ASSIGNMENT 4

CS 474: Object-Oriented Languages and Environments / Fall 2020

Description

In this assignment, you will add object-oriented concepts to the interpreter presented in class (Lectures 16 and 17). 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.
  - A **class** is defined by its name, an array of **fields**, and an array of **methods**. A class may have a **superclass**.
  - A **field** is defined by its name and initializer expression, that evaluates to the initial value of the field.
  - A **method** is defined by its name, the name of its arguments, and its body.
  - Expressions in the body can refer to the class introduced by the `ClassDefExpression`.
- `NewExpression`: Creates a new instance of a known class, by using the name of the class.
  - New instances hold fields with the initial value obtained by executing each field initializer expression, as defined above.
  - Takes:
    - The name of the class to create a new object of.
  - 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 on the given object.
    - Takes:
        - An expression that evaluates to the object from which to read the field.
        - The name of the field to read.
    - Evaluates to the value held by the field on the provided object.
        - This value is either
            - The initial value, as provided in the class declaration, if the field has not been written yet on the provided object.
            - The value of the last write made to that field on that object.
- `WriteFieldExpression`: Writes the value of a field on the given object
    - Takes:
        - An expression that evaluates to the object from which to read the field.
        - The name of the field to read.
        - An expression that evaluates to the new value to write.
    - Evaluates to the new value written to the field.
        - Besides returning the new value written, this expression also updates the value of the field in the provided object. Later `ReadFieldExpression` return this value.
- `CallMethodExpression`: Invokes a method on a given object.
    - Takes:
        - An expression that evaluates to the object on which to invoke the method (i.e., the receiver).
        - The name of the method to invoke.
        - An array of expressions that evaluate to the value of the arguments to pass to the method.
    - Evaluates to the value resulting of calling the method on the receiver object.

Subtyping

Your interpreter should follow the subtyping relationship, as defined by the super-classes 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 super-classes. There are no static fields.
  - `class A { f = 474 }`
  - `class B extends A { }`
  - `new B().f // should evaluate to 474`
- Methods:
  - Subclasses inherit all the methods of their super-classes. If class B is a subclass of A, and A defined 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 super-classes. 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`

This

When invoking a method, 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) }`

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method `Assignment4.getSolution` so that it creates an instance the class you added. You cannot change any other part of the code that is provided to you.

```
public abstract class Assignment 4 {
    abstract Value evaluate(Expression c, Environment e);

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

The interpreter implemented in Lecture 17 is available through static method `Interpreter.evaluate(Expression,Environment)`, so you don't have to copy/paste any code into your solution that is not directly related to Assignment 4. The interpreter was also modified to invoke your `evaluate` method when evaluating sub-expressions.

Due Date and Resubmission Policy

This assignment is due on **November 7 2020** (Saturday) at **5pm CST**. There is no late policy.

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

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%**. You can resubmit your assignment until **November 14 2020** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment4` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking Travis. The automatic grade is determined by 10 tests, that will check if your project outputs the expected result. Each test is worth 10%.

Graduate students should also submit a video explaining their solution. For graduate students, each test is worth 9%, for a total of 90%, and the video is worth 10%. **The maximum length for the video is 5 minutes.**

This video should be a screencast of their IDE open on the code submitted, and the student should highlight the code and narrate the purpose of the highlighted code. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

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

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

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.