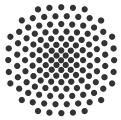

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Winter 2024/2025

Exercise 2: Names, Scopes and Bindings

Strict deadline for uploading solutions: **Dec 4, 2024, 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:

```
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
```

The submission must be compressed in a .zip file (not .rar, .7z, .g.z, or any other format) using exactly the folder structure and names shown above. The assignment must be implemented in in JavaScript (Node.js version >= 14). Submissions that do not comply with these requirements will not 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 value of `a` and `b` each time they are read and written in Code 1 depending on the used scoping rules. The `var` keyword is used to declare a variable within a particular scope. That is, a variable declaration in a function creates a new local variable within the function, and a variable declaration in the main script creates a new global variable.

Code 1

```
1  var a = 0
2  var b = 100
3
4  function1(){
5    a = a + 1
6  }
7
8  function2(){
9    a = a + 1
10   function1()
11 }
12
13 function3(){
14   var a = 98
15   function2()
16 }
17
18 function3()
19
20 if (a == b) {
21   print(True)
22 } else {
23   print(False)
24 }
```

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

The goal of this task is to implement an interpreter to directly execute instructions for a toy programming language called *TinyPL*. Figure 1 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. Variables in user-defined functions are declared in the beginning of the function. Each program must have a `Main` function, which is implicitly called at the beginning of the program. Before the first function declaration, a program may have a list of variable declarations, which declare global variables. 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.

```
Program ⟶ Globals Fcts
Globals ⟶ VarDecl*
Fcts ⟶ FctDecl*
VarDecl ⟶ var Identifier;
FctDecl ⟶ function Identifier { VarDecl* Stmt* }
Stmt ⟶ Assignment | FctDecl | Call
Assignment ⟶ Identifier = Val;
Val ⟶ Number | Identifier
Call ⟶ Identifier()
```

Figure 1: 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 sets of function names and variable names in a program may overlap. The language distinguishes between functions and variables based on the context.

For example, the following shows a TinyPL program that declares two global variables and three functions. The `Main` function gets implicitly called, 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.

Code 2: Example of a TinyPL program

```
1  var x;
2  var y;
3  function a {
4    var x;
5    x = y;
6    b();
7  }
8  function b {
9    print(x);
10 }
11 function Main {
12   x = 1;
13   y = 99;
14   a();
15 }
```

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 above program will produce the output "99". 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.

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.

Existing Code and Helper Classes

We provide a parser for TinyPL that has been automatically generated with ANTLR4. The `util.js` file provides methods for parsing a program given as a string into a parse tree. When we successfully parse a program, we get a parse tree that represents its content. A tree is a common data structure that starts with a root node that can have multiple children. The root node has no parent node, while all other nodes have one and only one parent node.

Please have a look at the `TinyPLListener` class, which will be useful for your implementation. `TinyPLListener` contains enter and exit methods for each rule in the grammar in `TinyPL.g4`. These functions will be invoked when a piece of code matching the rule is encountered.
To ease your implementation, we provide scaffolding code for the implementation of each milestone in files `Milestone1.js` and `Milestone2.js`. Notice that both of these files contain a `MyListener` class, which extends `TinyPLListener`. The complete solution for each milestone should overwrite the necessary methods from `TinyPLListener`, e.g. `enterFunctionDeclaration`.

In each milestone, `MyListener` is instantiated in a `run` function, e.g., at line 45 in `Milestone1.js`. Then, an instance of your class and a parse tree of a program is passed to the `antlr.ParseTreeWalker` class, which traverses the received parse tree using depth-first search. A depth-first search means that when a node is accessed, its children will be accessed, and if one of the children nodes has its own children, they will be accessed before continuing on with the other children of the first node.
Tip: When visiting a function declaration, i.e., at `enterFunctionDeclaration`, by default the function statements would be visited. However, these statements should be interpreted when the function is called. Therefore, you have to store functions as sub-procedures to be run at a later state. This is why we also provide the `visitFctBody` function.

In addition to the provided 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, but helpful for your general understanding): 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 TinyPL.g4`.

Existing Material on Parsers and Listeners in ANTLR and JavaScript

For a deeper understanding on parsers and listeners in ANTLR and JavaScript, we refer to the following material containing detailed definitions of various terms and examples:

- https://tomassetti.me/antlr-mega-tutorial/#chapter29.
- https://tomassetti.me/listeners-and-visitors/.
- https://medium.com/dailyjs/compiler-in-javascript-using-antlr-9ec53fd2780f.

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`. You are strongly advised to implement additional test cases to run your interpreters with other kinds of programs. We will use additional tests when grading your solution.