ASSIGNMENT 4

CS 474: Object-Oriented Languages and Environments / Spring 2023

Description

In this assignment, you will add object-oriented concepts to the interpreter presented in class. There are 10 test cases, worth 10 points each: undergraduate students need to get 70 points total for full credit (most likely test cases 1-6 and 10), while graduate students should get all 100 points. For this assignment, you need to write a class that provides an `evaluate` method that can evaluate the following expressions:

- `ClassDefExpression`: Introduces a new class that can be used in the body of the expression. This expression is very similar to the `LetExpression` seen in class. You may want to define a new type of value (e.g. `ClassValue`) to store the class definition.
    - The class definition includes the new class's **name**, its **superclass** if any (note that there is no top `Object` class in this language), an array of **fields**, and an array of **methods**.
    - A **field** is defined by its name and initializer expression, which evaluates to the initial value of the field.
    - A **method** is defined by its name, the names of its arguments, and its body.
    - A `ClassDefExpression` also has a **body**, which evaluates normally but can refer to the class defined in the expression. The value of the body is returned as the value of the whole `ClassDefExpression`.
- `NewExpression`: Creates a new instance of a known class, by using the name of the class. An object is a kind of value, so you should define a new type of value (e.g. `ObjectValue`) to store it.
    - Takes:
        - The **name** of the class to instantiate. (And nothing else; constructors in this language do not take arguments).
    - New instances hold fields with initial **values** obtained by executing the initializer expression for each field, as defined in the class definition.
    - Evaluates to a new value that is an instance of the specified class.
- `InstanceOfExpression`: Checks if an object is an instance of a given class.
    - Takes:
        - An **expression** that evaluates to the object to check.
        - The **name** of the class to check.
    - Evaluates to **true** if the object is an instance of the given class, **false** otherwise.

- `ReadFieldExpression`: Reads the value of a field of the given object.
    - Takes:
        - An **expression** that evaluates to the object whose field we want to read.
        - The **name** of the field to read.
    - Evaluates to the value held in the specified field of the provided object.
        - This value should be either:
            - The initial value, as provided in the class declaration, if the field of the object has not been written yet.
            - The value of the last write to that field of that object.
- `WriteFieldExpression`: Writes the value of a field of the given object.
    - Takes:
        - An **expression** that evaluates to the object whose field we want to write.
        - The **name** of the field to write.
        - An expression that evaluates to the **new value** to write.
    - Evaluates to the new value written to the field.
        - Evaluating this expression should both return the new value and also set the indicated field of the object to that value. You will know it worked if a subsequent `ReadFieldExpression` on that same field returns the new value.
- (grad students only) `CallMethodExpression`: Invokes a method of a given object. Similar to a function call.
    - Takes:
        - An **expression** that evaluates to the object whose method we want to invoke (i.e., the **receiver**).
        - The **name** of the method to invoke.
        - An **array of expressions** that evaluate to the values of the arguments to pass to the method.
    - Evaluates to the value that results from calling the method on the receiver object with the provided arguments. You should also make sure to set the value of the variable `this`, as described below.

This

When invoking a method, the variable `this` is implicit and should be bound to the method's receiver.

- `class Counter { value = 0 ; add(n) { this.value = this.value + n } }`
- `class Factorial { factorial(n) { if (n == 1) then 1 else n * this.factorial(n-1) }`

Inheritance and Subtyping

Your interpreter should take into account subtyping relationships, as defined by the superclasses provided in the `ClassDefExpression`.

- There is only single inheritance in this assignment.
- `InstanceOfExpression` follows the subtyping relationship: If class B is a subclass of A, then instances of B are also instances of A.
    - `class A { }`
    - `class B extends A { }`
    - `new B() instanceof A // should evaluate to true`
- Inherited fields: Subclasses inherit all the fields of their superclasses. There are no static fields.
    - `class A { f = 474 }`
    - `class B extends A { }`
    - `new B().f // should evaluate to 474`
- Methods (grad students only):
    - Subclasses inherit all the methods of their superclasses. If class B is a subclass of A, and A defines method m, then calling m on instances of B should invoke the inherited m defined on A.
        - `class A { m() { … } }`
        - `class B extends A { }`
        - `new B().m(); // this is valid, and calls A.m`
    - It is possible to override methods from superclasses. In this interpreter, a method is overridden if it has the same name (we don't care about arguments).
        - `class A { m() { … } }`
        - `class B { m() { … } }`
        - `new B().m(); // this results in calling B.m`

Entry Point

For this assignment, you need to write a class that extends the class **`Assignment4`**, as shown below. You should also change method `Assignment4.getSolution` so that it creates an instance of the class you added. You should not change any other part of the code that is provided to you.

```
public abstract class Assignment4 {
    abstract Value evaluate(Expression c, Environment e);

    static Assignment4 getSolution() {
        throw new Error("Not implemented");
    }
}
```

The interpreter implemented in class is available through static method `Interpreter.evaluate(Expression,Environment,Assignment4)`, so you don't have to copy/paste any code into your solution that is not directly related to Assignment 4. The third argument of

`evaluate` is meant to take the current instance of your solution, so it can call back into your solution when it encounters object-oriented expressions.

Due Date and Resubmission Policy

This assignment is due on **April 14 2023** (Friday) at **11:59pm CDT**.

The code and date used for your submission is defined by the last commit to your Git repository.

Submission and Grading

This assignment is submitted through GitHub. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests that will check if your project outputs the expected result. Each test is worth 10%.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Academic Integrity

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.