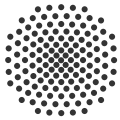

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2022

Exercise 2: Names, Scopes, and Bindings

(Deadline for uploading solutions: May 11, 2022, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the folder structure and the templates that must be used for the submission.**

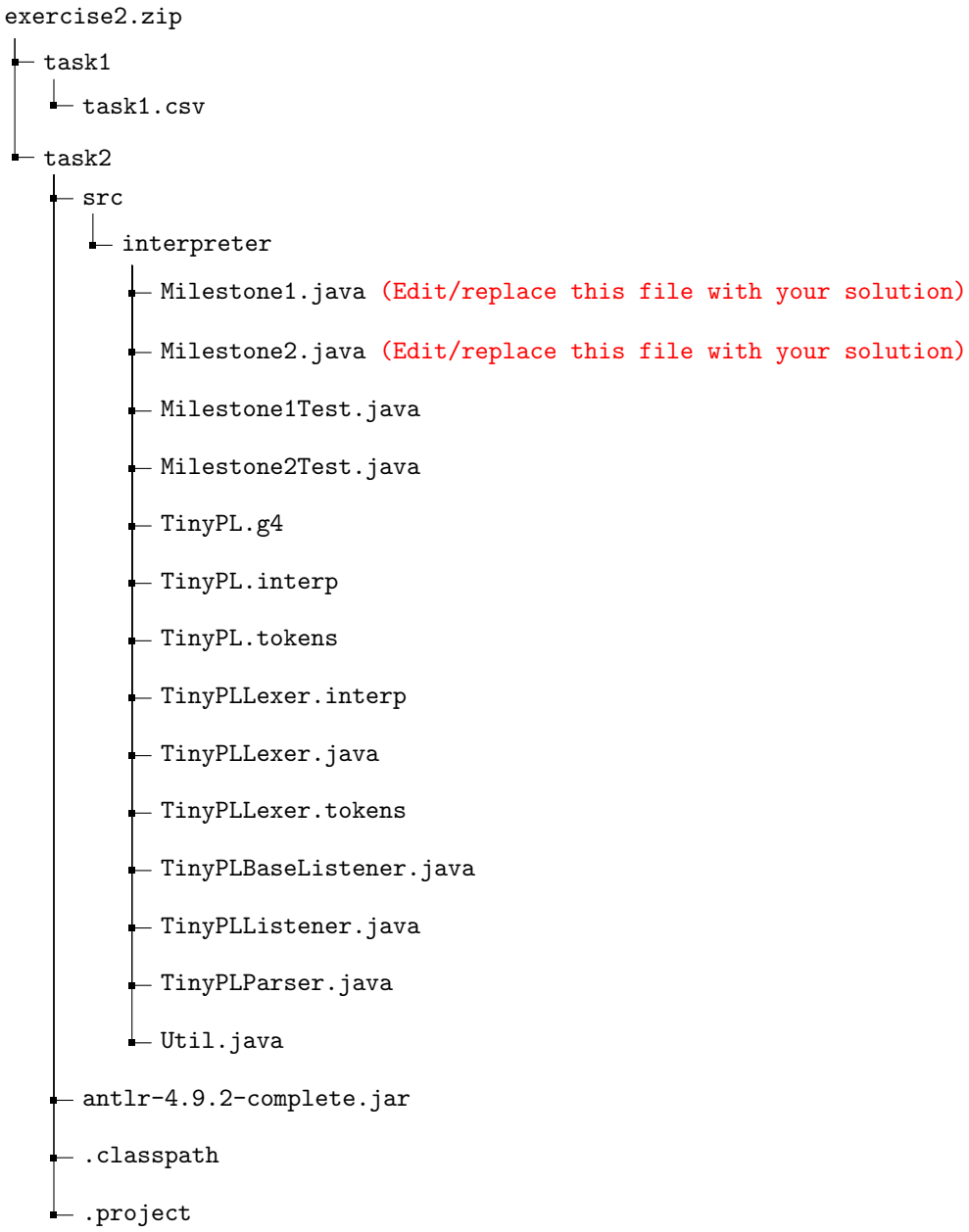

Figure 1: Folder structure to use for submission.

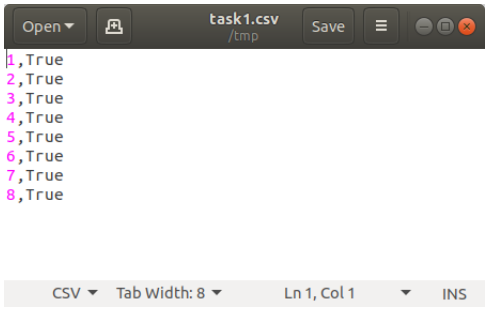

Figure 2: Example of the file containing the answer of Task I.

