University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2021

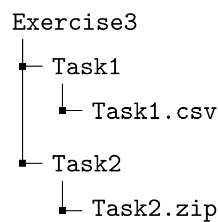
Exercise 3: Names, Scopes, and Bindings

(Deadline for uploading solutions: June 4, 2021, 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**.

The folder structure is:



The submission must be compressed in a zip file using the given folder structure. The name of folders and files must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (**not rar**, 7z, etc.).
- Your zip file should contain exactly one top-level directory *"Exercise3"*, as in the zip file provided by us.
- Do not rename or move files or folders, simply open the files provided and put your solutions.

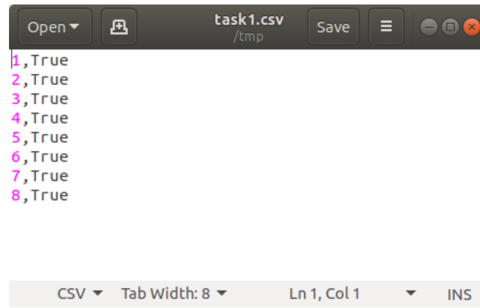


Figure 1: Example of the file containing the answer of Task I.

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 *Exercise3/Task1/Task1.csv* one answer per line as illustrated in Figure 1.

1. In a programming language without any automatic memory management, such as C, it is impossible to dynamically allocate data structures in the heap.
2. In a language with dynamic scoping, a name is resolved to value by using the most recent, still active binding established during the execution.
3. Deep binding means that a referencing environment is created when executing a call that passes another function as an argument.
4. Some programming languages to now allow function overloading.
5. During a program execution, the program may run out of memory on the stack, but never on the heap.
6. A compilation error can occur due to the usage of a variable declared in a different function scope.
7. The lifetime of a global variable can be longer than the execution time of the program.
8. In a statically scoped language, the binding of a name can only be derived during program execution.

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 2 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 2: 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 three interpreters for different variants of TinyPL, as outlined in the three 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 three milestones contributes equally to the total points for this task.

2.1 Milestone 1: TinyPL without User-Defined Functions

The first interpreter is for a subset of TinyPL that includes neither function declarations nor calls to user-defined functions. In other words, a program in this language is a sequence of variable declarations, variable assignments, and print statements. For example, the following program will print "3" and then "4":

Example of TinyPL program without user-defined functions.

```
1 var x;
2 x = 3;
3 var y;
4 print(x);
5 y = 4;
6 print(y);
```

Please use the provided class `InterpreterMilestone1` to implement your solution. Method `run` of this class takes a program, should interpret it, and then return the output produced by the program.

2.2 Milestone 2: TinyPL with Dynamic Scoping

The second 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 `InterpreterMilestone2` to implement your solution.

2.3 Milestone 3: TinyPL with Static Scoping

The third 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 `InterpreterMilestone3` to implement your solution.

2.4 Existing Code and Helper Classes

The syntax of the language is implemented using ANTLR4. See the `TinyPL.g4` file for the definition of tokens and for the grammar rules. To obtain a parser for the language, all you need to do is run the following command from the "Task2" directory:
`java -jar antlr-4.9.2-complete.jar src/TinyPL.g4`
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.

In addition, please have a look at the following classes provided as part of ANTLR4, which will be useful for your implementation:

- `org.antlr.v4.runtime.tree.ParseTreeWalker`, which 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 user any other public classes provided by ANTLR4 and all public classes of the Java standard library.

2.5 Testing

We provide test cases for each milestone: `TestInterpreterMilestone1`, `TestInterpreterMilestone2`, and `TestInterpreterMilestone3`. As usual, you are strongly advised to implement additional test cases to run your interpreter with other kinds of programs.

2.6 What to Change and What to Submit

Your implementation should be in the respective `InterpreterMilestone*` classes. You are allowed to add additional classes, but please do not change the directory structure of the project. The code must compile and run with Java 8. Once done, please export the project as in the previous exercises and update the main .zip file with the .zip of Task II.