Interpreter, Part 4

Due Apr 27 by 11:59pm **Points** 100 **Submitting** a file upload **Available** until Apr 28 at 8am

This assignment was locked Apr 28 at 8am.

IMPORTANT: Because April 29 is Reading Day, no late project submissions will be accepted.

For this and all programming project's, you are welcome to work in groups of up to three. The names of all group members should appear at the top of the file, and every member should submit the project on blackboard. All team members are responsible for understanding the code submitted in their name. You do ***not*** *have to keep the same group as the previous interpreter parts.*

A New Parser

This interpreter needs a new parser: classParser.rkt

As with the previous parser, this one is written for racket, and you will need to comment/uncomment some lines to use it with scheme.

The same lex.rkt file will work with the new parser.

The Language

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();
  }
}
```

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 ())

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

new A()                    =>   (new A)

a.x                        =>   (dot a x)

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

Sample Programs

Here are some sample programs in this Java-ish language that you can use to test your interpreter. Please note that these programs cover most of the usual situations, but they are not sufficient to completely test your interpreter. Be certain to write some of your own to fully test your interpreter.

part4tests.html

What Your Code Should Do

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 state 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.

A Recommended List of 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 class closure and an instance closure and to access the members of the class closure and instance closure. The class closure must contain the parent/super class, the list of instance field names, the list of methods/function names and closures, and (optionally) a list of class field names/values and a list of constructors. You may use your state/environment structure for each of these lists. The instance closure must contain 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 closure 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 name with its closure 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. I suggest first handling methods and then fields.

1. All M_state and M_value functions from interpreter, part 3 will need include the compile-time type (current type) as input and (optionally) the instance (to avoid having to continuously look up "this" in the state).
2. Add a fourth value to the function closure: a function (or equivalent information) so that you can look up the function's class in the environment/state.
3. Add "this" as an additional parameter to the parameter list in each (non-static) function's closure.
4. Create a function that takes the left hand side of a dot expression and returns that instance.
5. Adjust the function call so that it looks for the function in the closure of the class (i.e. the run-time type or the true type) of the object that the left hand side of the dot evaluates to.
6. Update the code that evaluates a function call to bind to the parameter ``this'' the value from the left side of the dot.
7. Update the code that calls the function to set the compile-time type (current type) of the function call to be the class from the function's closure.
8. See if you can interpret an example like:

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

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

9. 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 are in the class closure and the field values are in the instance closure.
10. Add code to the places where you do variable lookups so that it can handle the dot operator.
11. Change your code for a variable (without a dot) to first lookup the variable in the local environment and if that fails to look in the non-static fields.
12. 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.
13. Now test on the first 6 sample programs.

Finally, get polymorphism working.

1. 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.
2. Make sure the functions that create the new classes and instances correctly append their lists onto the lists from the parent class.
3. Adjust the function and field lookups to handle the case when the left side of the dot is "super".

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. This will also require you to change how dot is handled because the left side of the dot can now be a class name.

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 appropriate 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 should 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.

Interpreter Part 4

Criteria	Ratings								Pts
Abstraction	**5.0 pts** **Good Abstraction** Uses abstraction throughout.	**4.0 pts** **Good abstraction but the initial state** Uses abstraction throughout but hardcodes '() or '(()()) for the state instead of an abstraction	**2.0 pts** **Missing some abstraction** Accessing elements of the statements uses cars and cdrs instead of well-named functions.	**0.0 pts** **Over use of car/cdr/cons** Accessing the state in the M_ functions uses cars and cdrs instead of good abstraction					5.0 pts
Functional Coding	**10.0 pts** **Excellent functional style**	**8.0 pts** **Good functional style** Mostly uses good functional style, but overuses let or begin.	**7.0 pts** **Mostly functional** Uses the functional style, but also has very non-functional coding such as set!, global variables, or define used other than to name a function. (set-box! is allowed for the state values)	**5.0 pts** **Poor functional style** The code uses an iterative style throughout such as a list of statements executed sequentially.	**0.0 pts** **Violates functional coding** Significant use of set!, define inside of functions, global variables, or anything else that is grossly non-functional.				10.0 pts
Readibility	**5.0 pts** **Full Marks** Nicely readible code: good indentation, well organized functions, well named functions and parameters, clear comments.	**3.0 pts** **Reasonable** Reasonably readible code. Except for a few places there is good commenting, organization, indentation, short lines, and good naming.	**0.0 pts** **No Marks** Hard to read and follow the code due to poor organization or indentation, poorly named functions or parameters, and/or a lack of commenting.						5.0 pts
Input/Output	**5.0 pts** **Full Marks** The interpret function correctly inputs two strings: the file and the class, and the output is an integer or true/false.	**3.0 pts** **Partly correct** Inputs the class as an atom instead of a string and/or outputs #t/#f instead of true.	**0.0 pts** **No Marks** Does not accept two inputs to the interpret function.						5.0 pts
O-O Implementation	**75.0 pts** **Full Marks** 1. "Top level" only handles class definitions. 2. "Middle level" interprets the class and only handles function and variable declarations. 3. Correct class closure is created. 4. Compile-time type is passed to all M_value and M_state functions. 5. Correct instance closure is created and coded so the proper field is accessed. 6. Function closure now includes a means to get the compile time type. 7. Function calls evaluate the arguments in the correct environment and class, and the correct object closure is bound to "this". 8. Code to handle "this" is implemented by either looking up "this" or passing the current instance as a parameter to M_value/M_state 9. Dot's handled in variable names, function names, and M_value. 10. Correctly handles left side of dot to be variable, function, new, this, and super.	**70.0 pts** **Excellent** All 10 points implemented and all work with only small errors.	**65.0 pts** **Good** All 10 points implemented, but there could be some significant errors .	**55.0 pts** **Reasonable** Some of the 10 points are correctly implemented, but some are omitted.	**50.0 pts** **Poor** None of the 10 points are implemented correctly, but some are close.	**38.0 pts** **Minimal** A reasonable attempt at implementing a class structure and passing it to the M_value/M_state.	**20.0 pts** **Better than nothing...** An attempt to implement one of the 10 points, and done in a reasonable enough way that it does not obviously break the rest of the interpreter.	**0.0 pts** **No Marks**	75.0 pts
									Total Points: 100.0