4

Installing and testing ANTLR 4

This chapter covers

- How to install ANTLR 4 and its testing tool
- How to use ANTLR 4 to generate translator components
- How to test ANTLR 4's generated components

ANTLR 4 is a powerful compiler-development tool that you'll use to automatically generate key components of the translator software architecture that you saw in the previous chapter. It will save you from hundreds of lines of tedious hand-coding to implement a parser, a scanner, and a parse tree. You can instead focus on creatively designing your source language. Using ANTLR 4's powerful code generation, each time you make changes or additions to your language's grammar, you rerun ANTLR 4 on your modified grammar, and you'll quickly have a new parse, scanner, and parse tree routines that can process source programs containing your new language features.

ANTLR 4 reference

The reference book for ANTLR 4 is *The Definitive ANTLR 4 Reference* by Terence Parr, Pragmatic Bookshelf, 2013. ISBN-13: 978-1934356999. Parr is the developer of ANTLR through all its versions.

This book only covers the features of ANTLR 4 required to develop the book's translators, and you'll be introduced to features when you need them. Refer to Parr's book if you desire to learn about and use other ANTLR 4 capabilities.

There is online information about ANTLR 4 at https://www.antlr.org.

4.1 Installing ANTLR 4

ANTLR 4 is a Java application, and therefore, it will run on any platform that supports Java, including Microsoft Windows, Apple MacOS, and Linux. It is stored as a Java archive (jar) file, to install it, start by downloading the latest version of the jar file from the ANTLR 4 website https://www.antlr.org. As I'm writing this, the latest jar file is

```
antlr-4.13.2-complete.jar
```

You should use the latest versions whenever they become available.

After downloading the jar file, you need to set your platform's `CLASSPATH` environment variable to refer to it. On each platform, the goal is to create command script `antlr4` to run ANTLR 4 and command script `grun` to run its testing tool. You will see examples of using these scripts in sections 4.2 and 4.3 to generate and test the translator components.

4.1.1 Installing on Microsoft Windows

Figure 4.1 shows an example of how to set the `CLASSPATH` in Windows to include the file path of the ANTLR 4 jar file.

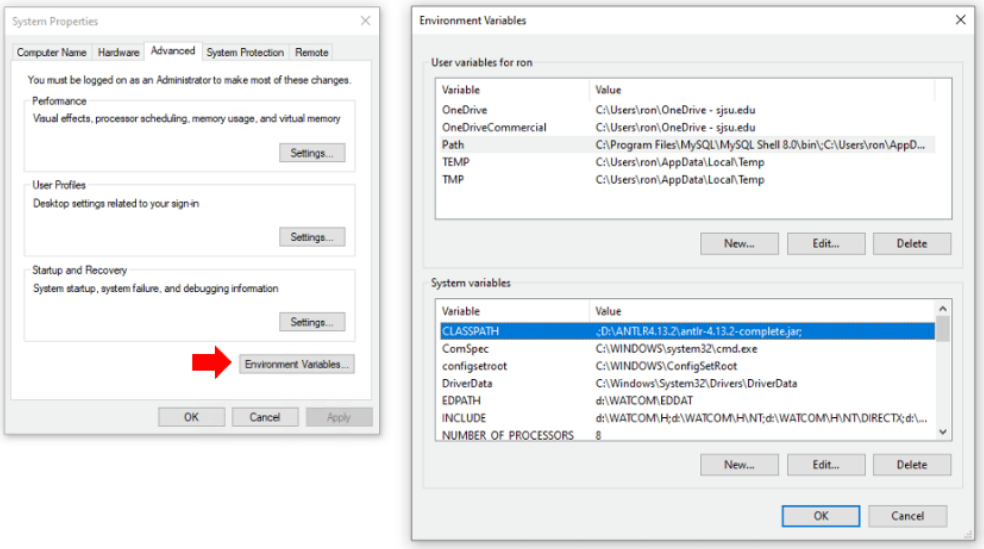

Figure 4.1: To set the `CLASSPATH` environment variable in Microsoft Windows, first run `sysdm.cpl` in the system search box or in a Command Prompt window to open the System Properties dialog box (left image). Select the Advanced tab and click on the Environment Variables button (indicated by the arrow). When the Environment Variables dialog box opens (right image), select the system variable `CLASSPATH` and click the Edit button. Add the file path of where you put the ANTLR 4 jar file to the `CLASSPATH`.

