MŰEGYETEM 1782

MODEL-BASED SOFTWARE DEVELOPMENT – PRACTICE 2 – ANTLR

Lexical and syntax analysis

Dr. Ferenc Somogyi

I. Introduction

The purpose of this document is to get acquainted with the first two phases of classic compilation in practice. To achieve this, we will use the ANTLR parser generator. During this practice, we will learn the basic use of ANTLR:

- Using ANTLR in an IntelliJ environment
- Defining the grammar: lexer and parser rules
- Display a derivation tree based on arbitrary input
- Rudimentary implementation of a prototype programming language (TinyScript) based on the above

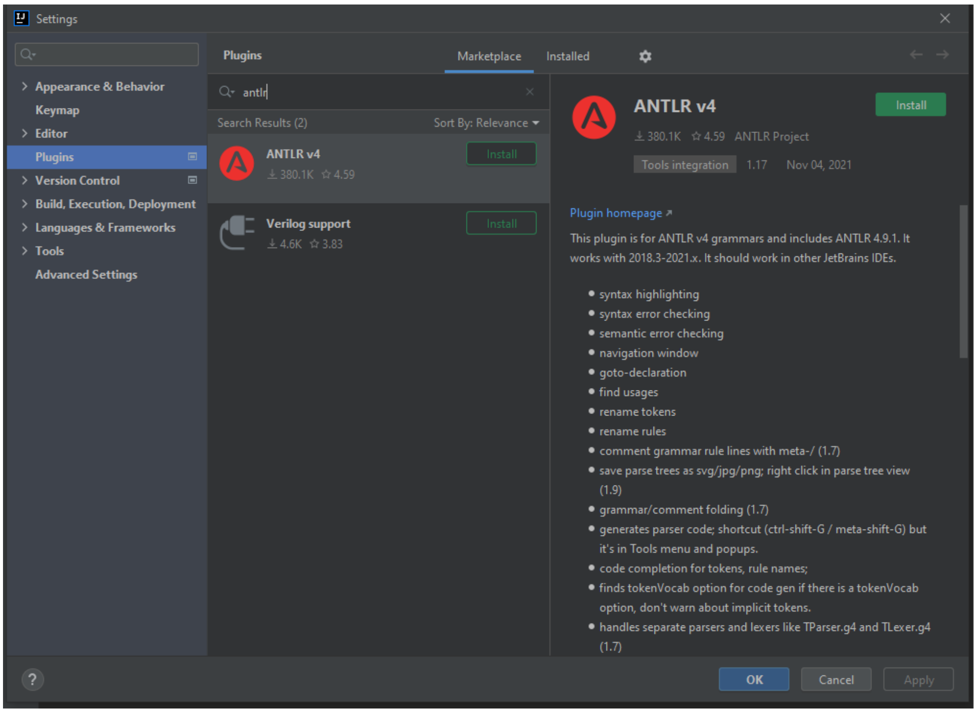
Important: This guide (and subsequent ones) is intended for **ANTLR 4**. The previous version, **ANTLR 3** is similar in many ways to ANTLR version 4, but also different in many more. When using stackoverflow or other helpful sites, it is important to know which version a question refers to.

Installing ANTLR in an IntelliJ environment

Download **IntelliJ IDEA Community Edition**: https://www.jetbrains.com/idea/download/#section=windows

We will also need a **JDK**, which can be downloaded from here for example (alternatively, IntelliJ can automatically offer a different JDK when you create a new Java project, which is also good):
http://www.oracle.com/technetwork/java/javase/downloads/index.html

It is highly recommended to install the ANTLR plugin - otherwise you will have to do the lexer and parser generation using the command line. With the plugin, IntelliJ provides many useful functions for editing and “debugging” ANTLR grammars. The plugin needs to be installed once, after that it is available for all projects. Go to ***File --> Settings...***, select ***Plugins*** from the side, search for ANTLR, and install the plugin (then restart IntelliJ):



This plugin is for ANTLR v4 grammars and includes ANTLR 4.9.1. It works with 2018.3-2021.x. It should work in other JetBrains IDEs.

