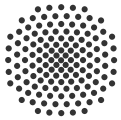

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2023

Exercise 2: Names, Scopes and Bindings

(Deadline for uploading solutions: May 18, 2023, 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 shown in Figure 1.

```
exercise2.zip
├─ task_1
│  └─ task1.json (Edit/replace this file with your solution)
└─ task_2
   ├─ Milestone1.js (Edit/replace this file with your solution)
   ├─ Milestone2.js (Edit/replace this file with your solution)
   ├─ Milestone1.test.js
   ├─ Milestone2.test.js
   ├─ package.json
   ├─ TinyPL.g4
   ├─ TinyPL.interp
   ├─ TinyPL.tokens
   ├─ TinyPLLexer.interp
   ├─ TinyPLLexer.js
   ├─ TinyPLLexer.tokens
   ├─ TinyPLListener.js
   ├─ TinyPLParser.js
   └─ util.js
```

Figure 1: Folder structure to use for submission.

The submission must be compressed in a zip file using the given folder structure. The names 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, gz, or anything else).
- Do not rename files or folders, simply open the files provided and put your solutions.
- The assignment should be implemented using Node.js $\geq$ v14.

Note: Submissions that do not comply with the exact folder structure and names in Figure 1, or that do not work with Node.js $\geq$ v14, cannot be graded.

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

The goal of this task is to understand the difference between static and dynamic scoping. Code 1 contains a program to be analyzed using these two approaches. Your answer has to be written to the `task1.json` file, where you replace the `-1` values with the output of Code 1 depending on the used scope.

Code 1.

```
function1(){
    x = x + 1
}

function2(){
    function1()
    x = x + 1
}

x = 0
function2()
print(x)
```

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.

```
function a {
  var x;
  x = 55;
  b();
}
function b {
  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 Environment Setup

This assigment must be implemented in JavaScript. Therefore, make sure you have `Node.js` $\geq$ `v14` and `npm` installed. To install `Node.js` and `npm` there are many tutorials available online, such as this one. To install antlr4, which is also required, run `npm install` in the `task_2` folder.

2.4 Existing Code and Helper Classes

To ease your implementation, we provide a `util.js` file, which provides methods for parsing a program given as a string into a parse tree.

We provide a parser for TinyPL that has been automatically generated with ANTLR4. Please have a look at the `TinyPLListener` class, which will be useful for your implementation. You should extend this class. Then, an instance of your class is passed to the `antlr.ParseTreeWalker` class, which performs some action whenever a specific kind of node is visited.

In addition to these classes, you are allowed to use any other public classes provided by ANTLR4 and all public classes of the Node.js built-in modules, 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 yourself: 1) download the `antlr-4.12.0-complete.jar` from https://www.antlr.org/download.html; 2) run the following command: `java -jar antlr-4.12.0-complete.jar -Dlanguage=JavaScript src/interpreter/TinyPL.g4`.

2.5 Testing

We provide test cases for each milestone: `Milestone1.test.js`, and `Milestone2.test.js`. To run the test cases, execute the following commands from the `task_2` folder: `node Milestone1.test.js` and `node Milestone2.test.js`. As usual, you are strongly advised to implement additional test cases to run your interpreters with other kinds of programs.