You can create two command scripts. The script named `antlr4.bat` should contain the following command to run ANTLR 4:

```
java org.antlr.v4.Tool -visitor -no-listener %1 %2 %3 %4 %5
```

The significance of the `-visitor` and `-no-listener` options is explained below.

The script named `grun.bat` should contain the following command to run the testing tool:

```
java org.antlr.v4.gui.TestRig %1 %2 %3 %4
```

Using the same procedure where you set the `CLASSPATH`, you should also set the `Path` environment variable to include the directory where you stored the `.bat` scripts. Windows looks at `Path` to find executables that you want to run. You'll be able to run both `antlr4` and `grun` when you're currently in any other directory.

Note that these scripts will work only if the ANTLR 4 jar file is already on the `CLASSPATH`.

4.1.2 Installing on MacOS and Linux

The procedure to install ANTLR 4 is the same on MacOS and Linux.

You can set the `CLASSPATH` environment variable by executing the following in a terminal window:

```
export CLASSPATH=".:~/antlr4/antlr-4.13.2-complete.jar:$CLASSPATH"
```

(assuming directory `antlr4` is in your home directory). By putting that command in your `.bashrc` startup file, you can have that `CLASSPATH` automatically set each time you log in or open a new terminal window. Include the `".:"` at the beginning of the string only if `"."` isn't already in the `CLASSPATH` — the dot must be part of `CLASSPATH` to allow you to run Java programs in any directory that you're currently in.

To simplify the commands to run ANTLR 4 and its testing tool, you'll want to create some command scripts. You can create a shell script named `antlr4` that contains two lines similar to

```
#!/bin/sh
java org.antlr.v4.Tool -visitor -no-listener
```

The significance of the `-visitor` and `-no-listener` options is explained below.

Instead of the shell script, you can instead create a command alias similar to

```
alias antlr4='java org.antlr.v4.Tool -visitor -no-listener'
```

A shell script named `grun` that contains two lines similar to the following will run the ANTLR 4 testing tool:

```
#!/bin/sh
java org.antlr.v4.gui.TestRig
```

Or, if you prefer, an alias:

```
alias grun='java org.antlr.v4.gui.TestRig'
```

Also put the two aliases in your `.bashrc` startup file. Then your aliases will always be available.

If you choose to use the shell scripts, using the same procedure where you set the `CLASSPATH`, you should set the `PATH` environment variable to include the directory where you stored the scripts. The operating system looks at `Path` to find executables that you want to run. You'll be able to run both `antlr4` and `grun` when you're currently in any other directory. You don't need to change `PATH` if you use the aliases.

Note that these scripts and aliases will work only if the ANTLR 4 jar file is already on the `CLASSPATH`.

4.2 Generating translator components

Figure 4.2 is based on the software architecture diagram you saw in the previous chapter. It shows that given a grammar file, ANTLR 4 will generate key components in the front end and in the intermediate tier.

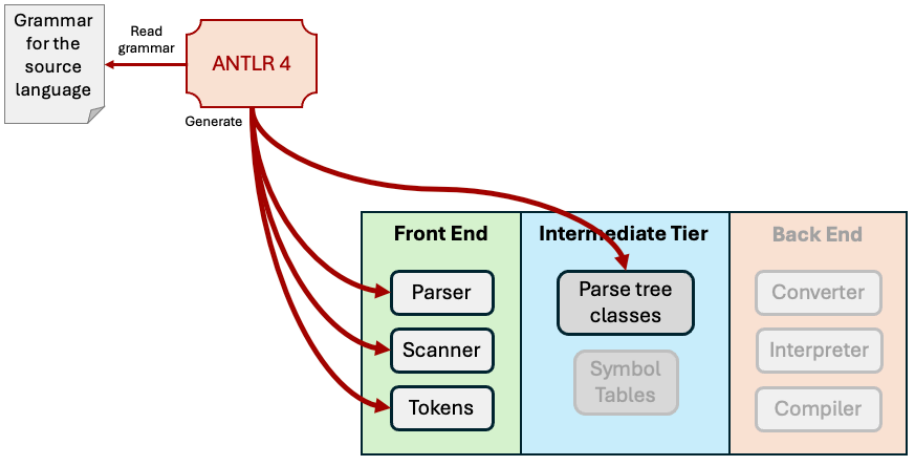

Figure 4.2: Components of the translator software architecture automatically generated by ANTLR 4.