- syntax highlighting
- syntax error checking
- semantic error checking
- navigation window
- goto-declaration
- find usages
- rename tokens
- rename rules
- comment grammar rule lines with meta-/ (1.7)
- save parse trees as svg/jpg/png; right click in parse tree view (1.9)
- grammar/comment folding (1.7)
- generates parser code; shortcut (ctrl-shift-G / meta-shift-G) but it's in Tools menu and popups.
- code completion for tokens, rule names;
- finds tokenVocab option for code gen if there is a tokenVocab option, don't warn about implicit tokens.
- handles separate parsers and lexers like TParser.g4 and TLexer.g4 (1.7)

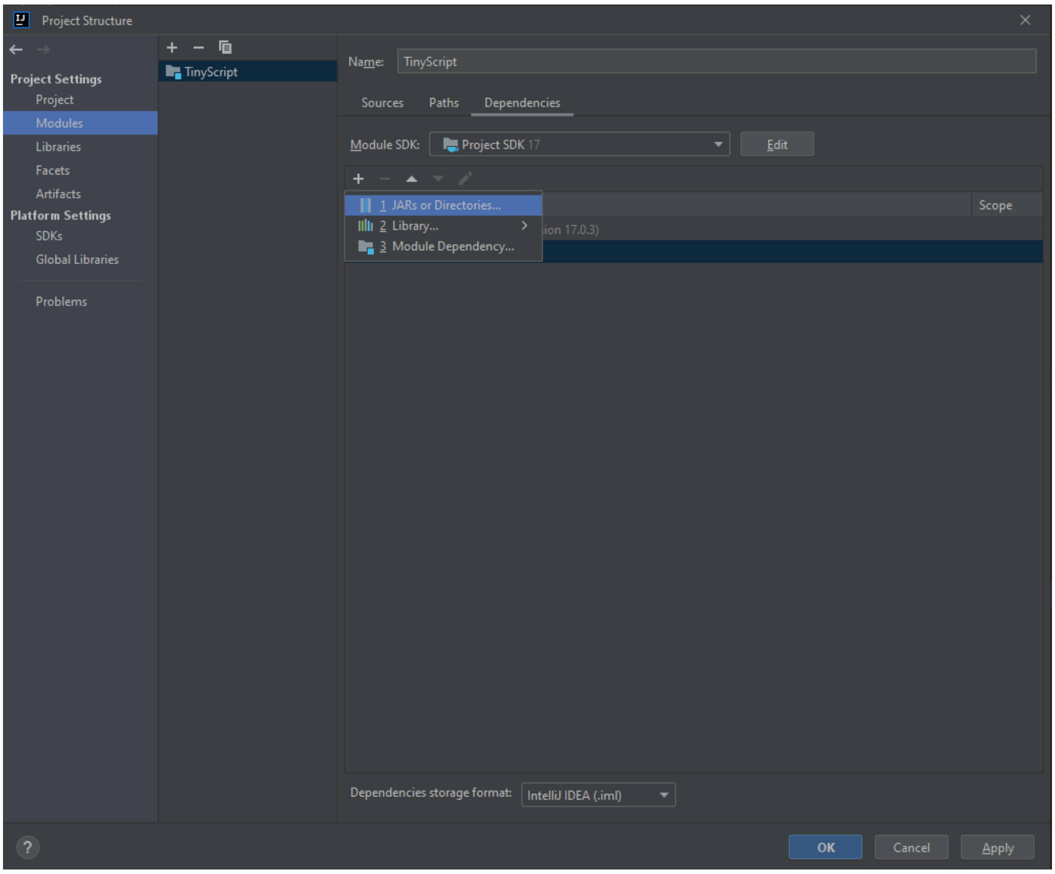
ANTLR supports several target languages (e.g. C++, C#, Java, Python) and there are integrations with several development environments (e.g. Visual Studio Code). In this guide we use Java and IntelliJ, but you could choose other languages.

