**Object Oriented Programming Language and Environment- CS474**

**Contents:**

|  |  |  |
| --- | --- | --- |
| **Title** |  | **Page** |
|  |  |  |
| Team Members |  | 2 |
| Abstract |  | 3 |
| Introduction  Grammar Rules |  | 4  4 |
| Run Procedure |  | 9 |
| Result |  | 10 |
| Tools Used |  | 10 |

**Team Members:**

1. Ashwin Krithekavasan: [akrith2@uic.edu](mailto:akrith2@uic.edu)
2. Deexith Mysore Nagaraj: [dmysor3@uic.edu](mailto:dmysor3@uic.edu)
3. Lakshmi Divya Jillellamudi Kamala: [ljille2@uic.edu](mailto:ljille2@uic.edu)
4. Parinitha Ganapathisubramaniam Kannan: [pganap3@uic.edu](mailto:pganap3@uic.edu)

**Abstract:**

The project auto generates compile time error free code in Java language with the requirements defined in the configuration file. Configuration file will be having the number of classes, methods, expression and so on which basically specifies the code length of the program that was auto generated.

A considerable number of subset of grammars that has been used while generating the Java code from the already defined Java grammar. When the “Launcher” is run, the program auto generates syntactically correct but semantically meaningless Java code.

There exists class file for each individual grammar rule defined in the production rule. Each of these files will be extending the base class which keeps track of all the attributes used in the randomly generated code. GrammarParser is the class which decodes the production rule. The recursion on the production rule is restricted by assigning a value for each tokens or the non-terminals.

ClassCreator object is instantiated first, which leads to MethodCreator, which in directly calls up the MethodBodyCreator, and this keeps going until the class body is populated with the minimum requirement until it results in a compile time error free code.

**Introduction:**

We start generating the code by first creating a class. This class will be populated with zero or more fields and methods. If a method is created, then the method might have zero or more expressions in it. These expressions can to nested to a level defined. The code populates the method with as many statements as possible. The following explanations gives the detail implementation.

**Grammar Rules:**

Brackets Usage in Grammar:

[ …, …] - > There can be zero or one.

% …, … % - > There can one or more.

{ …, …., …. } - > There can be zero or more.

( ……… ) -> Compulsion i.e., must be present.

**Identifier Generator:**

Identifiers are randomly generated by selecting a combination of the strings and numbers. The random string generator generates a random name. Even though the name generated is random, it generates the name which obeys rules set by the Java complier. Once the identifier is generated then the same identifier won’t be generated again. Also, these identifiers used will never be of any predefined keywords present in Java language.

**Class:**

Grammar:

ClassDeclaration: Modifier **class** Identifier [ extends TypeList ] [ implements TypeList ] ClassBody

Random code generation starts from this grammar rule. This is the top level of the grammar parse tree. A class will have a modifier. Modifiers are “Public, Private, Protected “, so selection of modifier for the class is random. Also, since there can be multiple class in a .java file but out of those only one can be a public class, resection on the usage of two public class has been implemented. So, the randomly generated Java class will have at most one public class. Next class will be printed. From the identifier generator we get a random name. Now there is a chance of extending a class or implementing an interface. Here you can’t implement an interface or extend a class that has not yet been generated. TypeList will be having all the list of classes that can be extended or the interface that can be implemented. A restriction on this has been implemented in the code. At the end Class body will be called. Now we just generated the class. Let’s see how it will be populated.

**Class Body:**

Grammar:

ClassBody: { FieldDeclaration } { MethodDeclaration } DefaultConstructor{ ConstructorDeclaration }

A class can have mainly four things at the abstract level. One is fields, Methods, DefaultConstructor and Constructors. All these can be present, or it may not be present hence each of the fields are included inside the curved braces, but the Default Constructor is mandatory. Here Field Declaration, Method Declaration and Constructors can be of count zero or more. To generate the fields Identifier will generate randomly and type and modifier will be selected randomly. Methods name are also generated uniquely. No methods will have the same name. While the Constructor is generated, it will be having the same name as the class name. All these restrictions are taken care in the code. One more thing to observe here is that all the private methods or the fields won’t be given access to other class files. In simple words is that private methods or the fields will not be called in other class files. In DefaultConstructor ,we will have the modifier followed by the identifier name same as the class name and then followed by the constructor body.

