MŰEGYETEM 1782

MODEL-BASED SOFTWARE DEVELOPMENT – PRACTICE 4 – ANTLR

Semantic analysis and code generation

Dr. Ferenc Somogyi

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

1117 Budapest, Magyar tudósok körútja 2.
Q. ép. B. szárny 207. • www.aut.bme.hu
Telefon: 463-2870 • Fax: 463-2871

I. INTRODUCTION

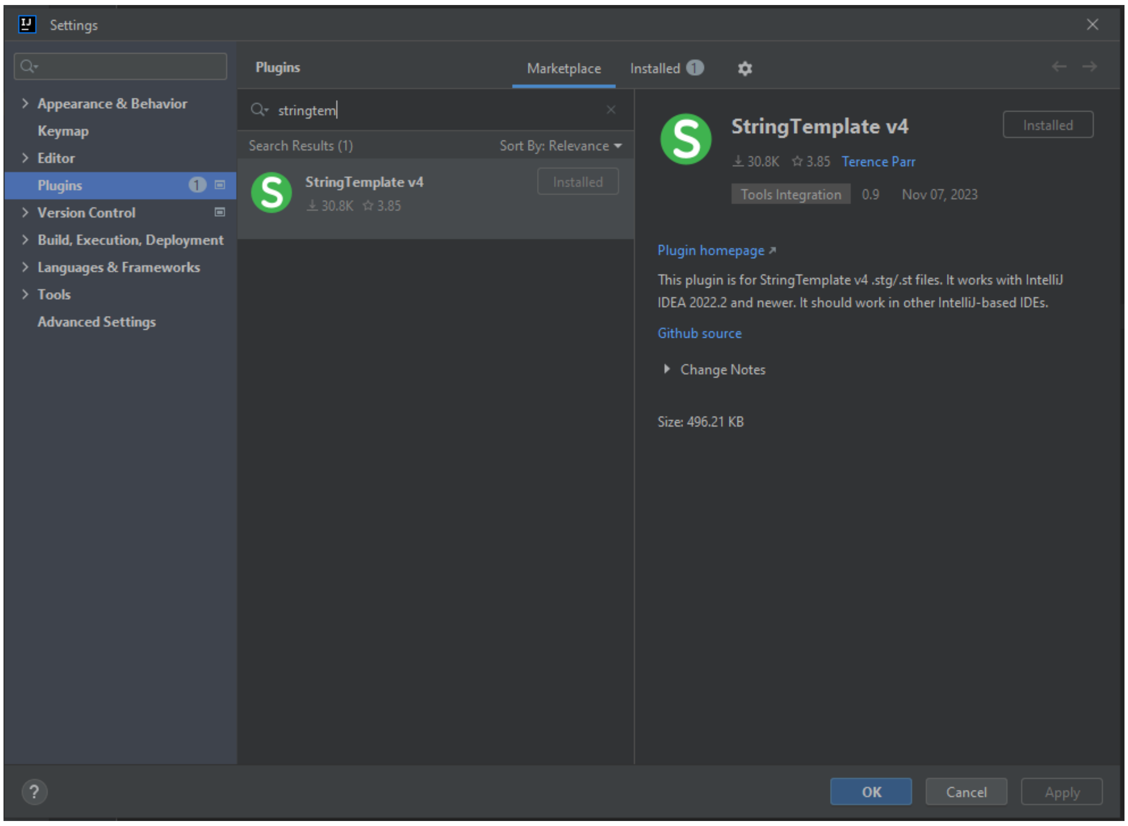
The purpose of this document is to build a semantic analyzer of the prototype programming language (TinyScript) created in Practice 2 and a code generator that can generate Java code from TinyScript input. The code for the semantic analyzer can be found in the initial project (we will look at it in this guide), but **parts of the code generator** will have to be written in **Homework 3**. In the course of this practie, we will be introduced to the following:

- Using the visitor pattern to walk the syntax tree
- Semantic analysis: implementing symbol tables and type checking
- Creating a template-based code generator using StringTemplate

Important: This guide (and subsequent ones) is intended for ANTLR 4. The previous version, ANTLR 3 is similar in many ways to ANTLR version 4, but also different in many more. When using stackoverflow or other helpful sites, it is important to know which version a question refers to.

