Project Part 5: Generator

New Attempt

Due Apr 18 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** java

In this assignment, you will implement the analyzer for our language. This is the fourth step in the compiler process which takes the AST processed by the analyzer and performs Java code generation.

Submission

You will submit `Generator.java`, which implements your generator. The `Generator` class includes structure for implementing the Visitor pattern and helping you write your generated Java code (check out the methods `newline` and `print`). You will not modify (and therefore not submit) `Ast.java`, `Environment.java`, and `Scope.java`. In addition, while it is recommended that you add your own test cases to `GeneratorTests.java`, you will not submit `GeneratorTests.java`.

- There will be one test submission on **Wednesday, April 14**.
- The final submission is **Sunday, April 18**.

Generator Overview

Recall that the job of the generator is to convert the Ast into Java code. Recall, from our lecture discussion, there are different types of code that could be generated. For this project, we have chosen to generate Java source code (similar to the process C++ uses, generating C code). Then, we can compile and execute the Java source code using the standard JVM. Note, the class Generator implements the interface Visitor (contained in `Ast.java`, see below).

Formatting

The Java source code you write must be formatted in a consistent manner that follows the guidelines described below. The `newline` and `print` methods given will assist you in doing this,. The `newline` can be used by passing in the indent property or by passing a hardcoded number for known cases, such as using `0` for empty lines.

You are required to follow all formatting, including spaces and indentations, in exactly the same manner as it is given here. Detailed examples follow for all of the nodes in our grammar. As well, examples are provided in `GeneratorTests.java`.

Ast Types for Code Generation

Each `Ast` class has its own `visit` method, which behave as described below.

- Additionally, there are a few parts of the AST where generating a list of statements (or even higher up, a list of anything) which are used multiple times. It may be helpful to create another function to handle this logic instead of duplicating it to make your work easier.

