Mega Tutorial

Everything you need to learn about ANTLR

Learn more at https://tomassetti.me

About this Tutorial

This tutorial is simply the most complete tutorial you will find about ANTLR. It will teach everything you need to know, starting from the basics and continuing to the most advanced topics.

Examples are provided in Java, C#, Python and JavaScript.

I hope this document will help you get jump-started on using ANTLR.

Remember, I would love to hear your feedback, advice on what could be improved and questions which remain unanswered: feel free to write to federico@tomassetti.me

Parsers are powerful tools and using ANTLR you could write all sorts of parsers, usable from many different languages.

In this complete tutorial we are going to:

- **explain the basics**: what a parser is, what it can be used for
- see **how to setup ANTLR** to be used from JavaScript, Python, Java, and C#
- discuss **how to test** your parser
- present the most **advanced and useful features** present in ANTLR: you will learn all you need to parse all possible languages
- show **tons of examples**

Maybe you have read some tutorial that was too complicated or so incomplete that seemed to assume that you already knew how to use a parser. This is not that kind of tutorial. We just expect you to know how to code and how to use a text editor or an IDE. That's it.

At the end of this tutorial:

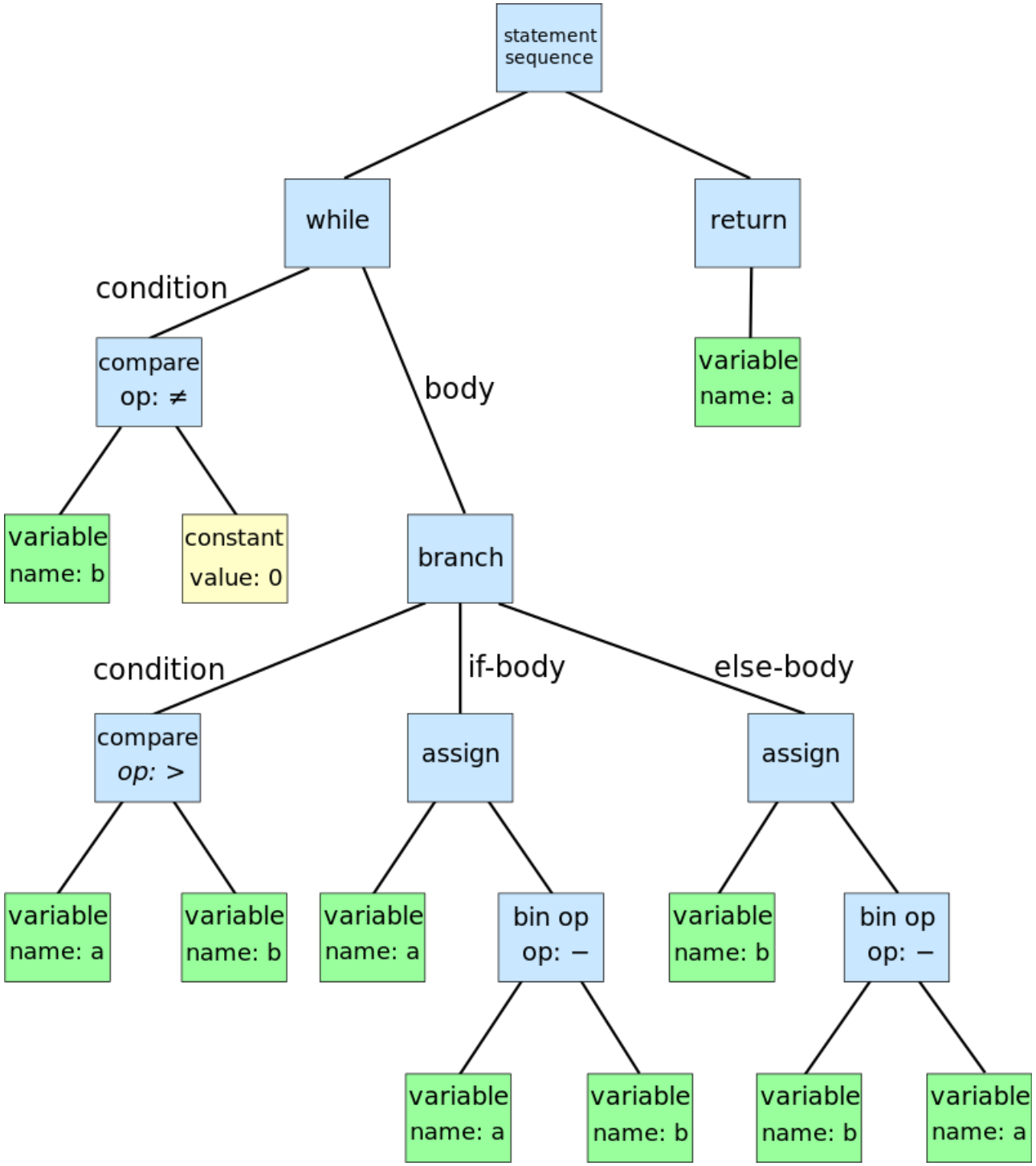
- you will be able to write a parser to recognize different formats and languages
- you will be able to create all the rules you need to build a lexer and a parser
- you will know how to deal with the common problems you will encounter
- you will understand errors and you will know how to avoid them by testing your grammar.

In other words, we will start from the very beginning and when we reach the end you will have learned all you could possibly need to learn about ANTLR to be productive.

ANTLR Mega Tutorial Giant List of Content

What is ANTLR?

ANTLR is a parser generator, a tool that helps you to create parsers. **A parser takes a piece of text and transforms it into an organized structure**, a *parse tree*, also known as an *Abstract Syntax Tree (AST)*. You can think of the AST as a story describing the content of the code, or also as its logical representation, created by putting together the various pieces.



Graphical representation of an AST for the Euclidean algorithm

What you need to do to get a parse tree:

1. define a lexer and parser grammar
2. invoke ANTLR: it will generate a lexer and a parser in your target language (e.g., Java, Python, C#, JavaScript)
3. use the generated lexer and parser: you invoke them by passing the code to recognize and they return to you a parse tree

So you need to start by defining a lexer and parser grammar for the thing that you are analyzing. Usually the "thing" is a language, but it could also be a data format, a diagram, or any kind of structure that is represented by text.

Notice that technically what you get from ANTLR is a *parse tree* rather than an *AST*. The difference is that a parse tree is exactly what comes out of the parser, while the AST is a more refined version of the parse tree. You create the AST by manipulating the parse tree, in order to get something that is easier to use by subsequent parts of your program. These changes are sometimes necessary because a parse tree might be organized in a way that make parsing easier or better performing. However, you might prefer something more user friendly in the rest of the program.

The distinction in moot in our examples shown here, given they are quite simple, so we use the terms interchangeably here. However, it is something to keep in mind while reading other documents.

Aren't regular expressions enough?

If you are the typical programmer, you may ask yourself *why can't I use a regular expression*? A regular expression is quite useful, such as when you want to find a number in a string of text, but it also has many limitations.

The most obvious is the lack of recursion: you cannot find a (regular) expression inside another one, unless you code it by hand for each level, something that quickly becomes unmaintainable. But the larger problem is that it is not really scalable: if you are going to put together even just a few regular expressions, you are going to create a fragile mess that would be hard to maintain.

> It is not that easy to use regular expressions

Have you ever tried parsing HTML with a regular expression? It's a terrible idea, for one thing you risk summoning Cthulhu, but more importantly **it does not really work**. You do not believe me? Let's see, you want to find the elements of a table, so you try a regular expression like this one: `<table>(.*?)</table>`. Brilliant! You did it! Except somebody adds attributes to their table, such as style or id. It does not matter, you write `<table.*?>(.*?)</table>`. Still, you actually cared about the data inside the table. So you then need to parse `tr` and `td`, but they are full of tags.

Therefore you need to eliminate that, too. And somebody dares even to use comments like `<!— my comment —>`. Comments can be used everywhere, and they are not easy to treat with your regular expression. Is it?

So you forbid the internet to use comments in HTML: *problem solved*.

Or alternatively you use ANTLR, whatever seems simpler to you.

ANTLR vs writing your own parser by hand

Okay, you are convinced, you need a parser, but why use a parser generator like ANTLR instead of building your own?

