {                =>  (constructor (x) body)
    body
  }
}
```

Constructors can be overloading, and constructors/new needs to have the following behavior:

1. Create the instance including space for all instance fields.
2. Lookup the appropriate constructor (if one exists). If no constructor exists, allow for a default constructor.
3. Call the constructor specified by the `super` or `this` constructor call that should be the first line of the constructor body (or automatically call the parent class constructor with no parameters if no `super()` is present).
4. Evaluate the initial value expressions for the fields of this class, in the order they are in the code.
5. Evaluate the rest of the constructor body.

As a hint, make the constructor list be a separate environment of the class from the method environment. That way constructors will not be inherited.