The grammar file for the source language is a `.g4` grammar file such as `Simple.g4`, reproduced below.

Listing 4.1: ANTLR 4 grammar file `Simple.g4`

```
grammar Simple ;

program : statement ( ';' statement )* ;
```

```
statement : assignment
          | write
          | /* empty */
          ;

assignment : variable ':=' expression ;
expression : operand ( operator operand )* ;
operand    : variable | number ;
operator   : '+' | '-' ;

write : WRITELN '(' variable ')' ;

variable : LETTER ;
number   : DIGIT ;

WRITELN : 'writeln' ;
LETTER  : [a-z] ;
DIGIT   : [0-9] ;

WHITESPACE : [ \n\r\t]+ -> skip ;
```

In a test directory containing only the grammar file `Simple.g4`, run ANTLR 4 on the command line in a terminal window:

```
antlr4 Simple.g4
```

If ANTLR 4 did not flag any errors in the grammar file, it will generate Java code for the parser and the scanner, along with `.tokens` and `.interp` support files used by those components. The test directory will then contain the files

```
Simple.g4
Simple.interp
Simple.tokens
SimpleLexer.interp
SimpleLexer.java
SimpleLexer.tokens
SimpleParser.java
SimpleVisitor.java
SimpleBaseVisitor.java
```

Scanner vs. lexer

ANTLR 4 calls the generated scanner a *lexer*, short for *lexical analyzer*. This name comes from the more formal name *lexeme* for token. However, ANTLR 4 still uses the name *token*.

We'll focus primarily on the generated Java files `SimpleLexer.java`, `SimpleParser.java`, `SimpleVisitor.java`, and `SimpleBaseVisitor.java`. The remaining files are support files used internally by ANTLR 4.

Java classes `SimpleParser` and `SimpleLexer` are the generated parser and scanner, respectively, for our Simple language, based on the grammar file `Simple.g4`. The Java interface `SimpleVisitor` and its base implementation class `SimpleBaseVisitor` are for accessing and "walking" (accessing the tree nodes in a specific order) the parse tree that the parser will later build from a Simple source program.

ANTLR 4 can generate two tree-walking interfaces. By default, it generates the *listener* interface which you'll use later when you develop a debugger command language. For now, you'll specify the *visitor* interface which you'll mostly use. Therefore, the `antlr4` scripts include the options

```
-visitor -no-listener
```

You'll learn how to use the generated Java interfaces and classes as you develop the translators. Even though the generated parser and scanner are written in Java, you'll rarely need to look at their code, and only if they aren't behaving as you think they should. Whenever the parser or scanner behave unexpectedly, the grammar file is usually the first place to look for the problem — not the generated Java code.

Since you know that the parser will build a parse tree from each Simple source program, you can take a quick peek at `SimpleParser.java` to see that ANTLR 4 generated class `SimpleParser` with the nested public static "context" classes shown in listing 4.2. Each context class is a subclass of a superclass named `ParserRuleContext` which is defined in the ANTLR 4 jar file.

Listing 4.2: Static subclasses of class SimpleParser in `SimpleParser.java`

```
public static class ProgramContext extends ParserRuleContext
public static class StatementContext extends ParserRuleContext
public static class AssignmentContext extends ParserRuleContext
public static class ExpressionContext extends ParserRuleContext
public static class OperandContext extends ParserRuleContext
public static class OperatorContext extends ParserRuleContext
public static class WriteContext extends ParserRuleContext
public static class VariableContext extends ParserRuleContext
public static class NumberContext extends ParserRuleContext
```

The context classes correspond to the production rules of the grammar file `Simple.g4`, and objects instantiated from these subclasses will become the nodes of the parse tree. Because you ran the `antlr4` script with the `–visitor` option, ANTLR 4 not only generated context classes to build the parse tree, but also the visitor interface and implementation classes to access and walk the parse tree. You'll learn to use the visitor code in the next chapter.

Compile all the generated Java files in the test directory with the command in the terminal window:

```
javac *.java
```

4.3 Using the ANTLR 4 testing tool

Once you've used ANTLR 4 to generate its files from the `Simple.g4` grammar file and you've compiled all the generated Java files, you can use ANTLR 4's testing tool to test and debug your grammar. You want to ensure that the generated parser and scanner will work properly on a Simple source program.