INSTALLING STRINGTEMPLATE IN AN INTELLIJ ENVIRONMENT

StringTemplate[1] is a template-based code generation tool. In the initial project, as in ANTLR, there is a jar file under the *lib* folder which is needed to run it. Similar to what we saw in the previous practice, make sure the jar file is there in the project's dependencies (***File --> Project Structure... --> Modules --> Dependencies***), and if not, add it. Also install the appropriate IntelliJ plugin, also similar to ANTLR. ***File --> Settings...***, select ***Plugins*** from the side, search for StringTemplate and install the plugin (then restart IntelliJ):



This plugin is for StringTemplate v4 .stg/.st files. It works with IntelliJ IDEA 2022.2 and newer. It should work in other IntelliJ-based IDEs.

[1] https://www.stringtemplate.org/

II. Semantic Analysis

Download and open the initial project! In the following, we will briefly review the semantic analysis process based on the source code; the entire semantic analyzer is already written.

Symbol table

In TinyScript, we can define variables and functions, which will be our symbols. Because a function has more properties than a variable (it can have parameters) and behaves differently (it can be called), they are treated separately in the scope. Under the **tinyscript.symboltable** package we can find the handling of variables, functions, and scopes. The *addParameter* function of the *FunctionSymbol* class is needed to provide a more customizable error message in case of duplicate parameter names. The *TSType* class describes a type of the type system, this will be discussed later.

```
public class FunctionSymbol {
    public String name;
    public TSType returnType;
    public final HashMap<String, VariableSymbol> parameters = new HashMap<>();

    public FunctionSymbol(String name, TSType returnType) {
        this.name = name;
        this.returnType = returnType;
    }

    public boolean addParameter(String name, VariableSymbol paramSymbol) {
        if (parameters.containsKey(name)) {
            System.out.println("parameter '" + name + "' is already in scope");
            return true;
        }

        parameters.put(name, paramSymbol);
        return false;
    }

    @Override
    public String toString() { … }
} }
```

```
public class VariableSymbol {
    public TSType type;
    public String name;


    public VariableSymbol(TSType type, String name) {
        this.type = type;
        this.name = name;
    }

    @Override
    public String toString() { … }
}
```

The *Scope* class handles the parent scope (*previous*) and supports the usual lookup and insert operations (*getXY* and *addXY* functions).

```
public class Scope {
    private final Scope previous;
    private final HashMap<String, VariableSymbol> variables = new HashMap<>();
    private final HashMap<String, FunctionSymbol> functions = new HashMap<>();

    public Scope getPrevious() {
        return previous;
    }

    public Scope(Scope previous) {
        this.previous = previous;
    }

    public VariableSymbol getVariable(String name) { … }

    public FunctionSymbol getFunction(String name) { … }

    public boolean addVariable(String name, VariableSymbol symbol) { … }

    public boolean addFunction(String name, FunctionSymbol symbol) { … }
    …
}
```

TYPE SYSTEM

The structure of the typesystem is described in the **tinyscript.typesystem** package. A type can have parent types, and we can check the compatibility of two types along this relation (*isCompatibleWith* function).

```
public class TSType {
    public String name;
    public HashSet<TSType> parents;

    public TSType(String name) {
        this.name = name;
        parents = new HashSet<>();
    }


    public boolean isCompatibleWith(TSType t) {
        return this == t || parents.stream().anyMatch(p -> p.isCompatibleWith(t));
    }
}
```