> The main advantage of ANTLR is productivity

If you actually have to work with a parser all the time, because your language, or format, is evolving, you need to be able to keep pace. This is something you cannot do if you have to deal with the details of implementing a parser. Since you are not parsing for parsing's sake, you must have the chance to concentrate on accomplishing your goals. And ANTLR makes it much easier to do that, rapidly and cleanly.

A second thing, once you defined your grammars you can ask ANTLR to generate multiple parsers in different languages. For example, you can get a parser in C# and one in JavaScript, to parse the same language in a desktop application and in a web application.

Some people argue that by writing a parser by hand you can make it faster and you can produce better error messages. There is some truth in this, but in my experience parsers generated by ANTLR are always fast enough. You can tweak them and improve both performance and error handling by working on your grammar, if you really need to. And you can do that once you are happy with your grammar.

Table of Contents

What is ANTLR? **3**
Aren't regular expressions enough? **5**
ANTLR vs writing your own parser by hand **6**
Table of Contents **7**
Setup **9**
1. Setup ANTLR **9**
Instructions 9
2. JavaScript Setup **11**
3. Python Setup **12**
4. Java Setup **12**
5. C# Setup **15**
Alternatives If You Are Not Using Visual Studio Code **15**
Picking the Right Runtime **15**
Beginner **16**
6. Lexers and Parsers **16**
7. Creating a Grammar **18**
Top-down approach 18
Bottom-up approach 18
8. Designing a Data Format **19**
9. Lexer Rules **19**
10. Parser Rules **21**
11. Mistakes and Adjustments **22**
Mid-Level **25**
12. Setting Up the Chat Project with JavaScript **25**
13. Antlr.js **30**
14. HtmlChatListener.js **31**
15. Working with a Listener **34**
16. Solving Ambiguities with Semantic Predicates **36**
17. Continuing the Chat in Python **38**
18. The Python Way of Working with a Listener **39**
19. Testing with Python **42**
20. Parsing Markup **45**
21. Lexical Modes **45**
22. Parser Grammars **46**
Advanced **47**
23. The Markup Project in Java **48**
24. The Main App.java **48**
25. Transforming Code with ANTLR **49**
26. Joy and Pain of Transforming Code **50**
27. Advanced Testing **53**
28. Dealing with Expressions **56**
29. Parsing Spreadsheets **58**
30. The Spreadsheet Project in C# **60**
31. Excel is Doomed **61**
32. Testing Everything **63**
Final Remarks **66**
33. Tips and Tricks **66**
Catchall Rule 67
Channels 67

RULE ELEMENT LABELS 67
PROBLEMATIC TOKENS 67
34. CONCLUSIONS **68**

Two small notes:

- in the companion repository of this tutorial you are going to find all the code with testing, even where we don't see it in the article
- the examples will be in different languages, but the knowledge would be generally applicable to any language

Setup

In this section we prepare our development environment to work with ANTLR: the parser generator tool, the supporting tools and the runtimes for each language.

1. Setup ANTLR

ANTLR is actually made up of two main parts: the tool, used to generate the lexer and parser, and the runtime, needed to run them.

The tool will be needed just by you, the language engineer, while the runtime will be included in the final software created by you.

The tool is always the same no matter which language you are targeting: it is a Java program that you need on your development machine. It is used to generate the lexer and parser. While the runtime is different for every language and must be available both to the developer and to the user. It is needed to run the program.

The only requirement for the tool is that you have installed at least **Java 1.7**. To install the Java program, you need to download the latest version from the official site, which at the moment is:

```
1  https://www.antlr.org/download/antlr-4.9.1-complete.jar
```

Instructions

1. copy the downloaded tool where you usually put third-party java libraries (ex. /usr/local/lib or C:\Program Files\Java\lib)
2. add the tool to your CLASSPATH. Add it to your startup script (ex. .bash_profile)
3. (optional) add also aliases to your startup script to simplify the usage of ANTLR

Executing the instructions on Linux/Mac OS

```
1  // 1.
2  sudo cp antlr-4.9.1-complete.jar /usr/local/lib/
3  // 2. and 3.
4  // add this to your .bash_profile
5  export CLASSPATH=".:/usr/local/lib/antlr-4.9-1-complete.jar:$CLASSPATH"
6  // simplify the use of the tool to generate lexer and parser
```

```
alias antlr4='java -jar /usr/local/lib/antlr-4.9.1-complete.jar'
// simplify the use of the tool to test the generated code
alias grun='java org.antlr.v4.gui.TestRig'
```

Executing the instructions on Windows

```
// 1.
// Copy antlr-4.9.1-complete.jar in C:\Program Files\Java\libs (or wherever you prefer)
// 2. Create or append to the CLASSPATH variable the location of antlr:
// you can do to that by going to WIN + R and typing sysdm.cpl
// then selecting Advanced (tab) > Environment variables > System Variables
//CLASSPATH -> .;C:\Program Files\Java\libs\antlr-4.9.1-complete.jar;%CLASSPATH%
// 3. Add aliases
// create antlr4.bat
java org.antlr.v4.Tool %*
// create grun.bat
java org.antlr.v4.gui.TestRig %*
// put them in the system path or any of the directories included in %path%
```

Typical Workflow

When you use ANTLR you start by writing a *grammar*, a file with extension .g4 which contains the rules of the language that you are analyzing. You then use the antlr4 program to generate the files that your program will actually use, such as the lexer and the parser.

```
antlr4 <options> <grammar-file-g4>
```

There are a couple of important options you can specify when running antlr4.

First, you can specify the target language, to generate a parser in Python or JavaScript or any other target different from Java (which is the default one). The other ones are used to generate visitor and listener (do not worry if you do not know what these are, we are going to explain it later).

By default only the listener is generated, so to create the visitor you use the `-visitor` command line option, and `-no-listener` if you do not want to generate the listener. There are also the opposite options, `-no-visitor` and `-listener`, but they represent the default behavior.

```
1  antlr4 -visitor <Grammar-file>
```

You can optionally test your grammar using a little utility named `TestRig` (although, as we have seen, it is usually aliased to `grun`).

```
1  grun <grammar-name> <rule-to-test> <input-filename(s)>
```

The filename(s) are optional and you can instead analyze the input that you type on the console.

If you want to use the testing tool you need to generate a Java parser, even if your program is written in another language. This can be done just by selecting a different option with antlr4.

Grun is useful when testing manually the first draft of your grammar. As it becomes more stable you may want to relay on automated tests (we will see how to write them).

Grun also has a few useful options: `-tokens`, to show the tokens detected, `-gui` to generate an image of the AST.

2. Javascript Setup

You can put your grammars in the same folder as your Javascript files. The file containing the grammar must have the same name of the grammar, which must be declared at the top of the file.

In the following example the name is Chat and the file is Chat.g4.

We can create the corresponding Javascript parser simply by specifying the correct option with the ANTLR4 Java program.

```
1  antlr4 -Dlanguage=JavaScript Chat.g4
```

Notice that the option is case-sensitive, so pay attention to the uppercase 'S'. If you make a mistake you will receive a message like the following.

```
1  error(31): ANTLR cannot generate Javascript code as of version 4.9
```

ANTLR can be used both with node.js and in the browser. For the browser you need to use webpack or require.js. If you don't know how to use either of the two you can look at the official documentation for some help or read this tutorial on antlr in the web. We are going to use node.js, for which you can install the ANTLR runtime simply by using the following standard command.

```
1  npm install antlr4
```

3. Python Setup

When you have a grammar you put that in the same folder as your Python files. The file must have the same name of the grammar, which must be declared at the top of the file. In the following example the name is `Chat` and the file is `Chat.g4`.

We can create the corresponding Python parser simply by specifying the correct option with the ANTLR4 Java program. For Python, you also need to pay attention to the version of Python, 2 or 3.

```
1  antlr4 -Dlanguage=Python3 Chat.g4
```

The runtime is available from PyPi so you just can install it using pip.

```
1  pip install antlr4-python3-runtime
```

Again, you just have to remember to specify the proper python version.

4. Java Setup

To setup our Java project using ANTLR you can do things manually. Or you can be a civilized person and use Gradle or Maven. We are using Gradle here, but you can look at a typical setup using Maven in the ANTLR documentation.

Also, you can look in ANTLR plugins for your IDE.

This is how I typically setup my Gradle project.