**Field, Method and Constructor:**

Grammar:

FieldDeclaration: Modifier localVariableDeclaration  
MethodDeclaration: Modifier Types Identifier ( ParameterList ) MethodBody

ConstructorDeclaration: Modifier Identifier ( ParameterList ) ConstructorBody

Class fields will be having modifier and name generator. Modifier will be the same as for the class generator, just generates a random name. Since the generated field can be used in the class it must be initialized. Hence, we initialize the fields with some default random values. Int type will be assigned with integer value, float will be assigned with floats values and so on.

Methods declaration with have the modifier, the return type and the method name which will be randomly generated obeying the java complier rules. Signature can have many or zero parameters in it. Each method might have method body or might not have any. If there are more than one method generated, then there won’t a clash between any of the other methods as the next method generated won’t be having the same number or same type of signature as the previously generated method. Probability for this case has been kept low.

Constructor will again have modifier and here the identifier, we won’t take it from the random identifier generator. Rather we will take the name of the class in which it has been declared. Again, the constructor can have zero or more parameters to it. If there are more than one constructor generated, there won’t a clash between any of the constructors as the next constructor can’t have the same number or same type of parameter as the previously generated constructor. This case also will have the probability to a lower value.

**Constructor Body and Method Body:**

Grammar:

MethodBody: Block

ConstructorBody: Block

Block: BlockStatement

BlockStatement: { localVariableDeclaration } { Statement }

Method body will have blocks of statements. It may also include zero or more local variable declarations. If a variable is declared then it should be initialized to some default values, else if these variables are used in the statements then it will lead to a compile time error, so to overcome this we have implemented rules so that a variable can’t be used before declaration and initialization. Hence, we will be initializing the variables to default value which will again be randomly generated. Also, a restriction has been placed for the usage of the variables before the declaration. Since in-line functions are not allowed in Java, a method can’t have another method inside in it. Hence restriction on creation of nested methods is implemented using have production rules. The same applies to the constructor body.

**Statements:**

Grammar:

% Assignment,TerminalExpression,IfThenStatement,DoStatement,WhileStatement,ForStatemen%

Assignment: Identifier = Expression

TerminalExpression: Expression

IfThenStatement: if ( TestingExpression ) Statement [ else Statement ]

DoStatement: do Statement while ( TestingExpression )

TestingExpression: Identifier RelationalOperation RHS

WhileStatement: while ( TestingExpression ) Statement

ForStatement: for ( LoopAssignment TestingExpression ; [ PostIncrementDecrement ] ) Statement

These are the possible set of statements that the random generator generates. Limit for each of the statement usage has been defined in the program.

Assignment means we assign an identifier that has already been generated and it will be randomly picked up and each of variables used types are recorded and we assign it to an expression which evaluates to the variable type if the expression evaluates to some other type we type cast it to the RHS type. If the for instance if the variable is of type int and we have an expression as 7/3 is results in float, then it will be converted into int.

For Statement should have three main parts in it, initialization, condition, and the increment or decrement. Here condition should evaluate to a Boolean value, hence we are using the testing expression, which indirectly checks goes to logical expression. Loop Assignment has a choice where it can generate a new variable, or it can just use the already defined variable. After this it might have statements.

If statement should have a conditional expression which evaluates to a Boolean type. So, here we select the expression from the list of expressions that evaluates to a Boolean result and randomly select it and place it inside the parentheses. There might be a chance that the statements after the IF might again call up the IF block. There might be a good chance of ending with an infinite nested IF statements. To avoid this, restrictions have been placed for the depth of the nested IF statements. Since there might be an ELSE block we have kept else block in a square bracket so the ELSE can be include or it might not be included.