The submission must be compressed in a zip file (not rar, 7z, gz, or anything else) using exactly the folder structure and names in Figure 1. To ensure that your submission has the expected structure, check it with the validator.jar tool provided on the course page, using the following command (where exercise2.zip is the original file provided to you, and your_exercise2.zip is your modified version of it):

java -jar validator.jar exercise2.zip your_exercise2.zip

Make sure that your IDE is configured to use Java 8, as described by the .project files in the template, or alternatively, make sure that your code can be compiled and executed with Java 8.

Note: Submissions that do not comply with the exact folder structure and names in Figure 1, or that do not work with Java 8, cannot be graded.

1 Task I (10% of total points of the exercise)

Which of the following statements are correct? Your answer, either **True** or **False**, must be written into the file *exercise2/task1/task1.csv* one answer per line as illustrated in Figure 2.

1. In languages with dynamic binding you cannot get an undefined variable error.
2. Variables can be bound statically or dynamically, but functions are always bound statically.
3. Aliases can exist for objects in static memory, objects in the stack, or objects in the heap.
4. When determining the binding of a method to call in Java, the arguments passed to the call may be relevant.
5. Interpreted languages cannot have static scoping.
6. In Java, an object on the stack can have a reference to an object on the heap.
7. Two scopes either have no overlap, or one is nested in the other; but object lifetimes can overlap arbitrarily.
8. In C, if you have a “dangling reference”, all aliases of that reference are also “dangling”.

2 Task II (90% of total points of the exercise)

The goal of this task is to implement an interpreter for a toy programming language called *TinyPL*. Figure 3 shows the grammar of the language. TinyPL has declared variables, assignments to variables, declared functions, and function calls. User-defined functions can be nested and do not take any parameters. In addition to user-defined functions, there is a built-in function `print`, which expects a single variable as a parameter and outputs the value of that variable. The language includes neither control flow statements (e.g., `if` or `while`) nor any other language features found in a real programming language.

```
P ⟶ Stmt*
Stmt ⟶ VarDecl | Assign | FctDecl | Call
FctDecl ⟶ function Name Stmt*
VarDecl ⟶ var Name;
Assign ⟶ Name = Val;
Val ⟶ Integer | Name
Call ⟶ Name()
```

Figure 3: Grammar of the toy programming language TinyPL.

Assume that the following is true about every program written in TinyPL:

- Each variable has been declared before its being used.
- Each function has been declared before its being called.
- The set of function names and variable names in a program are disjoint.

For example, the following shows a TinyPL program that declares two functions and two variables, and then calls function `a`, which in turns calls function `b`. The behavior of this program depends on the rules of the language regarding bindings and scopes.

Example of TinyPL program with nested calls.

```
1  function a {
2    var x;
3    x = 55;
4    b();
5  }
6  function b {
7    print(x);
8  }
9  var x;
10 x = 1;
11 a();
```

Your task is to implement two interpreters for different variants of TinyPL, as outlined in the two milestones below. Each interpreter takes the parse tree of a program P as the input and gives the output produced by P as an output. Since the only way to produce output in TinyPL is through calls to `print`, the output is represented as the list of printed strings.
Note on grading: Each of the milestones contributes equally to the total points for this task.

2.1 Milestone 1: TinyPL with Dynamic Scoping

The first interpreter covers the full TinyPL language and assumes that the language uses dynamic scoping. For example, the example program with nested functions given above will produce the output "55". Please use the provided class `Milestone1` to implement your solution.

2.2 Milestone 2: TinyPL with Static Scoping

The second interpreter also covers the full TinyPL language, but it assumes that the language uses static scoping. For the example program with nested functions given above, the interpreter will produce the output "1". Please use the provided class `Milestone2` to implement your solution.

2.3 Existing Code and Helper Classes

To ease your implementation, we provide several helper classes:

- `Util`, which provides methods for parsing a program given as a string into a parse tree.
- `ParseTreeViewer`, which parses a given program into a parse tree and display the tree. This class is useful to understand how the parse tree of a program looks like.

We provide a parser for TinyPL that has been automatically generated with ANTLR4. Please have a look at the following classes, which will be useful for your implementation:

- `org.antlr.v4.runtime.tree.ParseTreeWalker`, which is provide in the `antlr-4.9.2-complete.jar` file. The class allows you to traverse a parse tree and to perform some action whenever a specific kind of node is visited. We recommend to implement your interpreters by using this class to visit each of the statements in the given program.
- `TinyPLBaseListener`, which is generated by ANTLR4 from the TinyPL grammar. You should extend this class and pass an instance of your subclass into the parse tree walker above.

In addition to these classes, you are allowed to use any other public classes provided by ANTLR4 and all public classes of the Java standard library, but no other, external libraries.
Side note (not required to read for solving the exercise): The syntax of the language is implemented using ANTLR4. See the `TinyPL.g4` file for the definition of tokens and for the grammar rules. If you are curious, you can re-generate the parser for the language using this command: `java -jar antlr-4.9.2-complete.jar src/TinyPL.g4`

2.4 Testing

We provide test cases for each milestone: `Milestone1Test`, and `Milestone2Test`. As usual, you are strongly advised to implement additional test cases to run your interpreters with other kinds of programs.