Next, open the initial project (**TinyScript**)! Under the **lib** folder, you will find the ANTLR runtime (a *jar* file), which is needed to access the runtime functions (e.g. build a syntax tree at runtime, do the semantic analysis, etc.). There are four files in the *tinyscript* and *tinyscript.exceptions* packages:

- **input.tys**: contains an example input text; this will be processed later
- **TinyScriptRunner.java**: contains a *main* function; this is where we will later use the runtime jar and process the input text using the generated lexer and parser
- **TinyScriptException.java** and **TinyScriptExceptionHandler.java**: an exception also contains the exact location of the error (row, column); all exceptions are collected, instead of just the first error

If a new project is created, the runtime *jar* file must be added to the project manually (or using a build system) as follows:

- Download the ANTLR runtime jar: https://www.antlr.org/download
- Copy the jar into the folder structure of the project (e.g. the **lib** folder)
- Add the jar as a dependency to the project: **File --> Project Structure... --> Modules --> Dependencies**


Name: TinyScript

Module SDK: Project SDK 17

Dependencies storage format: IntelliJ IDEA (.iml)

When using the initial project, the above steps should only be done if the jar file in the lib folder is not recognized under Dependencies. Be sure to check this!

II. Specification of the TinyScript language

TinyScript is a prototype programming language that supports some basic features common to programming languages. The language is not intended to be a full-fledged programming language, as it would require a lot of work that is beyond the scope of this practice. However, the language does give us an insight into the world of compilers in practice, i.e. how we can describe and process certain concepts that we have learned from programming. By the end of this practice, we will have completed the syntax of the TinyScript language, i.e. we will be able to read the code written in the language and detect syntax errors. Semantic analysis and code generation will be covered in a later exercise.

In TinyScript, we can define functions and describe how the program runs within a *main* block. We can define functions before and after the *main* block. The language supports the following features:

- **main** block: **arbitrary statements** can be included within the block
- **function** definition: a function has a **name**, **return type**, **parameters** (optional), and a **body**
- of the primitive types, the language supports only **int**, **string**, **bool**
  - the **null** value also exists in the language, strings are **nullable**; later, classes can be introduced which should also be nullable
- **variable declaration**
  - using the **var** keyword or defining its **type**
  - can have an **initial value**, when using the var keyword, it must always have an initial value
- **variable assignment**: an arbitrary expression can be assigned as a value to an existing variable
- **while** loop: condition and body
- **if** statement: condition, body, optionally an **else** branch
- **function call**
  - parameters must be passed correctly
- **return** statement: can only be used inside function body, not inside main block
- **expressions**
  - can be used in: variable declaration, variable assignment, function call parameter, while and if condition, return statement
  - **literal** values (null, int, string, bool), variable **reference** by name
  - **grouping**, **function call**
  - **arithmetic** (+, -, *, /) and **logical** operators (>=, <=, >, <, ==, !=)
- **comments**: single and multiline

If you want to see an example of the syntax of the language, you should look at the **input.tys** file in the initial project.

III. Writing the grammar

In the **tinyscript** package, ***right-click --> New --> File***, the grammar name should be **TinyScript.g4**. g4 is the file extension of ANTLR, the "g" stands for grammar, the "4" for the main ANTLR version. In the following, we will first look at the general structure of an ANTLR grammar, and then write the TinyScript grammar.

About ANTLR grammars in general

The grammar always starts with the name of the grammar, using the keyword ***grammar***, and ending with a semicolon:

```
grammar TinyScript;
```

An ANTLR grammar is built from rules, similar to a context-independent (**CF**) grammar. Like CF grammars, ANTLR rules have left and right sides. Rules must be terminated by semicolons. The rules can be divided into two groups:

- **Lexer rules**: only text (*terminal symbols*) can be on the right side, between apostrophes. On the left side is the name of the rule (*non-terminal symbol*), which is also the name of the resulting token in the lexical analysis. The left side is written in all capital letters.
- **Parser rules**: on the right side we can have the name of a *lexer* or a *parser* rule, in any combination (*terminal* and *non-terminal symbols*). The right side can also be raw text, but as a matter of best practice, it is advisable to outsource all such rules to lexer rules. There are code organization and performance reasons for this. The left side is the name of the rule (*non-terminal symbol*). The left side is traditionally written in camelCase.

When specifying rules, we can use quantifiers (?, *, +, etc.) familiar from the world of regular expressions, similar to the EBNF notation. ANTLR also provides us with so-called **parser actions**, which give the parser special instructions when it finds text matching the given rule. For example, the *skip* action can be used to tell the parser not to include a given rule (token) in the derivation tree by skipping it.