4.3.1 Test the generated scanner

Use the following very short Simple source program `example2.simp` to test the generated scanner:

Listing 4.3: `example2.simp`

```
n := i + j - 7;
k := n;
writeln(k);
```

You want to ensure that the generated Simple scanner properly recognizes all the tokens in the source program according to the token definitions in `Simple.g4`. Enter the following command in the terminal window:

```
grun Simple program -tokens example2.simp
```

The first argument `Simple` is the grammar name and the second argument `program` is the starting rule of the grammar. The testing tool invokes the generated scanner. Option `-tokens` will generate the list of recognized tokens, and the last argument `example2.simp` is the source file. Listing 4.4 shows the output.

Listing 4.4: Output from running `grun` with option `-tokens`

```
[@0,0:0='n',<LETTER>,1:0]
[@1,2:3=':=',<':='>,1:2]
[@2,5:5='i',<LETTER>,1:5]
[@3,7:7='+',<'+'>,1:7]
[@4,9:9='j',<LETTER>,1:9]
[@5,11:11='-',<'-'>,1:11]
[@6,13:13='7',<DIGIT>,1:13]
[@7,14:14=';',<';'>,1:14]
[@8,16:16='k',<LETTER>,2:0]
[@9,18:19=':=',<':='>,2:2]
[@10,21:21='n',<LETTER>,2:5]
[@11,22:22=';',<';'>,2:6]
[@12,24:30='writeln',<'writeln'>,3:0]
[@13,31:31='(',<'('>,3:7]
[@14,32:32='k',<LETTER>,3:8]
[@15,33:33=')',<')'>,3:9]
[@16,34:34=';',<';'>,3:10]
[@17,36:35='<EOF>',<EOF>,4:0]
```

The output has one line per token in the order that the scanner recognized them in source file. There are 18 tokens, numbered `@0` through `@17`. Each line shows a token's starting and ending

character position (starting with position 0) within the file, the token text, the token's name, its source line number (starting with 1), and its starting character position in the line (starting with 0). When counting positions, each tab and line feed character counts as one character.

For example, the eleventh output line

```
[@10,21:21='n',<LETTER>,2:5]
```

states that the eleventh token (`@10`) is in character positions 21 through 21 (`21:21`) of the source file, its text is the single character `n`, the token is a `LETTER`, and it's in line 2 starting at character position 5 of the line `(2:5)`. As specified in `Simple.g4`, each blank, tab, and line feed character is skipped (ignored), but each takes up one character position.

From the printed list of tokens, you can see that the generated Simple scanner did its job properly with the source file `example2.simp`.

4.3.2 Test the generated parser

A parser does its work by repeatedly requesting tokens one at a time from the scanner. You want to verify that the generated Simple parser properly parses a source file and builds a correct parse tree from the tokens. Again using the source file `example2.simp`, enter the following command in the terminal window:

```
grun Simple program -gui example2.simp
```

The testing tool invokes the generated parser, which in turn calls upon the generated scanner. The option `-gui` says to display the parse tree that the parser built from `example2.simp`.

Figure 4.3 shows the parse tree displayed in Microsoft Windows.

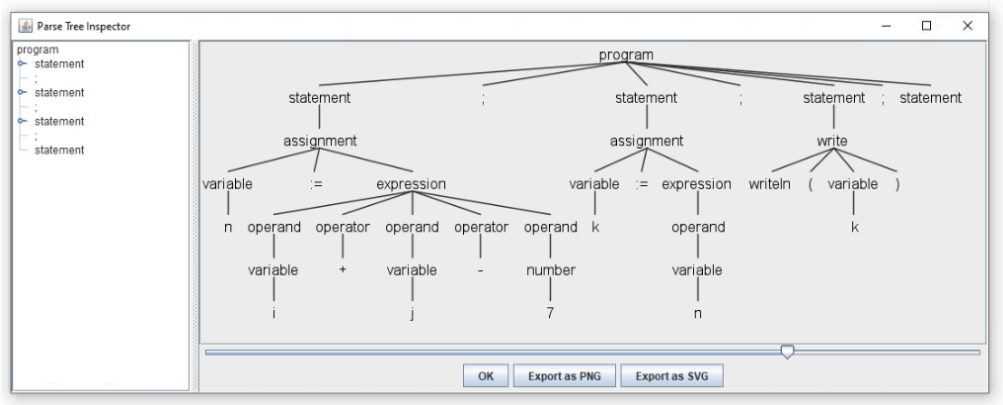

Figure 4.3: A parse tree displayed in Microsoft Windows.