I use a Gradle plugin to invoke ANTLR. Since I use IntellJ IDEA, I also use the IDEA plugin to generate the correct configuration for that IDE.

```
1  plugins {
2      id 'java'
3      id 'antlr'
4      id 'idea'
5  }
6
7  repositories {
8      mavenCentral()
9      jcenter()
10 }
```

```
11
12  dependencies {
13      antlr "org.antlr:antlr4:4.9.1"
14      compile "org.antlr:antlr4-runtime:4.9.1"
15      testImplementation(platform('org.junit:junit-bom:5.7.0'))
16      testImplementation('org.junit.jupiter:junit-jupiter')
17  }
18
19  generateGrammarSource {
20      maxHeapSize = "128m"
21      arguments += ['-package',
    'me.tomassetti.examples.MarkupParser', '-visitor', '-no-listener']
22  }
23  compileJava.dependsOn generateGrammarSource
24
25  sourceSets {
26      generated {
27          java.srcDir 'generated-src/antlr/main/'
28      }
29  }
30  compileJava.source sourceSets.generated.java, sourceSets.main.java
31
32  clean{
33      delete "generated-src"
34  }
35
36  idea {
37      module {
38          sourceDirs += file("generated-src/antlr/main/")
```

```
    }
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}
```

I put my grammars under *src/main/antlr/* and the gradle configuration make sure they are generated in the directory corresponding to their package. For example, if I want the parser to be in the package *me.tomassetti.mylanguage* it has to be generated into *generated-src/antlr/main/me/tomassetti/mylanguage*.

At this point I can simply run:

```
# Linux/Mac
./gradlew generateGrammarSource

# Windows
gradlew generateGrammarSource
```

And I get my lexer and parser generated from my grammar(s).

Then I can also run:

```
# Linux/Mac
./gradlew idea

# Windows
gradlew idea
```

And I have an IDEA Project ready to be opened.

There is a clear advantage in using Java for developing ANTLR grammars: there are plugins for several IDEs and it is the language that the main developer of the tool actually works on. So they are tools, like the `org.antlr.v4.gui.TestRig`, that can be easily integrated in your workflow and are useful if you want to easily visualize the AST of an input.

5. C# Setup

There is support for .NET Framework, Mono and .NET core. We are going to use Visual Studio Code to create our ANTLR project, because there is an excellent extension for it that automate the generation of the parser. It can also can generate lots of graph to help you debug your parser or to create documentation for the users of your parser. It is called ANTLR4 grammar syntax support and is created by one of the main contributors to ANTLR.

Alternatives If You Are Not Using Visual Studio Code

There is also a nice extension for Visual Studio 2015 and 2017 created by the same author of the C# target, called ANTLR Language Support. You can install it by going in Tools -> Extensions and Updates. This extension will automatically generate parser, lexer and visitor/listener when you build your project. Furthermore, the extension will allow you to create a new grammar file, using the well known menu to add a new item. Last, but not least, you can setup the options to generate listener/visitor right in the properties of each grammar file. This extension is now a bit outdated, but it is still useful if you want to use Visual Studio.

You can also use the usual Java tool to generate everything, even a parser for C#. You can do that just by indicating the right language. In this example the grammar is called `Spreadsheet`.

```
1  antlr4 -Dlanguage=CSharp Spredsheet.g4
```

Notice that the 'S' in CSharp is uppercase.

Picking the Right Runtime

There are a few things to remember if you are using the Visual Studio extension. Whether you are using an extension or the standard Java tool you also need an ANTLR4 runtime for your project, and you can install it with the good ol' **nuget**. But you have to remember that not all runtimes are created equal.

The issue is that in the past there was only a separate C#-optimized package of ANTLR published on nuget. Now instead the main authors of ANTLR published an official package on nuget. However, the author of old C#-optimized version keeps publishing its own package, that is incompatible with the standard ANTLR4 tool. This is not strictly a fork, since the same person continues to be a core contributor to the main ANTLR4 tool, but it is more of a parallel development. The creator of the C#-optimized package is also the author of the Visual Studio extension, Sam Harwell (with the nickname `sharwell`).

So, only if you are using the Visual Studio Extension you need to use the nuget package ANTLR4.runtime, authored by sharwell. If you are using the ANTLR4 tool, or the Visual Studio Code extension, to generate your C# lexer and parser then you need to use the ANTLR4.Runtime.Standard.

Notice that the C#-optimized release is a bit behind the official release. For that reason, if you are just starting now, I would suggest using the official standard runtime. Therefore it would be better to also either use Visual Studio Code as your IDE or not using the Visual Studio extension. This gets you the most updated version of ANTLR.

You also have to remember that both extensions comes with their own internal ANTLR tool, for ease of use. This way you do not need to have ANTLR installed in your system. However, this means that the ANTLR version included might be outdated. You can check the version mentioned in the generated parser file. This may lead to the issue that the parser is generated with an older version of the ANTLR, while the runtime you get with Nuget uses a new version of ANTLR.

Beginner

In this section we lay the foundation you need to use ANTLR: what lexer and parsers are, the syntax to define them in a grammar and the strategies you can use to create one. We also see the first examples to show how to use what you have learned. You can come back to this section if you do not remember how ANTLR works.

6. Lexers and Parsers

Before looking into parsers, we need first to look into lexers, also known as tokenizers. They are basically the first stepping stone toward a parser, and of course ANTLR allows you to build them too. **A lexer takes the individual characters and transforms them in tokens**, the atoms that the parser uses to create the logical structure.

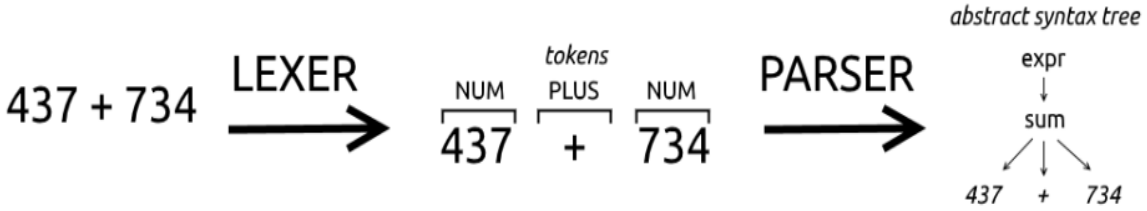
Imagine this process applied to a natural language such as English. You are reading the single characters, putting them together until they make a word, and then you combine the different words to form a sentence.

Let's look at the following example and imagine that we are trying to parse a mathematical operation.

```
1  437 + 734
```

The lexer scans the text and find '4', '3', '7' and then the space ' '. So it knows that the first characters actually represent a number. Then it finds a '+' symbol, so it knows that it represents an

operator, and lastly it finds another number.



How does it know that? Because we tell it to.

```
1   /*
2   * Parser Rules
3   */
4
5   operation  : NUMBER '+' NUMBER ;
6
7   /*
8   * Lexer Rules
9   */
10
11  NUMBER           : [0-9]+ ;
12
13  WHITESPACE : ' ' -> skip ;
```

This is not a complete grammar, but we can already see that lexer rules are all uppercase, while parser rules are all lowercase. Technically the rule about case applies only to the first character of their names, but usually they are all uppercase or lowercase for clarity.
Rules are typically written in this order: first the parser rules and then the lexer ones, although logically they are applied in the opposite order. It's also important to remember that **lexer rules are analyzed in the order that they appear**, and they can be ambiguous.

The typical example is the identifier: in many programming language it can be any string of letters, but certain combinations, such as "class" or "function" are forbidden because they indicate a *class* or a *function*. So the order of the rules solves the ambiguity by using the first match and that's why the tokens identifying keywords such as *class* or *function* are defined first, while the one for the identifier is put last.

> The basic syntax of a rule is easy: **there is a name, a colon, the definition of the rule and a terminating semicolon**

The definition of **NUMBER** contains a typical range of digits and a '+' symbol to indicate that one or more matches are allowed. These are all very typical indications with which I assume you are familiar with, if not, you can read more about the syntax of regular expressions.

The most interesting part is at the end, the lexer rule that defines the **WHITESPACE** token. It's interesting because it shows how to indicate to ANTLR to ignore something. Consider how ignoring whitespace simplify parser rules: if we couldn't say to ignore WHITESPACE we would have to include it between every single subrule of the parser, to let the user puts spaces where he wants. Like this:

```
1  operation: WHITESPACE* NUMBER WHITESPACE* '+' WHITESPACE* NUMBER;
```

And the same typically applies to comments: they can appear everywhere and we do not want to handle them specifically in every single piece of our grammar so we just ignore them (at least while parsing) .