There are two ways of thinking about constructing the grammar. The *bottom-up* method is to first build the basic building blocks of the language (lexer rules), and then specify the structure of the language (parser rules). The *top-down* method is to first specify the structure of the language (parser rules), and then specify what each token looks like (lexer rules). You can use either method, or a mixture of the two.

In an ANTLR grammar file, the parser rules are typically written first, followed by the lexer rules, regardless of which ones are created first. There's no functional reason for this, it is just convention in most grammars so it makes sense for us to follow it.

ANTLR also allows us to specify lexer and parser rules in a separate grammar file. Then we can import the .g4 file of the lexer into the .g4 file of the parser (see "options" and "tokenVocab").

Creating the grammar of TinyScript

The grammar is now written in a *bottom-up* way, first the lexer rules, then the parser rules. We start with the keywords, operators, and other special characters (e.g. parentheses, semicolons) used in TinyScript.

```
LPAREN: '(';
RPAREN: ')';
LCURLY: '{';
RCURLY: '}';
EOS: ';';
COMMA: ',';

EQ: '==';
NEQ: '!=';
NEG: '!';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';

ASSIGN: '=';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';

IF: 'if';
ELSE: 'else';
WHILE: 'while';
VAR: 'var';
MAIN: 'main';
RETURN: 'return';
VOID: 'void';
```

The following parser rules can be written to describe literal values. The INT rule can be specified in the grammar (so that it does not start with 0), but such special requirements (if they are important for the language at all) are typically checked in the semantic analysis rather than overcomplicating the grammar.

```
NULL : 'null';
TRUE: 'true';
FALSE: 'false';
STRING: '"' (~[\r\n"])* '"';
INT: [0-9]+;
```

Finally, let's write some more general lexer rules (most of which we saw in previous lectures - in a slightly different form) that are needed in more places later.

```
ID:     [a-zA-Z][a-zA-Z0-9_]*;
WS:     (' '| '\t' | '\n' | '\r') -> skip;
COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
```

For lexer rules, the **order is important**, because ANTLR tries to generate tokens from lexemes in the order in which the rules are specified. Thus, always put the more general rules (ID, WS etc.) at the end of the file!

Next are the parser rules, which should conventionally be placed at the beginning of the file. The **initial rule** (*program*) describes the structure of the language, i.e. that there can be function definitions before and after the *main* block.

```
program
       :      functionDefinition* main functionDefinition*
       ;
```

The formatting of parser and lexer rules in the grammar (.g4) file does not matter.

Functions are defined by the following parser rules. The number of function parameters (separated by commas) is arbitrary, can also be 0. The *parameterType* and similar rules (with only an ID on the right side) are needed for a better structure, so that it is easier to process and read the syntax tree compared to using only ID-s everywhere. In the latter case we would have to remember, for example, the order of the parameter type and name during processing. In the *returnType* rule, the *VOID* token is highlighted because it can be used as a return type, but not in other places (e.g. parameter type). Since the *VOID* lexer rule is earlier in the file than the *ID*, the lexer always matches the former when it encounters the text 'void'.

```
functionDefinition
    :   returnType
        functionName
        LPAREN (functionParameter (COMMA functionParameter)*)? RPAREN
        LCURLY statement* RCURLY
    ;
returnType: VOID | ID;
functionName: ID;

functionParameter
    :   parameterType parameterName
    ;
parameterType: ID;
parameterName: ID;
```

The m**ain block** can be structured as follows. A **statement** can be any statement from the specification. It is important to point out that although *returnStatement* is part of the *statement* rule (i.e. it is itself a *statement*), a return statement can only be included in the body of a function. We could write a rule that handles this, but this would complicate the structure of the language, so it is better to check this constraint during semantic analysis.

```
main: MAIN LCURLY statement* RCURLY;

statement
       :      variableDeclaration
       |      assignmentStatement
       |      whileStatement
       |      ifStatement
       |      functionCall EOS
       |      returnStatement
       ;
```

The following rules are responsible for **declaring and assigning variables**. When declaring, the initial value is optional, but when it is mandatory (in the case of *var* keywords) will be checked during the semantic analysis. The *expression* rule covers an arbitrary expression, as described in the specification, which will be explained later.

```
variableDeclaration
       :      (VAR | typeName) varName (ASSIGN expression)? EOS
       ;
typeName: ID;
varName: ID;

assignmentStatement
       :   varName ASSIGN expression EOS
       ;
```