AST Class	Specification	Examples
`Ast.Source`	Generates a source. This includes a definition for the `Main` class that contains our code as well as the `public static void main(String[] args)` method used as the entry point for Java. The order of generation is the class header, the source's fields, Java's main method, the source's methods, and finally the closing brace for the class. Pay close attention to spacing and indentation; fields are grouped together while methods are separated by an empty line (hint: use `newline(0)` for empty lines, giving it an explicit indent of `0`). The Java `main` method will be the following, which creates an instance of our `Main` class and calls our language's `main` method (which has a different signature since it does not take arguments). `System.exit` is used to specify the exit code of a Java program, unlike C/++ which does so automatically. • This is not critical for understanding the assignment, but is another 'flare' to draw attention to important concepts that you'll almost certainly make use of later on. `public static void main(String[] args) { System.exit(new Main().main()); }` Returns `null`.	• `DEF main(): Integer DO print("Hello, World!"); RETURN 0; END` ◦ `public class Main { public static void main(String[] args) { System.exit(new Main().main()); } int main() { System.out.println("Hello, World!"); return 0; } }` • `LET x: Integer; LET y: Integer = 10; DEF main(): Integer DO RETURN x + y; END` ◦ `public class Main { int x; int y = 10; public static void main(String[] args) { System.exit(new Main().main()); } int main() { return x + y; } }`
`Ast.Field`	Generates a field expression. The expression should consist of the type name and the variable name stored in the Ast separated by a single space character. If a value is present, then an equal sign character with surrounding single spaces is generated followed by the variable value. A semicolon should be generated at the end. Returns `null`.	• `LET x: String;` ◦ `String x;` • `LET y: Boolean = TRUE AND FALSE;` ◦ `boolean y = true && false;`
`Ast.Method`	Generates a method expression. The method should begin with the method's jvm type name followed by the method name both found in the Ast. Then the method should generate a comma-separated list of the method parameters surrounded by parenthesis. Each parameter will consist of a Jvm type name and the parameter name. Following a single space, the opening brace should be generated on the same line. If the statements is empty the closing brace should also be on the same line, otherwise each statement is generated on a new line with increased indentation followed by a closing brace on a new line with the original indentation. Returns `null`.	• `DEF area(radius: Decimal): Decimal DO RETURN 3.14 * radius * radius END` ◦ `double area(double radius) { return 3.14 * radius * radius; }`
`Ast.Stmt.Expression`	Generates an expression. It should consist of the generated expression found in the Ast followed by a semicolon. Though the Analyzer requires the contained expression to be a function expression, your generator should still work with other expression types. Returns `null`.	• `print("Hello World");` ◦ `print("Hello World");` • `1;` ◦ `1;`
`Ast.Stmt.Declaration`	Generates a declaration expression. The expression should consist of the type name and the variable name stored in the Ast separated by a single space. If a value is present, then an equal sign with surrounding single spaces is generated followed by the generated variable value. A semicolon should be generated at the end. Returns `null`.	• `LET name: Integer;` ◦ `int name;` • `LET name = 1.0;` ◦ `double num = 1.0;`
`Ast.Stmt.Assignment`	Generates a variable assignment expression. The name should be the receiver of the variable stored in the Ast and the value should be the generated value of the variable. An equal sign character with surrounding single spaces should be generated between the name and value. A semicolon should be generated at the end. Returns `null`.	• `variable = "Hello World";` ◦ `variable = "Hello World";`
`Ast.Stmt.If`	Generates an If expression. The expression should consist of the `if` keyword followed by the generated condition with the surrounding parenthesis. The opening brace should be generated on the same line. After a single space, the opening brace should be generated followed by a newline with an increase in the indentation and the generation of all the statements each ending with a newline. Following this should be a decrease in the indentation and the corresponding closing brace. If there is an else block, then generate the `else` keyword on the same line with the same block formatting. There is **no concept** of **else-if** in our grammar, so nested if statements will still appear nested. If there's not an else block, then the entire `else` section is left out of the generated code. Returns `null`.	• `IF expr DO stmt; END` ◦ `if (expr) { stmt; }` • `IF expr DO stmt1; ELSE stmt2; END` ◦ `if (expr) { stmt1; } else { stmt2; }`
`Ast.Stmt.For`	Generates a for loop expression. The expression should consist of the `for` keyword. It is followed by a single space with the following in parenthesis • The variable type, `int` • The variable name found in the Ast • A colon with surrounding spaces • The generated value expression The opening brace should be generated on the same line after a single space and followed by a newline with an increase in the indentation and the generation of all the statements ending with a newline. Following this should be a decrease in the indentation and a closing brace. Returns `null`.	• `FOR num IN list DO print(num); END` ◦ `for (int num : list) { System.out.println(num); }`
`Ast.Stmt.While`	Generates a while loop expression. The expression will consist of the `while` keyword followed by a single space and then the generated condition expression surrounded by parenthesis. Following a single space, the opening brace should be generated on the same line. If the statements is empty the closing brace should also be on the same line, otherwise each statement is generated on a new line with increased indentation followed by a closing brace on a new line with the original indentation. Returns `null`.	• `WHILE condition DO stmt1; stmt2; END` ◦ `while (condition) { stmt1; stmt2; }`
`Ast.Stmt.Return`	Generates a return expression. The expression will consist of the `return` keyword followed by a single space and the corresponding generated expression value. A semicolon should be generated at the end. Returns `null`.	• `RETURN 5 * 10;` ◦ `return 5 * 10;`
`Ast.Expr.Literal`	Generates a literal expression. The expression should generate the value of the literal found in the Ast. For characters and strings, remember that you will need to include the surrounding quotes. You do **not**, however, have to worry about converting escape characters back to their escape sequence (though a full language would absolutely need to). • Note: The BigDecimal class represents numbers with a specific precision, and therefore you need to pay close attention to the precision it has when writing test cases. It is recommended to use the `BigDecimal(String)` constructor for this reason so you know what the precision is. Returns `null`.	• `TRUE` ◦ `true` • `1` ◦ `1` • `"Hello World"` ◦ `"Hello World"`
`Ast.Expr.Group`	Generates a group expression. The expression used should be a generated expression surrounded by parentheses. Though the Analyzer requires the contained expression to be a binary expression, your generator should still work with other expression types. Returns `null`.	• `(1)` ◦ `(1)` • `(1 + 10)` ◦ `(1 + 10)`
`Ast.Expr.Binary`	Generates a binary expression. It should first generate the Ast's left expression, then generate the corresponding Jvm binary operator, and lastly generate the right expression. The binary operator should be generated with a single space on each side. Returns `null`.	• `TRUE AND FALSE` ◦ `true && false` • `"Ben" + 10` ◦ `"Ben" + 10`
`Ast.Expr.Access`	Generates an access expression. The name used should be the `jvmName` of the variable stored in the Ast. If a receiver is present, it should be generated first followed by a period. Returns `null`.	• `variable` ◦ `variable` • `object.field` ◦ `object.field`
`Ast.Expr.Function`	Generates a function expression. The name used should be the `jvmName` of the function stored in the Ast. It should be followed by a comma-separated list of the generated argument expressions surrounded by parenthesis. If a receiver is present, it should be generated first followed by a period. Returns `null`.	• `print("Hello world");` ◦ `System.out.print("Hello World");` • `"string".slice(1, 5)` ◦ `"string".substring(1, 5)`

Provided Code

The following files are provided to help you help implement the Generator.

- Source Files (`src/main/java/plc/project`)
  - Generator.java
- Test Files (`src/test/java/plc/project`)
  - GeneratorTests.java