7. Creating a Grammar

Now that we have seen the basic syntax of a rule, we can take a look at the two different approaches to define a grammar: top-down and bottom-up.

Top-down approach

This approach consist in starting from the general organization of a file written in your language.

What are the main section of a file? What is their order? What is contained in each section?

For example a Java file can be divided in three sections:

- package declaration
- imports
- type definitions

This approach works best when you already know the language or format that you are designing a grammar for. It is probably the strategy preferred by people with a good theoretical background or people who prefer to start with "the big plan".

When using this approach, you start by defining the rule representing the whole file. It will probably include other rules, to represent the main sections. You then define those rules and you move from the most general, abstract rules to the low-level, practical ones.

Bottom-up approach

The bottom-up approach consists in focusing in the small elements first: defining how the tokens are captured, how the basic expressions are defined and so on. Then we move to higher level constructs until we define the rule representing the whole file.

I personally prefer to start from the bottom, the basic items, that are analyzed with the lexer. And then you grow naturally from there to the structure, that is dealt with the parser. This approach permits to focus on a small piece of the grammar, build tests for that, ensure it works as expected and then move on to the next bit.

This approach mimics the way we learn. Furthermore, there is the advantage of starting with real code that is actually quite common among many languages. In fact, most languages have things like identifiers, comments, whitespace, etc. Obviously, you might have to tweak something, for example a comment in HTML is functionally the same as a comment in C#, but it has different delimiters.

The disadvantage of a bottom-up approach rests on the fact that the parser is the thing you actually care about. You were not asked to build a lexer, you were asked to build a parser, that could provide a specific functionality. So by starting on the last part, the lexer, you might end up doing some refactoring, if you do not already know how the rest of the program will work.

8. Designing a Data Format

Designing a grammar for a new language is difficult. You have to create a language simple and intuitive to the user, but also unambiguous to make the grammar manageable. It must be concise, clear, natural and it should not get in the way of the user.

So we are starting with something limited: a grammar for a simple chat program.

Let's start with a better description of our objective:

- there are not going to be paragraphs, and thus we can use newlines as separators between the messages
- we want to allow emoticons, mentions and links. We are not going to support HTML tags
- since our chat is going to be for annoying teenagers, we want to allow users an easy way to SHOUT and to format the color of the text.

Finally teenagers could shout, and all in pink. What a time to be alive.

9. Lexer Rules

We start with defining lexer rules for our chat language. Remember that lexer rules actually are at the end of the files.

```
1  /*
2  * Lexer Rules
3  */
4
5  fragment A      : ('A'|'a') ;
6  fragment S      : ('S'|'s') ;
```

```
7   fragment Y        : ('Y'|'y') ;
8   fragment H        : ('H'|'h') ;
9   fragment O        : ('O'|'o') ;
10  fragment U        : ('U'|'u') ;
11  fragment T        : ('T'|'t') ;
12
13  fragment LOWERCASE  : [a-z] ;
14  fragment UPPERCASE  : [A-Z] ;
15
16  SAYS              : S A Y S ;
17
18  SHOUTS            : S H O U T S;
19
20  WORD              : (LOWERCASE | UPPERCASE | '_')+ ;
21
22  WHITESPACE        : (' ' | '\t') ;
23
24  NEWLINE           : ('\r'? '\n' | '\r')+ ;
25
26  TEXT              : ~[\])]+ ;
```

In this example we use rules **fragments**: they are reusable building blocks for lexer rules. You define them and then you refer to them in lexer rules. If you define them but do not include them in lexer rules they have simply no effect.

We define a fragment for the letters we want to use in keywords. Why is that? Because we want to support case-insensitive keywords. Other than to avoid repetition of the case of characters, they are also used when dealing with floating numbers. To avoid repeating digits, before and after the dot/comma. Such as in the following example.

```
1   fragment DIGIT : [0-9] ;
2   NUMBER            : DIGIT+ ([.,] DIGIT+)? ;
```

The **TEXT** token shows how to capture everything, except for the characters that follow the tilde (‘~’). We are excluding the closing square bracket ‘]’, but since it is a character used to identify the end of a group of characters, we have to escape it by prefixing it with a backslash ‘\’.

The newlines rule is formulated that way because there are actually different ways in which operating systems indicate a newline, some include a carriage return ('\r') others a newline ('\n') character, or a combination of the two.

10. Parser Rules

We continue with parser rules, which are the rules with which our program will interact most directly.

```
1   /*
2   * Parser Rules
3   */
4
5   chat            : line+ EOF ;
6
7   line            : name command message NEWLINE;
8
9   message             : (emoticon | link | color | mention | WORD | WHITESPACE)+ ;
10
11  name            : WORD WHITESPACE;
12
13  command             : (SAYS | SHOUTS) ':' WHITESPACE ;
14
15  emoticon            : ':' '-'? ')'
16                  | ':' '-'? '('
17                  ;
18
19  link            : '[' TEXT ']' '(' TEXT ')' ;
20
21  color               : '/' WORD '/' message '/';
22
23  mention             : '@' WORD ;
```

The first interesting part is **message**, not so much for what it contains, but the structure it represents. We are saying that a message could be anything of the listed rules in any order. This is a simple way to solve the problem of dealing with whitespace without repeating it every time. Since we, as users, find whitespace irrelevant we see something like WORD WORD mention, but the parser actually sees WORD WHITESPACE WORD WHITESPACE mention WHITESPACE.

Another way of dealing with whitespace, when you can't get rid of it, is more advanced: lexical modes. Basically it allows you to specify two lexer parts: one for the structured part, the other for simple text. This is useful for parsing things like XML or HTML. We are going to show it later.

The **command** rule is obvious, you have just to notice that you cannot have a space between the two options for command and the colon, but you need one **WHITESPACE** after. The **emoticon** rule shows another notation to indicate multiple choices, you can use the pipe character '|' without the parenthesis. We support only two emoticons, happy and sad, with or without the middle line.

Something that could be considered a bug, or a poor implementation, is the **link** rule, as we already said, in fact, **TEXT** capture everything apart from certain special characters. You may want to only allows **WORD** and **WHITESPACE,** inside the parentheses, or to force a correct format for a link, inside the square brackets. On the other hand, this allows the user to make a mistake in writing the link without making the parser complain.

> You have to remember that the parser cannot check for semantics

For instance, it cannot know if the **WORD** indicating the color actually represents a valid color. That is to say, it doesn't know that it's wrong to use "dog", but it's right to use "red". This must be checked by the logic of the program, that can access which colors are available. You have to find the right balance of dividing enforcement between the grammar and your own code.

The parser should only check the syntax. So the rule of thumb is that when in doubt you let the parser pass the content up to your program. Then, in your program, you check the semantics and make sure that the rule actually have a proper meaning.

Let's look at the rule **color:** it can include a **message**, and it itself can be part of **message;** this ambiguity will be solved by the context in which is used.

11. Mistakes and Adjustments

Before trying our new grammar we have to add a name for it, at the beginning of the file. The name must be identical to the file name, which should have the .g4 extension.

```
1  grammar Chat;
```

You can find how to install everything, for your platform, in the official documentation. After everything is installed, we create the grammar, compile the generate Java code and then we run the testing tool.

```
1  // lines preceded by $ are commands
2  // > are input to the tool
3  // - are output from the tool
4  $ antlr4 Chat.g4
5  $ javac Chat*.java
6  // grun is the testing tool, Chat is the name of the grammar, chat the rule that we want to parse
7  $ grun Chat chat
8  > john SAYS: hello @michael this will not work
9  // CTRL+D on Linux, CTRL+Z on Windows
10 > CTRL+D/CTRL+Z
11 - line 1:0 mismatched input 'john SAYS: hello @michael this will not work' expecting WORD
```

Okay, it doesn't work. Why is it expecting **WORD**? It's right there! Let's try to find out, using the option -tokens to make it show the tokens it recognizes.

```
1  $ grun Chat chat -tokens
2  > john SAYS: hello @michael this will not work
3  - [@0,0:44='john SAYS: hello @michael this will not work',<TEXT>,1:0]
4  - [@1,45:44='<EOF>',<EOF>,2:0]
```

So it only sees the **TEXT** token. But we put it at the end of the grammar, what happens? The problem is that it always try to match the largest possible token. And all this text is a valid **TEXT** token. How do we solve this problem? There are many ways, the first, of course, is just getting rid of that token. But for now we are going to see the second easiest.

```
1  [..]
2
3  link            : TEXT TEXT ;
4
```