The **while** loop and **if** statements are described by the following rules. The condition is separated in a different rule (*condition*) because it can be more easily checked during semantic analysis that it must always be of type *bool*. We could also omit using the *condition* rule and simply use *expression* instead. The *dangling else* problem is solved here by always demanding the use of delimiter characters (in this case, parentheses).

```
whileStatement
       :   WHILE LPAREN condition RPAREN LCURLY statement* RCURLY
       ;
condition: expression;

ifStatement
       :      IF LPAREN condition RPAREN LCURLY statement* RCURLY elseStatement?
       ;
elseStatement
       :   ELSE (ifStatement | LCURLY statement* RCURLY)
       ;
```

One could solve the dangling else problem by deterministically assigning the else branch to either the inner or outer if branch. Here it is worth looking into the ?? operator, although its use is not recommended in practice because of its inferior performance due to it being a non-greedy operation.

A **function call** (*functionCall*) can be a stand-alone statement or, as we will see later, part of an expression. The parameters passed in the call can be any *expression*. A r**eturn** statement may also return an arbitrary expression, type checking is a matter of semantic analysis.

```
functionCall
    :   functionTarget LPAREN parameterList? RPAREN
    ;
functionTarget: ID;

parameterList
    :   (expression (COMMA expression)*)
    ;

returnStatement
    :   RETURN expression EOS
    ;
```

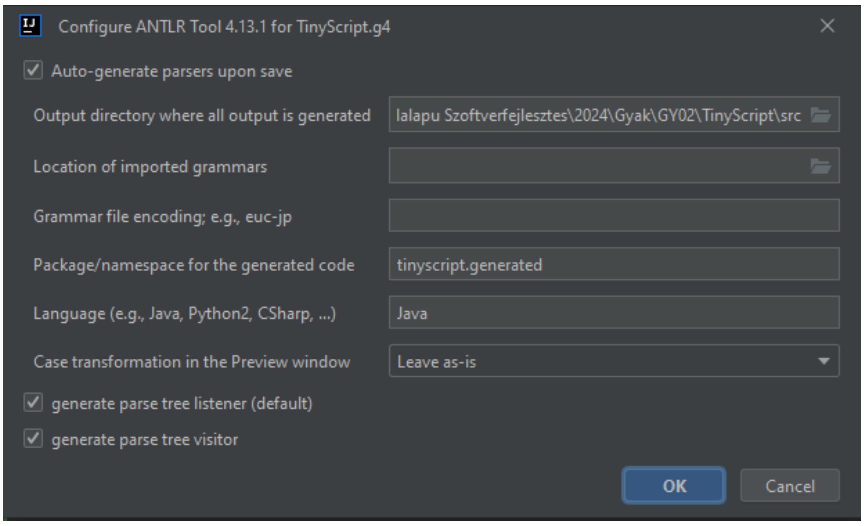
The only thing left to do is to define **expressions**. The newer versions of ANTLR support *self left recursion*, so it is easy to define the different operators and their precedence. Using labels (e.g. *#primaryExpression*), we can better separate different expression types, and in the generated code, a separate class will be generated after each label. The precedence is defined from top to bottom (from strongest to weakest), it is worth testing the *expression* rule with different examples and see what kind of tree is generated. Walking the tree also gives the precedence.

```
expression
    :   primary                                             #primaryExpression
        | expression mulDivOp expression                    #mulDivExpression
        | expression addSubOp expression                    #addSubExpression
        | expression logicalOp expression                   #logicalExpression
    ;

addSubOp
    : PLUS | MINUS
    ;
mulDivOp
    : MUL | DIV
    ;
logicalOp
    : GTE | LTE | GT | LT | EQ | NEQ
    ;

primary
    : parenthesizedExpression
    | functionCall
    | literalExpression
    ;

parenthesizedExpression
    :   LPAREN expression RPAREN
    ;

literalExpression
    : nullLiteral
    | intLiteral
    | stringLiteral
    | boolLiteral
    | varRef
    ;
varRef: ID;
nullLiteral : NULL;
intLiteral : MINUS? INT;
stringLiteral: STRING;
boolLiteral: TRUE | FALSE;
```

IV. Generating the lexer and parser, testing the grammar

From the grammar, ANTLR can generate a *lexer* and *parser* that automatically perform the lexical and syntactic analysis. In an IntelliJ environment, right-clicking anywhere in the grammar file allows you to configure the generation using the ***Configure ANTLR...*** menu item, and the ***Generate ANTLR Recognizer*** menu item executes the generation. By default, the generated files are placed in the ***gen*** folder, this is worth re-configuring.