To demonstrate that ANTLR 4 and its testing tool can run on various platforms, Figure 4.4 shows the tree in a separate window displayed in Apple's MacOS.

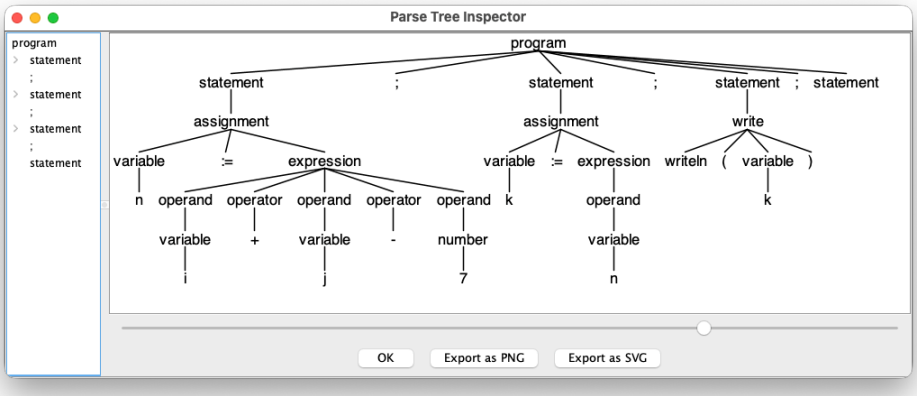

Figure 4.4: A parse tree displayed in a window in Apple's MacOS.

Figure 4.5 shows the tree in a separate window displayed in the Ubuntu distribution of Linux.

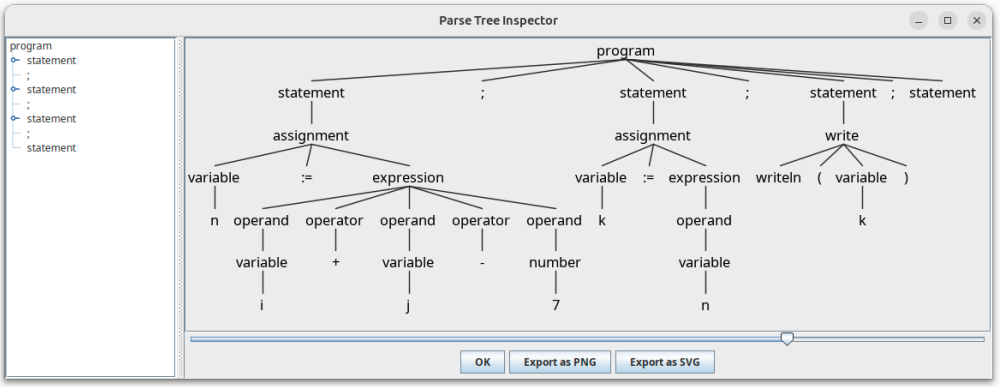

Figure 4.5: A parse tree displayed in a window in the Ubuntu distribution of Linux.

Examine the displayed parse tree carefully. Each non-leaf node is an object instantiated from one of the context classes in listing 4.2: the node labeled "program" is instantiated from class `ProgramContext`, the node labeled "statement" is instantiated from class `StatementContext`, the node labeled "assignment" is instantiated from class

`AssignmentContext`, etc. Each leaf node is labeled by one of the tokens, such as "n" and ":=". Each and every token in source file `example2.simp` is represented by a leaf node in the parse tree.

If you prefer to create your own parse tree display, you can generate the tree as text in Lisp notation:

```
grun Simple program -tree example2.simp
```

The output is the single line

```
(program (statement (assignment (variable n) := (expression
[CA] (operand (variable i)) (operator +) (operand (variable j))
[CA] (operator -) (operand (number 7))))) ; (statement (assignment
[CA] (variable k) := (expression (operand (variable n))))) ;
[CA] (statement (write writeln ( (variable k) ))) ; statement)
```

You can then write your own tree display program to process the Lisp text.

4.4 The translation workflow

Running the `antlr4` and `grun` scripts exemplifies the workflow of using ANTLR 4 to generate parser, scanner, token, and parse tree translator components and then running those components on a source program to build a parse tree. This is shown in figure 4.6.