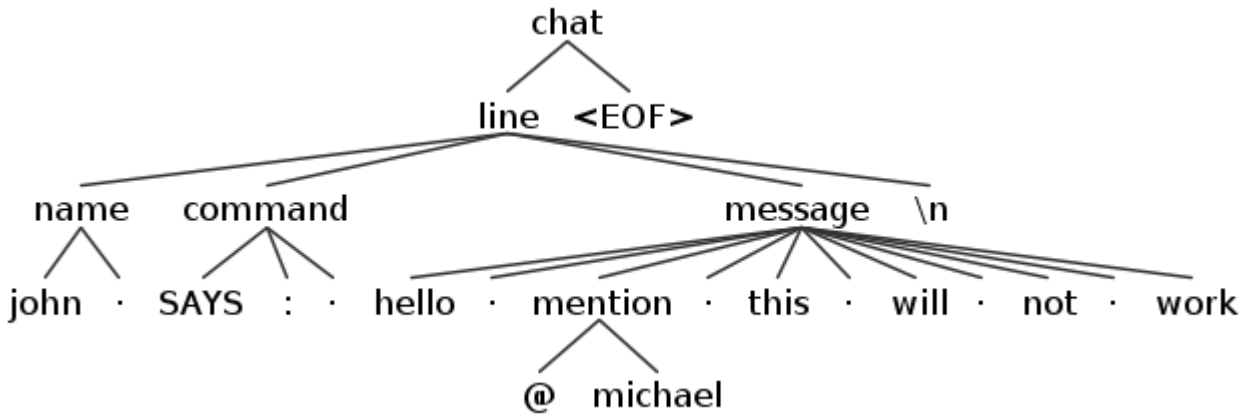
```
5  [..]
6
7  TEXT        : ('['|'(') ~[\])]+ (']'|')');
```

We have changed the problematic token to make it include a preceding parenthesis or square bracket. Note that this isn't exactly the same thing, because it would allow two series of parenthesis or square brackets. But it is a first step and we are learning here, after all.

Let's check if it works:

```
1  $ grun Chat chat -tokens
2  > john SAYS: hello @michael this will not work
3  - [@0,0:3='john',<WORD>,1:0]
4  - [@1,4:4=' ',<WHITESPACE>,1:4]
5  - [@2,5:8='SAYS',<SAYS>,1:5]
6  - [@3,9:9=':',<':'>,1:9]
7  - [@4,10:10=' ',<WHITESPACE>,1:10]
8  - [@5,11:15='hello',<WORD>,1:11]
9  - [@6,16:16=' ',<WHITESPACE>,1:16]
10 - [@7,17:17='@',<'@'>,1:17]
11 - [@8,18:24='michael',<WORD>,1:18]
12 - [@9,25:25=' ',<WHITESPACE>,1:25]
13 - [@10,26:29='this',<WORD>,1:26]
14 - [@11,30:30=' ',<WHITESPACE>,1:30]
15 - [@12,31:34='will',<WORD>,1:31]
16 - [@13,35:35=' ',<WHITESPACE>,1:35]
17 - [@14,36:38='not',<WORD>,1:36]
18 - [@15,39:39=' ',<WHITESPACE>,1:39]
19 - [@16,40:43='work',<WORD>,1:40]
20 - [@17,44:44='\n',<NEWLINE>,1:44]
21 - [@18,45:44='<EOF>',<EOF>,2:0]
```

Using the option `-gui` we can also have a nice, and easier to understand, graphical representation.



The dot in mid air represents whitespace.

This works, but it isn't very smart or nice, or organized. But don't worry, later we are going to see a better way. One positive aspect of this solution is that it allows to show another trick.

```
1   TEXT            : ('['|'(') .*? (']'|')');
```

This is an equivalent formulation of the token **TEXT**: the '.' matches any character, '*' says that the preceding match can be repeated any time, '?' indicate that the previous match is non-greedy. That is to say the previous subrule matches everything except what follows it, allowing to match the closing parenthesis or square bracket.

Mid-Level

In this section we see how to use ANTLR in your programs, the libraries and functions you need to use, how to test your parsers, and the like. We see what is and how to use a listener. We also build up on our knowledge of the basics, by looking at more advanced concepts, such as semantic predicates. While our projects are mainly in JavaScript and Python, the concept are generally applicable to every language. You can come back to this section when you need to remember how to get your project organized.

12. Setting Up the Chat Project with JavaScript

In the previous sections we have seen how to build a grammar for a chat program , piece by piece. Let's now copy that grammar we just created in the same folder of our Javascript files.

```
1   grammar Chat;
2
3   /*
4   * Parser Rules
```

```
5   */
6
7   chat            : line+ EOF ;
8
9   line            : name command message NEWLINE ;
10
11  message         : (emoticon | link | color | mention | WORD | WHITESPACE)+ ;
12
13  name            : WORD WHITESPACE;
14
15  command         : (SAYS | SHOUTS) ':' WHITESPACE ;
16
17  emoticon        : ':' '-'? ')'
18                  | ':' '-'? '('
19                  ;
20
21  link            : TEXT TEXT ;
22
23  color           : '/' WORD '/' message '/';
24
25  mention         : '@' WORD ;
26
27
28  /*
29  * Lexer Rules
30  */
31
32  fragment A      : ('A'|'a') ;
33  fragment S      : ('S'|'s') ;
34  fragment Y      : ('Y'|'y') ;
35  fragment H      : ('H'|'h') ;
```

```
36  fragment O        : ('O'|'o') ;
37  fragment U        : ('U'|'u') ;
38  fragment T        : ('T'|'t') ;
39
40  fragment LOWERCASE  : [a-z] ;
41  fragment UPPERCASE  : [A-Z] ;
42
43  SAYS              : S A Y S ;
44
45  SHOUTS            : S H O U T S ;
46
47  WORD              : (LOWERCASE | UPPERCASE | '_')+ ;
48
49  WHITESPACE        : (' ' | '\t')+ ;
50
51  NEWLINE           : ('\r'? '\n' | '\r')+ ;
52
53  TEXT              : ('['|'(') ~[\])]+ (']'|')');
```

We can create the corresponding Javascript parser simply by specifying the correct option with the ANTLR4 Java program.

```
1  antlr4 -Dlanguage=JavaScript Chat.g4
```

Now you will find some new files in the folder, with names such as ChatLexer.js, ChatParser.js and there are also *.tokens files, none of which contains anything interesting for us, unless you want to understand the inner workings of ANTLR.

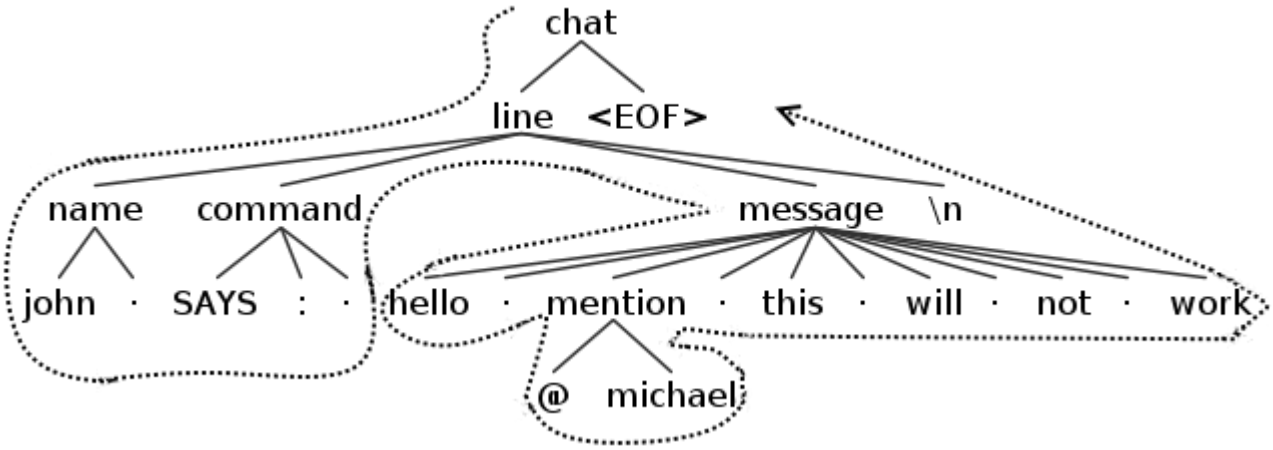
The file you want to look at is ChatListener.js, you are not going to modify anything in it, but it contains methods and functions that we will override with our own listener. We are not going to modify it, because changes would be overwritten every time the grammar is regenerated.

