Software Construction (L+E) HS 2023

Instructor: Prof. Dr. Alberto Bacchelli — Assignment 2

Teaching Assistant: Konstantinos Kitsios — Week 06

To correctly complete this assignment you **must**:

- Carry out the assignment with the team you have formed for assignment 01. Work with your team only (unless otherwise stated). You are allowed to discuss solutions with other teams, but each team should come up its own personal solution. A strict plagiarism policy is going to be applied to all the artifacts submitted for evaluation
- Prepare the solutions to the exercises by strictly following this structure:
  - A root folder named: `Group[id on OLAT]-a[AssignmentNumber]`[1]
  - Inside the folder:

    `README.md` : a Markdown file explaining the decisions taken in the source code and documents your solution;

    `lgl_interpreter.py` : a python file implementing the interpreter of the LGL [2] language, as described in the 3 exercises below;

    `example_operations.gsc` : a LGL file that will contain a simple program of your choice that demonstrates the functionalities of Exercise 1;

    `example_class.gsc` : a LGL file that will contain the implementation of classes as requested in exercise 2

    `example_trace.gsc` : a LGL file that we provide you to demonstrate how the output of the tracer of Exercise 3.1 should look like

    `reporting.py` : a python file that will gather tracer results and display them in a tabular format as requested in Exercise 3.2
- The final structure of the directory should match that in Figure 1
- Package your root folder into a single ZIP file named:
  `Group[id on OLAT]-a[AssignmentNumber].zip`[3]
- Upload the solution to the right OLAT task by the deadline (*i.e.*, **Nov 14, 2023 @ 18:00**)

Figure 1: Directory Structure

```
Groupg99-a2/
├── README.md
├── lgl_interpreter.py
├── example_operations.gsc
├── example_class.gsc
├── example_trace.gsc
└── reporting.py
```

---

[1] *e.g.*, a correct name would be: `Group99-a2`

[2] Stands for Little German Language; materials from Session04 in OLAT contain examples of such files

[3] *e.g.*, a correct name would be: `Group99-a2.zip`

Goal

The goal of this assignment is to expand the Little German Language (LGL) introduced in class and based on Chapter 7 and Chapter 8 of the book. You will add support for (1) more basic operations, (2) classes (following the ideas seen Chapter 2) and (3) tracing. After successfully completing the assignment you will have a deeper understanding on how interpreters of real-world languages are designed, thus enhancing your software design skill in this domain and beyond.

Grading Criteria

Your assignment will be graded based on the following criteria:

- Correctness of implementation.
- Correctness of design process: document your design decisions in the `README.md` file.

Assignments will be scored based on two main factors: fulfillment of criteria and evidence of effort:

- **2 out of 2:** For assignments that meet all criteria and clearly demonstrate substantial effort.
- **1 out of 2:** For assignments that meet some criteria but still show evident effort.
- **0 out of 2:** For assignments that lack clear evidence of effort, regardless of criteria met.

1 More Capabilities

1.1 Implementation

Add the following new functionalities to LGL:

1. **Multiplication**, **Division**, and **Power** operations
2. **Print** statements
3. **While** loops
4. **Arrays**:
    - Creating a new array of fixed size
    - Getting the value at position `i` of an array
    - Setting the value at position `i` of an array to a new value
5. **Dictionaries**:
    - Creating a new dictionary
    - Getting the value of a key
    - Setting the value of a key to a new value
    - Merging two dictionaries (i.e, implement the `|` operator of Python)

To do so, implement the `lgl_interpreter.py` file. You can get ideas from the materials from Session 04 (available on OLAT) and Chapter 7 and Chapter 8 of the book.

1.2 Use

Write a file, named `example_operations.gsc`, to showcase the use of all the aforementioned functionalities you implemented with a simple program of your choice. It should run from terminal with the following command:

```
python lgl_language.py example_operations.gsc
```

2 An Object System

2.1 Implementation

In Chapter 2 and Chapter 8, we have seen how an object system can be implemented (e.g., using dictionaries and functions). Using these ideas and what you learned about adding the support for functions LGL from Chapter 7, add support for Objects and Classes in LGL. In particular, the LGL interpreter should support:

- Class definition and Object instantiation
- Single Inheritance
- Polymorphism

Modify the `lgl_interpreter.py` file you created for exercise 1 accordingly.

2.2 Use

As an example to test the capabilities of your object system, your LGL interpreter should be able to correctly handle the classes defined in the UML diagram of Figure 2. You have already seen these classes in Chapter 2, but this time you will implement them using LGL syntax instead of Python syntax.