The *TypeSystem* class contains all the built-in types, and the relationships between them are specified here. The *ERROR* type will have as parent all other built-in types, to avoid escalation of type errors. We could also add our own type (*createType* function, e.g. classes), but this is currently not possible in TinyScript. We would then also need to handle the parent relationships of the *NULL* and *ERROR* types, for example a new class would need to be the parent of both.

```
public class TypeSystem {
    private final HashMap<String, TSType> types;

    public TypeSystem() {
        types = new HashMap<>();
        initializeConstants();
    }
    public TSType BOOLEAN;
    public TSType INT;
    public TSType STRING;
    public TSType NULL;
    public TSType ERROR;
    public TSType VOID;

    private void initializeConstants() {
        this.ERROR = new TSType("ErrorType");
        this.STRING = new TSType("string");
        this.INT = new TSType("int");
        this.BOOLEAN = new TSType("bool");
        this.NULL = new TSType("NullType");
        this.VOID = new TSType("void");

        this.NULL.parents.add(this.STRING);
        this.ERROR.parents.add(this.INT);
        this.ERROR.parents.add(this.BOOLEAN);
        this.ERROR.parents.add(this.NULL);
        this.ERROR.parents.add(this.VOID);

        types.put(this.ERROR.name, this.ERROR);
        types.put(this.INT.name, this.INT);
        types.put(this.STRING.name, this.STRING);
        types.put(this.BOOLEAN.name, this.BOOLEAN);
        types.put(this.NULL.name, this.NULL);
        types.put(this.VOID.name, this.VOID);
    }

    public TSType getType(String name) {
        if (types.containsKey(name))
            return types.get(name);

        return this.ERROR;  // type error escalation
    }

    …
}
```

The *TypeSystemHelper* class determines the type of an expression based on the type system and symbol table (scope). The type system and current scope will be passed by the visitor, the latter of course depending on where we are in the code. For more complex languages, additional helper classes and data structures can be created.

```
public class TypeSystemHelper {
    public static TSType getType(TinyScriptParser.ExpressionContext ctx, TypeSystem ts, Scope scope) {
        …
    }

    public static TSType getType(TinyScriptParser.PrimaryContext ctx, TypeSystem ts, Scope scope) {
        …
    }

    public static TSType getType(TinyScriptParser.FunctionCallContext ctx, TypeSystem ts, Scope scope) {
        …
    }

    public static TSType getType(TinyScriptParser.LiteralExpressionContext ctx, TypeSystem ts, Scope scope)
    {
        …
    }
}
```

Walking the syntax tree using the visitor pattern

Each parser rule in the ANTLR grammar generates its own Context class (e.g., *IfStatementContext*), which provides a typed API to retrieve its children in the tree. The basic idea of the visitor pattern is to decouple the walking of the data structure from what we need to do at each node. ANTLR defaults to depth-first search (DFS), which can be overridden in part or in whole by reorganizing the *return* statements in the *visit* functions, but this is often unnecessary.

The *TinyScriptSemanticAnalyzer* class in the **tinyscript** package is responsible for performing the entire semantic analysis. It handles the *scope* (which will be dynamically changed) and the type system (*ts*). The *exceptionHandler* is responsible for handling error messages. The *currentFunction* attribute is needed to know which function we are currently in, for example, when handling function parameters and return statements.

```
public class TinyScriptSemanticAnalyzer extends TinyScriptBaseVisitor<Object> {
    private Scope scope;
    private TypeSystem ts;
    private final TinyScriptExceptionHandler exceptionHandler;
    private FunctionSymbol currentFunction = null;

    public TinyScriptSemanticAnalyzer(TinyScriptExceptionHandler exceptionHandler) {
        this.exceptionHandler = exceptionHandler;

        scope = new Scope(null);
        ts = new TypeSystem();
    }
…
```

In the following, we will not go through the semantic analyzer in its entirety but will highlight some of the more important details. When entering a function definition (*FunctionDefinitionContext*), we first insert the function into the current scope and open a new local scope for the function, since there may be local member variables that can only be accessed from inside the function. Next, we call the function's statements (*super.visitFunctionDefinition* is extracted, we don't return with it immediately), and then we reset the scope. Notice that we can access the functions we have defined as parser rules in the grammar in the *FunctionDefinitionContext* class!

```
    @Override
    public Object visitFunctionDefinition(TinyScriptParser.FunctionDefinitionContext ctx) {
        var name = ctx.functionName().getText();
        var returnType = ts.getType(ctx.returnType().getText());
        currentFunction = new FunctionSymbol(name, returnType);
        addFunctionToScope(ctx, currentFunction);

        scope = new Scope(scope);
        System.out.println("Function entry");

        var result = super.visitFunctionDefinition(ctx);

        System.out.println("Function exit");
        System.out.println(scope.toString());
        scope = scope.getPrevious();

        currentFunction = null;
        return result;
    }
```