Looking into it you can see several enter/exit functions, a pair for each of our parser rules. These functions will be invoked when a piece of code matching the rule will be encountered. This is the default implementation of the listener that allows you to just override the functions that you need, on your derived listener, and leave the rest as is.

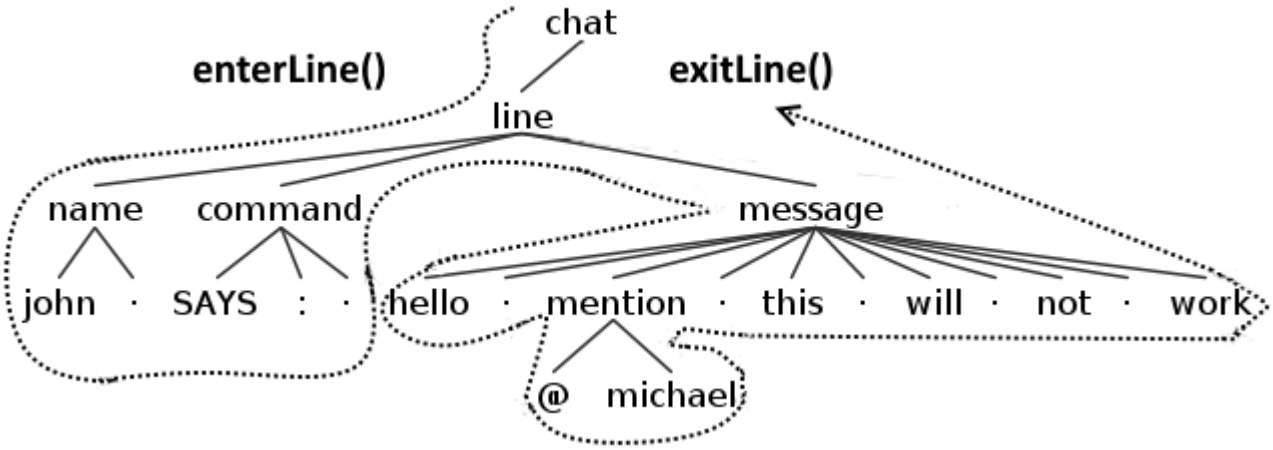
```
1  import antlr4 from 'antlr4';
2
3  // This class defines a complete listener for a parse tree produced by ChatParser.
4  export default class ChatListener extends antlr4.tree.ParseTreeListener {
5
6  // Enter a parse tree produced by ChatParser#chat.
7  enterChat(ctx) {
8  }
9
10 // Exit a parse tree produced by ChatParser#chat.
11 exitChat(ctx) {
12 }
13
14 [..]
```

The alternative to creating a Listener is creating a Visitor. The main differences are that you can neither control the flow of a listener nor return anything from its functions, while you can do both of them with a visitor. So if you need to control how the nodes of the AST are entered, or to gather information from several of them, you probably want to use a visitor. This is useful, for example, with code generation, where some information that is needed to create new source code is spread around many parts. Both the listener and the visitor use depth-first search.

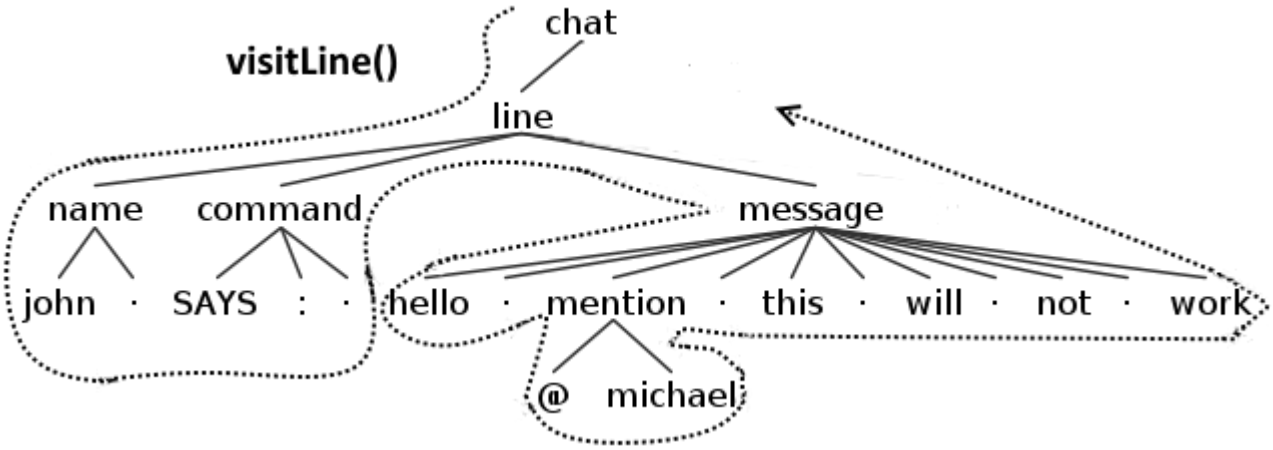
A depth-first search means that when a node will be accessed its children will be accessed, and if one of the children nodes had its own children they will be accessed before continuing on with the other children of the first node. The following image will make it simpler to understand the concept.



So in the case of a listener an enter event will be fired at the first encounter with the node and an exit one will be fired after having exited all of its children. In the following image you can see the example of what functions will be fired when a listener would met a **line** node (for simplicity only the functions related to **line** are shown).



With a standard visitor the behavior will be analogous except, of course, that only a single visit event will be fired for every single node. In the following image you can see the example of what function will be fired when a visitor would met a **line** node (for simplicity only the function related to **line** is shown).



Remember that **this is true for the default implementation of a visitor and it's done by returning the children of each node in every function**. If you override a method of the visitor it's your responsibility to make it continuing the journey or stop it right there.

13. Antlr.js

It is finally time to see how a typical ANTLR program looks.

```
1  import { createServer } from 'http';
2  import antlr4 from 'antlr4';
3  const { CommonTokenStream, InputStream } = antlr4;
4  import ChatLexer from './ChatLexer.js';
5  import ChatParser from './ChatParser.js';
6  import HtmlChatListener from './HtmlChatListener.js';
7
8  createServer((req, res) => {
9
10   res.writeHead(200, {
11       'Content-Type': 'text/html',
12   });
13
14   res.write('<html><head><meta charset="UTF-8"/></head><body>');
15
16   var input = "john SHOUTS: hello @michael /pink/this will work/ :-) \n";
17   var chars = new InputStream(input, true)
18   var lexer = new ChatLexer(chars);
19   var tokens  = new CommonTokenStream(lexer);
20   var parser = new ChatParser(tokens);
21
22   parser.buildParseTrees = true;
23   var tree = parser.chat();
24   var htmlChat = new HtmlChatListener(res);
25   antlr4.tree.ParseTreeWalker.DEFAULT.walk(htmlChat, tree);
```

```
26
27    res.write('</body></html>');
28    res.end();
29
30  }).listen(1337);
```

At the beginning of the main file we import the necessary libraries and file, `antlr4` (the runtime) and our generated parser, plus the listener that we are going to see later.

For simplicity we get the input from a string, while in a real scenario it would come from an editor.

> Lines 17-20 shows the foundation of every ANTLR program: you create the stream of chars from the input, you give it to the lexer and it transforms them in tokens, that are then interpreted by the parser.

It is useful to take a moment to reflect on this: the lexer works on the characters of the input, a copy of the input to be precise, while the parser works on the tokens generated by the lexer. **The lexer does not work on the input directly, and the parser does not even see the characters**.

This is important to remember in case you need to do something advanced like manipulating the input. In this case the input is a string, but, of course, it could be any stream of content.

The line 22 is redundant, since the option already defaults to true, , but it shows that you can enable or disable it.

Then, on line 23, we set the root node of the tree as a **chat** rule. You want to invoke the parser specifying a rule which typically is the first rule. However, you can actually invoke any rule directly, like **color**.

Once we get the parse tree from the parser typically we want to process it using a listener or a visitor. In this case we specify a listener. Our particular listener takes a parameter: the response object. We want to use it to put some text in the response to send to the user. After setting the listener up, we finally walk the tree with our listener.