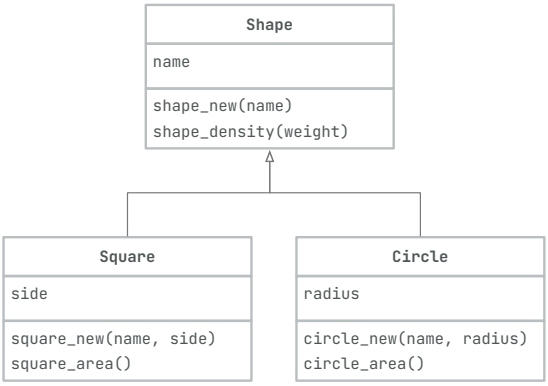

Figure 2: UML class diagram, matching the example in Chapter 2 (Section 2.4)

More specifically, you will write an `example_class.gsc` file that will implement the steps described in natural language below:

1. Define a `Shape` class, which also defines:
   - a constructor `shape_new(name)`
   - a method `shape_density(thing, weight)`
2. Define a `Square` class that inherits from the `Shape` class, and also defines:
   - a constructor `square_new(name, side)`
   - a method `square_area(thing)`
3. Define a `Circle` class that inherits from the `Shape`, and also defines:
   - a constructor `circle_new(name, radius)`
   - a method `circle_area(thing)`
4. Create a new `Square` object with `side=3` and `name='sq'` and assign it to a new variable
5. Create a new `Circle` object with `radius=2` and `name='ci'` and assign it to a new variable

6. Return the sum of the densities of the two objects (use `weight=5` for both objects)

Given you implement these classes in LGL in a file named `example_class.gsc`, the following program must produce the shown output:

$$\texttt{python lgl\_interpreter.py example\_class.gsc} \xrightarrow{\text{outputs}} 0.96\,^{4}$$

[4] or 0.955556 or similar

3 Tracing

Tracing is an important concept in software engineering that enables the building of debuggers, profilers and more. We are going to add basic tracing functionalities to the `LGL interpreter`.

3.1 Logging

You are tasked with enhancing the `LGL interpreter` by adding an optional argument: `--trace trace_file.log`. When this argument is used, it should log details about the start and end times of each function into the specified `trace_file.log`. Logging should be implemented using Decorators introduced in Chapter 9. After implementing this feature in `lgl_interpreter.py`, executing the command:

```
python lgl_interpreter.py example_trace.gsc --trace trace_file.log
```

should generate a `trace_file.log` like the one presented below:

```
id,function_name,event,timestamp
358887,add_cubes,start,2023-10-24 19:20:08.045086
270697,get_cube_power,start,2023-10-24 19:20:08.045279
270697,get_cube_power,stop,2023-10-24 19:20:08.045657
137801,get_cube_power,start,2023-10-24 19:20:08.045788
137801,get_cube_power,stop,2023-10-24 19:20:08.045850
358887,add_cubes,stop,2023-10-24 19:20:08.045898
```

The output file is in csv format and has four columns:

- The first column has a unique ID for distinguishing separate calls of the same function. For instance, `get_cube_power` appears to have been called twice, with each instance having a distinct ID.
- The second column shows the name of the function that was called.
- The third column indicates the event type, specifying if the function has started ('start') or ended ('stop').
- The fourth column logs the timestamp of the event.

The use of unique IDs is crucial, particularly for recursive function calls. Consider the scenario where `get_cube_power` calls itself. We would see two consecutive 'start' entries, and the unique IDs help determine which function call ends upon seeing a 'stop'.

Use: To demonstrate the new tracing capability use the LGL file you implemented in Exercise 2:

```
python lgl_interpreter.py example_class.gsc --trace trace_file.log
```

3.2 Reporting

By utilizing the output file of 3.1, you are requested to build a simple reporting tool for the LGL. To do so, implement a `reporting.py` file that, when ran from command line like this: `python reporting.py trace_file.log` displays the tracing results in an aggregated format as below:

```
| Function Name  | Num. of calls | Total Time (ms) | Average Time (ms)|
|-------------------------------------------------------------------|
| add_cubes      |       1       |      0.812      |      0.812       |
| get_cube_power |       2       |      0.440      |      0.220       |
```

The table has 4 columns:

- The first column contains the name of the function
- The second column contains the number of calls of the function
- The third column contains the total execution time of the function, summed across all the executions
- The fourth column contains the average execution time, which is obtained by dividing the Total Time with the Num. of calls

Use: Demonstrate the new reporting capability using the `trace_file.log` created in 3.1.