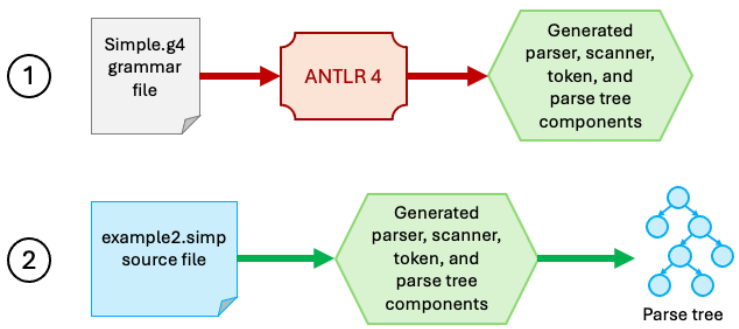

Figure 4.6: The ANTLR 4 workflow. Step 1 is to generate the parser, scanner, token, and parse tree translator components. Step 2 is to use the generated components to translate a source program and build a parse tree.

In step 1, run `antlr4` to make ANTLR 4 generate the components. Repeat this step each time you make changes to the grammar file. In step 2, run `grun` to use the generated components to build a parse tree from a source file. You repeat this step for each source file, or for each time you make changes to a source file. The `grun` testing tool is an example of step 2 when it builds a parse tree. It takes the place of the `main()` function that you will write when you develop a translator.

Now that you know how to run ANTLR 4 to generate translator components from a grammar file, and how to run ANTLR 4's testing tool to verify that the generated parser and scanner are both working properly to build a parse tree, you're ready to walk the parse tree using the visitor interface and perform some semantic operations, which is the topic of the next chapter.

4.5 Summary

- Download and install the ANTLR 4 jar file and set the system `CLASSPATH` to include its file path.
- Create command scripts or aliases `antlr4` and `grun` to run ANTLR 4 and its testing tool, respectively.
- Run the `antlr4` command with the grammar file `Simple.g4` to generate the parser and scanner components.
- Test the generated scanner by running the `grun` command with the `-tokens` option on the source file `example2.simp` to verify that the scanner recognized all the tokens in the source file.
- Test the generated parser by running the `grun` command with the `-gui` option on the source file `example2.simp` to verify that the parser built a proper parse tree.

4.6 Exercises

4.1. Explain how ANTLR 4 is itself a translator. What is the source language? During a translation, what is the source file and what is its output? What are its syntactic and semantic operations? A compiler development tool like ANTLR 4 is often called a *compiler-compiler*.

4.2. In the translation workflow, explain why the two steps should be separate. Which step do you need to rerun if you change the grammar file? If you change the source file?

4.3. When you use ANTLR 4 to develop a translator, the translator components fall into two categories: components generated by ANTLR 4 and components you must write yourself. What is the primary distinguishing feature (other than the author of the components) of the two categories?

4.4. Consider the development of translators for multiple source languages. What components of the translators can be directly shared among the languages? What components need to be modified to allow sharing?

4.5. Without running ANTLR, predict the scanner output for a Simple source program containing

```
a := b - 5;
writeln(a);
```

Then run the `antlr4` command and the `grun` command with the `-tokens` option and compare your prediction with the actual output.

4.6. Run the `grun` command with the `-tokens` option and then with the `-gui` option on a Simple source file containing the erroneous statement

```
k := i +- j
```

Explain the output from the two commands.

4.7. Modify the grammar file `simple.g4` to allow a number to have multiple digits. Allow a variable name to have multiple characters subject to the rule that the first character must be a letter and subsequent characters can be letters or digits.

4.8. Modify the grammar file `simple.g4` to allow Java-style comments using the tokens `//` and `/*` and `*/`. Comments should be treated like whitespace.

4.9. Modify the grammar file `simple.g4` to include the `*` and `/` operators in expressions and include expressions containing those operators in source file `example.simp`. Rerun the `antlr4` and `grun` commands. What parse tree is generated? Can you modify the grammar file such that the parse tree incorporates the operator precedence rule of performing multiplications and divisions before additions and subtractions?

4.10. Deliberately (if you haven't already done so accidentally) make a syntax error in a grammar file, such as leaving off a semicolon or referring to an undefined production rule name. What does ANTLR 4 do when you give it the erroneous grammar file?