14. HtmlChatListener.js

We continue by looking at the listener of our *Chat* project.

```
1  import antlr4 from 'antlr4';
2  import ChatLexer from './ChatLexer.js';
3  import ChatParser from './ChatParser.js';
4  import ChatListener  from './ChatListener.js';
5
```

```
6  export default class HtmlChatListener extends ChatListener {
7      constructor(res) {
8          super();
9          this.Res = res;
10     }
11
12     enterName(ctx) {
13         this.Res.write("<strong>");
14     }
15
16     exitName(ctx) {
17         this.Res.write(ctx.WORD().getText());
18         this.Res.write("</strong> ");
19     }
20
21     exitEmoticon(ctx) {
22         var emoticon = ctx.getText();
23
24         if(emoticon == ':-)' || emoticon == ':)')
25         {
26             ctx.text = "☺";
27         }
28
29         if(emoticon == ':-(' || emoticon == ':(')
30         {
31             ctx.text = "☹";
32         }
33     }
34
35     enterCommand(ctx) {
```

```
36          if(ctx.SAYS() != null)
37              this.Res.write(ctx.SAYS().getText() + ':' + '<p>');
38
39          if(ctx.SHOUTS() != null)
40                      this.Res.write(ctx.SHOUTS().getText()  +  ':'  +  '<p
style="text-transform: uppercase">');
41      }
42      exitLine(ctx) {
43          this.Res.write("</p>");
44      }
45  }
```

After the required function calls, we make our **HtmlChatListener** to extend **ChatListener.** The interesting stuff starts at line 12.

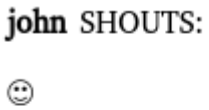
The **ctx** argument is an instance of a specific class context for the node that we are entering/exiting. So for `enterName` is `NameContext`, for `exitEmoticon` is `EmoticonContext`, etc. This specific context will have the proper elements for the rule, that would make possible to easily access the respective tokens and subrules. For example, `NameContext` will contain fields like **WORD()** and **WHITESPACE();** `CommandContext` will contain fields like **WHITESPACE()**, **SAYS()** and **SHOUTS().**

> These functions, enter* and exit*, are called by the walker everytime the corresponding nodes are entered or exited while it's traversing the AST that represents the program newline. A listener allows you to execute some code, but it's important to remember that **you can't stop the execution of the walker and the execution of the functions**.

On line 13, we start by printing a strong tag because we want the name to be bold, then on exitName we take the text from the token **WORD** and close the tag. Note that we ignore the **WHITESPACE** token, nothing says that we have to show everything. In this case we could have done everything either on the enter or exit function.

On the function *exitEmoticon* we simply transform the emoticon text in an emoji character. We get the text of the whole rule because there are no tokens defined for this parser rule. On *enterCommand*, instead there could be any of two tokens **SAYS** or **SHOUTS**, so we check which one is defined. And then we alter the following text, by transforming in uppercase, if it's a **SHOUT.** Note that we close the p tag at the exit of the **line** rule, because the command, semantically speaking, alter all the text of the message.

All we have to do now is launching node, with `node antlr.js`, and point our browser at its address, usually at `http://localhost:1337/` and we will be greeted with the following image.



So all is good, we just have to add all the different listeners to handle the rest of the language. Let's start with **color** and **message**.

15. Working with a Listener

We have seen how to start defining a listener. Now let's get serious and see how to evolve in a complete, robust listener. Let's start by adding support for **color** and checking the results of our hard work.

```
1  enterColor(ctx) {
2      var color = ctx.WORD().getText();
3      this.Res.write('<span style="color: ' + color + '">');
4  }
5
6  exitColor(ctx) {
7      this.Res.write("</span>");
8  }
9
10 exitMessage(ctx) {
11     this.Res.write(ctx.getText());
12 }
```

john SHOUTS:

THIS WILL WORK☺HELLO @MICHAEL /PINK/THIS WILL WORK/ :-)

Except that it does not work. Or maybe it works too much: we are writing some part of **message** twice ("this will work"): first when we check the specific nodes, children of **message**, and then at the end.

Luckily with JavaScript we can dynamically alter objects, so we can take advantage of this fact to change the `*Context` object themselves.

```
1  exitColor(ctx) {
2      ctx.text += ctx.message().text;
3      ctx.text += '</span>';
4  }
5
6  exitEmoticon(ctx) {
7      var emoticon = ctx.getText();
8
9      if(emoticon == ':-)' || emoticon == ':)')
10     {
11         ctx.text = "☺";
12     }
13
14     if(emoticon == ':-(' || emoticon == ':(')
15     {
16         ctx.text = "☹";
17     }
18 }
19
20 exitMessage(ctx) {
```

```
21      var text = '';
22
23      for (var index = 0; index <  ctx.children.length; index++ ) {
24          if(ctx.children[index].text != null)
25              text += ctx.children[index].text;
26          else
27              text += ctx.children[index].getText();
28      }
29
30      if(ctx.parentCtx instanceof ChatParser.LineContext == false)
31      {
32          ctx.text = text;
33      }
34      else
35      {
36          this.Res.write(text);
37          this.Res.write("</p>");
38      }
39  }
```

Only the modified parts are shown in the snippet above. We add a **text** field to every node that transforms its text, and then at the exit of every **message** we print the text if it's the primary message, the one that is directly child of the **line** rule. If it is a message, that is also a child of color, we add the **text** field to the node we are exiting and let **color** print it. We check this on line 30, where we look at the parent node to see if it is an instance of the object `LineContext`. This is also further evidence of how each **ctx** argument corresponds to the proper type.

Between lines 23 and 28 we can see another field of every node of the generated tree: `children`, which obviously contains the children node. You can observe that if a field **text** exists we add it to the proper variable, otherwise we use the usual function to get the text of the node.

16. Solving Ambiguities with Semantic Predicates

So far we have seen how to build a parser for a chat language in JavaScript. Let's continue working on this grammar but switch to Python. Remember that all code is available in the repository.

Before that, we have to solve an annoying problem: the **TEXT** token. The solution we have is terrible, and furthermore, if we tried to get the text of the token we would have to trim the edges, parentheses or square brackets. So what can we do?

We can use a particular feature of ANTLR called *semantic predicates.* As the name implies, they are expressions that produce a boolean value. They selectively enable or disable the following rule and thus permit to solve ambiguities. Another reason that they could be used is to support different versions of the same language, for instance a version with a new construct or an old without it.

Technically they are part of the larger group of *actions*, that allows to embed arbitrary code into the grammar. **The downside is that the grammar is no more language independent**, since the code in the action must be valid for the target language. For this reason, usually it's considered a good idea to only use semantic predicates, when they can't be avoided, and leave most of the code to the visitor/listener.

```
1  link            : '[' TEXT ']' '(' TEXT ')';
2
3  TEXT            : {self._input.LA(-1) == ord('[') or self._input.LA(-1) == ord('(')}? ~[\])]+ ;
```

We restored **link** to its original formulation, but we added a semantic predicate to the **TEXT** token, written inside curly brackets and followed by a question mark. We use `self._input.LA(-1)` to check the character before the current one, if this character is a square bracket or the open parenthesis, we activate the **TEXT** token. It is important to repeat that this must be valid code in our target language, it's going to end up in the generated Lexer or Parser, in our case in `ChatLexer.py`.

This matters not just for the syntax itself, but also because different targets might have different fields or methods, for instance LA returns an int in python, so we have to convert the `char` to a `int`.

Let's look at the equivalent form in other languages.

```
1  // C#. Notice that is .La and not .LA
2  TEXT : {_input.La(-1) == '[' || _input.La(-1) == '('}? ~[\])]+ ;
3  // Java
4  TEXT : {_input.LA(-1) == '[' || _input.LA(-1) == '('}? ~[\])]+ ;
5  // Javascript
6  TEXT : {this._input.LA(-1) == '[' || this._input.LA(-1) == '('}? ~[\])]+ ;
```

If you want to test for the preceding token, you can use the `_input.LT(-1)`, but you can only do that for parser rules. For example, if you want to enable the **mention** rule only if preceded by a **WHITESPACE** token.

```
// C#
mention: {_input.Lt(-1).Type == WHITESPACE}? '@' WORD ;
// Java
mention: {_input.LT(1).getType() == WHITESPACE}? '@' WORD ;
// Python
mention: {self._input.LT(-1).text == ' '}? '@' WORD ;
// Javascript
mention: {this._input.LT(1).text == ' '}? '@' WORD ;
```

17. Continuing the Chat in Python

Before seeing the Python example, we must modify our grammar and put the **TEXT** token before the **WORD** one. Otherwise ANTLR might assign the incorrect token, in cases where the characters between parentheses or brackets are all valid for **WORD**, for instance if it where `[this](link)`.