Same has been implemented for the While Statement and for DO While. Since the conditional statements works on the Boolean expression we are using only Relational operators only so that it evaluates to “true” or “false”.

**Expressions:**

Grammar:

Expression: % NumericExpression,TestingExpression,Logical,StringExpression,LiteralExpression,Identifier %

NumericExpression: % PreIncrementDecrement,PostIncrementDecrement,ArithmeticExpression %

ArithmeticExpression: Expression ArithmeticOperation Expression

PreIncrementDecrement: PrePostOperations Identifier

PostIncrementDecrement: Identifier PrePostOperations

LogicalExpr: Expression LogicalOps Expression

TernaryExpression: Expression ? Expression : Expression

StringExpression: Expression StringOperations Expression

LiteralExpression: % IntegerLiteral,FloatLiteral,StringLiteral,CharacterLiteral %

Now all these expressions can appear in constructor body or in method body. We can classify the expressions into Numeric Expression, Arithmetic Expression, Logical Expression, Ternary expression, String Expression, Literal Expression. These are the expression that are covered in the code generated. Numeric expression will be having the increment, decrement or arithmetic expression. Since the increment and decrement operations cannot operate on strings or characters, we have restricted the numeric expression to operate only on the int and float types. But the arithmetic operator “+ “is still applicable to strings, we have defined a different set of grammar rule as String Expression, where we are using addition (concatenation) of two strings and assigning a string to other. Logical expression covers all the logical operations. Ternary expression the first part must evaluate to a Boolean value and then based on the result of the value we evaluate the first or the second expression. Thus, we define the first part as the logical expression so that the result will always yield Boolean result. Literal expression is just the base, which generates the default values based on the type it’s going to get assigned. Also since the integers or float or double cannot be started with zero, an restriction has also been implemented where none of the numbers randomly generated numbers will start with zero.

**Switch:**

Grammar:

SwitchStatement: switch ( Expression ) \u{ { CaseStatement } [ DefaultStatement ] \u}

CaseStatement: case Expression : Statement

DefaultStatement: default : Statement

Switch statements comes under the expression and it has the case statements and default statement. Here we have considered the case where we can have zero or more case statements and there might be only one default statement hence we have included that in the square brackets. There might be cases where each of the case statements executed can be broke or it may continue executing the next case statements. Even though this is not logical, it’s not a compiler time error. Hence, we have given as option for the break to be included at the end of each case statements.

**Break and Continue:**

Grammar:

LoopStatement: % ContinueStatement,BreakStatement %

BreakStatement: break;

ContinueStatement: continue;

Usually break and continue comes inside a loop statements. Hence, we have restricted their usage only inside the loop statements. If break and continue appears outside loop statements, then it will lead to a compile time error.

**Return Statement:**

Grammar:

ReturnStatement: return Type;

We need to return the value from a method of the method type only else needs with compile time error. Hence, a restriction has been implemented which restricts a method to return its type only. If the method is of type void, then return won’t be even printed.

**Run procedure:**

* Make sure the dependency library issues are all resolved.
* Specify the path where the random Java program must be generated in config file.
* Specify the number of class, methods user required in the config file.
* Once the above has been mentioned properly then run “Launcher”
* Random Java program will be generated with auto creation of respective class files in a different folder specified by the user in the config file.

**Running using Gradle:**

* To build use the terminal command **"gradle build"** to build the project.
* **"gradle run"** to run the project and check the results in the terminal.

**Running using SBT:**

* Open terminal, type "**sbt**"
* Type “**compile”** to compile the entire application using SBT.
* Type "**run**" to run the application using SBT.

**Result:**

A syntactically correct but semantically meaningless Java code with defined restriction on the number of class files or methods or statements inside a method is generated.

**Tools Used:**

Language: JAVA 8

Grammar Defined: JAVA

IDE: IntelliJ IDEA 2017