For a return statement, we need to check if we are inside a function, again using the *currentFunction* attribute mentioned earlier. We also check the return type.

```
    @Override
    public Object visitReturnStatement(TinyScriptParser.ReturnStatementContext ctx) {
        if (currentFunction == null) {
            addException(ctx, "return statements can only be present inside a function");
        }
        else {
            var type = TypeSystemHelper.getType(ctx.expression(), ts, scope);
            if (!type.isCompatibleWith(currentFunction.returnType))
                addException(ctx, "return statement type " + type.name + " is not compatible with function
return type " + currentFunction.returnType.name);
        }

        return super.visitReturnStatement(ctx);
    }
```

When declaring a variable, we have several tasks: if we use the "var" keyword, we have to check that there is an initial value; if there is an initial value (but no "var" keyword), then the expression describing the initial value must be compatible with the type of the variable. In addition, the variable must be inserted into the current scope.

```
    @Override
    public Object visitVariableDeclaration(TinyScriptParser.VariableDeclarationContext ctx) {
        var varName = ctx.varName().getText();
        // var
        if (ctx.VAR() != null) {
            if (ctx.expression() == null)
                addException(ctx, "variables declared using 'var' must have an initial value");
            else {
                var type = TypeSystemHelper.getType(ctx.expression(), ts, scope);
                var symbol = new VariableSymbol(type, varName);
                addVariableToScope(ctx.varName(), symbol);
            }
        }
        // not var
        else {
            var type = ts.getType(ctx.typeName().getText());

            if (ctx.expression() == null) {
                var symbol = new VariableSymbol(type, varName);
                addVariableToScope(ctx.varName(), symbol);
            }
            else {
                var expressionType = TypeSystemHelper.getType(ctx.expression(), ts, scope);
                if (!expressionType.isCompatibleWith(type))
                    addException(ctx,  "an  expression  of  type  '"  +  expressionType.name  +  "'  is  not
assignable" + " to variable '" + varName + "' of type " + type.name);
                else {
                    var symbol = new VariableSymbol(type, ctx.varName().getText());
                    addVariableToScope(ctx.varName(), symbol);
                }
            }
        }
        System.out.println("Var declaration visited");
        System.out.println(scope.toString());
        return super.visitVariableDeclaration(ctx);
    }
```

The visitor displays on console the main steps of the analysis and any errors found. It is worth running it (*TinyScriptRunner* class) and following exactly how it works and testing the parser by modifying the *input.tys* file.

During the semantic analysis, when we need to build a symbol table (or similar data structure), we usually create several visitors. The first visitor builds the symbol table and collects any errors while building the data structure (e.g. if two global functions have the same name), and then the second (or even the Nth, there can be more) visitor performs such semantic analysis checks on the built symbol table that require the full symbol table (e.g. whether a function with a given name exists). In the initial project we have circumvented this, by changing the order of walking the syntax tree (see visitProgram function) the functions are inserted into the scope first. If we assume that we cannot call any other function inside a function, this is a good solution, otherwise we would need another visitor that first builds the symbol table based on the above.

III. Code Generation

In the case of TinyScript, the transformation and optimization phases are omitted, thus, code generation follows semantic analysis. Code generation is based on the visitor pattern. A template-based transpiler will be created to transform the code written in TinyScript into Java code using StringTemplate[2] technology. We will always generate exactly one Java class (e.g. *TinyScriptOutput*) with a main function; additional functions written in TinyScript will be generated in this class. The generated file is created in the **tinyscript.generated** package by default.

Since TinyScript and Java have a very high syntactic similarity, it is easier to generate code. We can exactly copy almost all of the statements, with a few exceptions (type names) the syntax of the statements is the same in both languages. Part of the code generation can be found in the initial project, which will be reviewed below. The rest of the code generation is part of the homework.

IMPORTANT: if a Java file containing errors is generated, IntelliJ IDEA (since a Java project is open) may not allow the application to run. In this case, delete the generated file and run the main function of the TinyScriptRunner class again.