Állítsuk be a konfigurációt az alábbihoz hasonlóan (a package nevére különösen ügyelve), majd generáljuk le a fájlokat a ***Generate ANTLR Recognizer*** menüponttal.



Configure ANTLR Tool 4.13.1 for TinyScript.g4

☑ Auto-generate parsers upon save

Output directory where all output is generated: lalapu Szoftverfejlesztes\2024\Gyak\GY02\TinyScript\src

Location of imported grammars:

Grammar file encoding; e.g., euc-jp:

Package/namespace for the generated code: tinyscript.generated

Language (e.g., Java, Python2, CSharp, ...): Java

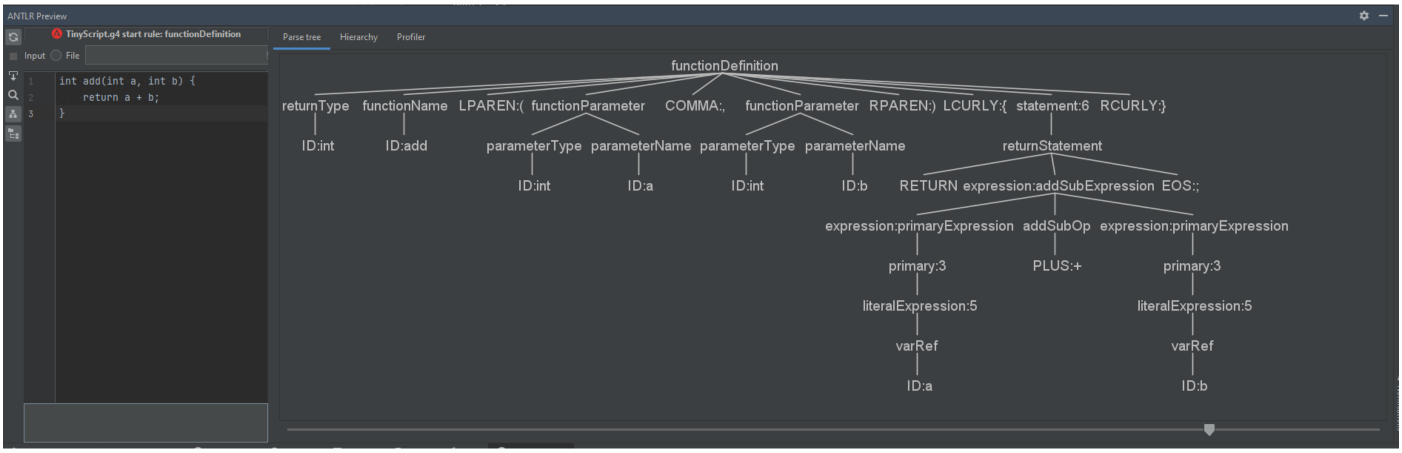
Case transformation in the Preview window: Leave as-is

☑ generate parse tree listener (default)

☑ generate parse tree visitor

We will not detail them here, but it is worth briefly reviewing what files ANTLR generates (lexer, parser, visitor, listener). In short, each of the parser rules defined in the grammar generates a separate class (e.g. *ProgramContext* from the initial rule) that can be used later to traverse and process the tree.

If you want to test ("debug") the grammar, right-click on a rule (e.g. the *program* start rule) and select the ***Test Rule program*** option. You can then test the grammar in the ***ANTLR Preview*** window below. For example, we can enter the example code in the **input.tys** file and see what parse tree ANTLR generates from it. The figure below instead shows the definition of a function (*functionDefinition* rule):



TinyScript.g4 start rule: functionDefinition

○ File

```
int add(int a, int b) {
    return a + b;
}
```