Let's start by using StringTemplate: under the **tinyscript.codegen.stringtemplate** package, there is a so-called group file (*JavaGen.stg*) that can contain any number of templates. When writing the actual code (visitor), we can provide the templates with parameters (e.g. *packageName*, *functions*, etc.) and embed their contents in other templates. Within a template, we can insert a parameter using the < > characters. The *java* template contains the skeleton of the generated code: a package and class definition, functions within the class, and a main function. The **function generation** is not solved, it is **part of the homework**. The *statements* template is very simple, it collects statements (arbitrary strings) one after the other, separated by newline characters. We could use StringBuilder in the code instead, but one of the advantages of StringTemplate is that there is no need for manual string concatenation, instead we use parameterized templates.

[2] https://github.com/antlr/stringtemplate4/blob/master/doc/index.md

```
java(packageName, className, functions, mainBody) ::= <<
package <packageName>;

public class <className> {
    <functions; separator="\n">

    public static void main(String[] args) {
        <mainBody>
    }
}
>>

statements(statement) ::= <<
<statement; separator="\n">
>>
```

The *TinyScriptCodeGenerator* class in the **tinyscript.codegen** package is – like the semantic analyzer – a visitor, performing code generation. This requires the StringTemplate *groupFile*, the main template (*javaTemplate*) contained in it, and a separate attribute for statements inside the main function (*mainStatementsTemplate*), which uses the above-mentioned *statements* template. By overriding the *visit* function, we specify the generation process: we instantiate the *statements* template, walk the syntax tree, and finally add the collected statements to the main (*java*) template as a parameter (*mainBody*). Finally, the generated code will be the text rendered by the *java* template.

```
public class TinyScriptCodeGenerator extends TinyScriptBaseVisitor<Object> {
    public String generatedCode = "";
    private final STGroupFile groupFile;
    private final ST javaTemplate;
    private ST mainStatementsTemplate;

    public TinyScriptCodeGenerator(String stGroupPath, String javaTemplate, String packageName, String className) {
        this.groupFile = new STGroupFile(stGroupPath);
        this.javaTemplate = groupFile.getInstanceOf(javaTemplate);
        this.javaTemplate.add("packageName", packageName);
        this.javaTemplate.add("className", className);
    }

    @Override
    public Object visit(ParseTree tree) {
        mainStatementsTemplate = groupFile.getInstanceOf("statements");

        var result = super.visit(tree);
        javaTemplate.add("mainBody", mainStatementsTemplate);
        generatedCode = javaTemplate.render();
        return result;
    }
…
```

The syntax tree is walked by creating a string (*processStatement*) from each statement, replacing the two non-Java compatible type names (string and bool). Of course, in a more complex language (or where the syntax of the target language would not be so similar) this would be a much more complex task, here we simplify this step. We have to pay special attention to the *if* and *while* statements, so that we don't walk their statements again, otherwise we would duplicate the generated code, since the inside of these statements also contains statements. The *getFullSourceText* helper function is used to include whitespace and newline characters that are not included in the syntax tree in the generated code.

```
@Override
    public Object visitStatement(TinyScriptParser.StatementContext ctx) {
        mainStatementsTemplate.add("statement", processStatement(ctx));

        if (ctx.ifStatement() == null && ctx.whileStatement() == null)
            return super.visitStatement(ctx);
        else return null;
    }

    private String processStatement(TinyScriptParser.StatementContext ctx) {
        return getFullSourceText(ctx).replace("string", "String").replace("bool", "boolean");
    }

    private String getFullSourceText(ParserRuleContext context) {
        CharStream cs = context.start.getTokenSource().getInputStream();
        int stopIndex = context.stop != null ? context.stop.getStopIndex() : -1;
        return cs.getText(new Interval(context.start.getStartIndex(), stopIndex));
    }
```

We could also use a symbol table or type system when generating code, which is common in more complex languages. For example, in the current task, the conversion of type names between TinyScript and Java (string --> String and bool --> boolean) could be done in a more elegant way by using and extending the TypeSystem class, but here we opted for simplicity.

The code generation is parameterized in the *TinyScriptRunner* class (input and output files, StringTemplate template) and is executed by running the main function after the semantic analysis. The code generator in the initial project is **intentionally bugged**, it also generates the function statements inside the main function (e.g. *return a +b;*), we will have to fix this in the homework! The example input file in the initial project (*input.tys*) also has an example output file (*TinyScriptOutputExample.java*).