Using ANTLR in python is not more difficult than with any other platform, you just need to pay attention to the version of Python, 2 or 3.

```
antlr4 -Dlanguage=Python3 Chat.g4
```

And that's it. So when you have run the command, inside the directory of your python project, there will be a newly generated parser and a lexer. You may find interesting to look at ChatLexer.py and in particular the function TEXT_sempred (sempred stands for **sem**antic **pred**icate).

```
def TEXT_sempred(self, localctx:RuleContext, predIndex:int):
    if predIndex == 0:
        return self._input.LA(-1) == ord('[') or self._input.LA(-1) == ord('(')
```

You can see our predicate right in the code. This also means that you have to check that the correct libraries, for the functions used in the predicate, are available to the lexer.

18. The Python Way of Working with a Listener

The main file of a Python project is very similar to a JavaScript one, *mutatis mutandis* of course. That is to say we have to adapt libraries and functions to the proper version for a different language.

```
1  import sys
2  from antlr4 import *
3  from ChatLexer import ChatLexer
4  from ChatParser import ChatParser
5  from HtmlChatListener import HtmlChatListener
6
7  def main(argv):
8      input = FileStream(argv[1])
9      lexer = ChatLexer(input)
10     stream = CommonTokenStream(lexer)
11     parser = ChatParser(stream)
12     tree = parser.chat()
13
14     output = open("output.html","w")
15
16     htmlChat = HtmlChatListener(output)
17     walker = ParseTreeWalker()
18     walker.walk(htmlChat, tree)
19
20     output.close()
21
22 if __name__ == '__main__':
23     main(sys.argv)
```

We have also changed the input and output to become files, this avoid the need to launch a server in Python or the problem of using characters that are not supported in the terminal.

```
1  import sys
```

```
2  from antlr4 import *
3  from ChatParser import ChatParser
4  from ChatListener import ChatListener
5
6  class HtmlChatListener(ChatListener) :
7      def __init__(self, output):
8          self.output = output
9          self.output.write('<html><head><meta charset="UTF-8"/></head><body>')
10
11     def enterName(self, ctx:ChatParser.NameContext) :
12         self.output.write("<strong>")
13
14     def exitName(self, ctx:ChatParser.NameContext) :
15         self.output.write(ctx.WORD().getText())
16         self.output.write("</strong> ")
17
18     def enterColor(self, ctx:ChatParser.ColorContext) :
19         color = ctx.WORD().getText()
20         ctx.text = '<span style="color: ' + color + '">'
21
22     def exitColor(self, ctx:ChatParser.ColorContext):
23         ctx.text += ctx.message().text
24         ctx.text += '</span>'
25
26     def exitEmoticon(self, ctx:ChatParser.EmoticonContext) :
27         emoticon = ctx.getText()
28
29         if emoticon == ':-)' or emoticon == ':)' :
30             ctx.text = "☺"
31
32         if emoticon == ':-(' or emoticon == ':(' :
```

```
33          ctx.text = "☹"
34
35      def enterLink(self, ctx:ChatParser.LinkContext):
36          ctx.text = '<a href="%s">%s</a>' % (ctx.TEXT()[1], (ctx.TEXT()[0]))
37
38      def exitMessage(self, ctx:ChatParser.MessageContext):
39          text = ''
40
41          for child in ctx.children:
42              if hasattr(child, 'text'):
43                  text += child.text
44              else:
45                  text += child.getText()
46
47          if isinstance(ctx.parentCtx, ChatParser.LineContext) is False:
48              ctx.text = text
49          else:
50              self.output.write(text)
51              self.output.write("</p>")
52
53      def enterCommand(self, ctx:ChatParser.CommandContext):
54          if ctx.SAYS() is not None :
55              self.output.write(ctx.SAYS().getText() + ':' + '<p>')
56
57          if ctx.SHOUTS() is not None :
58              self.output.write(ctx.SHOUTS().getText() + ':' + '<p style="text-transform: uppercase">')
59
60      def exitChat(self, ctx:ChatParser.ChatContext):
61          self.output.write("</body></html>")
```

Apart from lines 35-36, where we introduce support for links, there is nothing new. Though you might notice that Python syntax is cleaner and, while having dynamic typing, it is not loosely typed

as Javascript. The different types of *Context objects are explicitly written out. If only Python tools were as easy to use as the language itself. But of course we cannot just fly over python like this, so we also introduce testing.

19. Testing with Python

While Visual Studio Code have a very nice extension for Python, that also supports unit testing, we are going to use the command line for the sake of compatibility.

```
1  python3 -m unittest discover -s . -p ChatTests.py
```

That's how you run the tests, but before that we have to write them. Actually, even before that, we have to write an ErrorListener to manage errors that we could find. While we could simply read the text outputted by the default error listener, there is an advantage in using our own implementation, namely that we can control more easily what happens.

```
1  import sys
2  from antlr4 import *
3  from ChatParser import ChatParser
4  from ChatListener import ChatListener
5  from antlr4.error.ErrorListener import *
6  import io
7
8  class ChatErrorListener(ErrorListener):
9
10     def __init__(self, output):
11         self.output = output
12         self._symbol = ''
13
14     def syntaxError(self, recognizer, offendingSymbol, line, column, msg, e):
15         self.output.write(msg)
16         self._symbol = offendingSymbol.text
17
18     @property
19     def symbol(self):
20         return self._symbol
```

Our class derives from ErrorListener and we simply have to implement syntaxError. Although we also add a property **symbol** to easily check which symbol might have caused an error.

```
1  from antlr4 import *
2  from ChatLexer import ChatLexer
3  from ChatParser import ChatParser
4  from HtmlChatListener import HtmlChatListener
5  from ChatErrorListener import ChatErrorListener
6  import unittest
7  import io
8
9  class TestChatParser(unittest.TestCase):
10
11     def setup(self, text):
12         lexer = ChatLexer(InputStream(text))
13         stream = CommonTokenStream(lexer)
14         parser = ChatParser(stream)
15
16         self.output = io.StringIO()
17         self.error = io.StringIO()
18
19         parser.removeErrorListeners()
20         errorListener = ChatErrorListener(self.error)
21         parser.addErrorListener(errorListener)
22
23         self.errorListener = errorListener
24
25         return parser
26
27     def test_valid_name(self):
28         parser = self.setup("John ")
```

```
29          tree = parser.name()
30
31          htmlChat = HtmlChatListener(self.output)
32          walker = ParseTreeWalker()
33          walker.walk(htmlChat, tree)
34
35          # let's check that there aren't any symbols in errorListener
36          self.assertEqual(len(self.errorListener.symbol), 0)
37
38      def test_invalid_name(self):
39          parser = self.setup("Joh-")
40          tree = parser.name()
41
42          htmlChat = HtmlChatListener(self.output)
43          walker = ParseTreeWalker()
44          walker.walk(htmlChat, tree)
45
46          # let's check the symbol in errorListener
47          self.assertEqual(self.errorListener.symbol, '-')
48
49  if __name__ == '__main__':
50      unittest.main()
```

The setup method is used to ensure that everything is properly set; on lines 19-21 we setup also our ChatErrorListener, but first we remove the default one, otherwise it would still output errors on the standard output. We are listening to errors in the parser, but we could also catch errors generated by the lexer. It depends on what you want to test. You may want to check both.

The two proper test methods check for a valid and an invalid name. The checks are linked to the property **symbol**, that we have previously defined, if it's empty everything is fine, otherwise it contains the symbol that created the error. Notice that on line 28, there is a space at the end of the string, because we have defined the rule **name** to end with a **WHITESPACE** token.

20. Parsing Markup

ANTLR can parse many things, including binary data, in that case tokens are made up of non-printable characters. But a more common problem is parsing markup languages such as XML or HTML. Markup is also a useful format to adopt for your own creations, because it allows to mix unstructured text content with structured annotations. They fundamentally represent a form of smart document, containing both text and structured data. The technical term that describes them is *island languages*. This type is not restricted to include only markup, and sometimes it is a matter of perspective.

For example, you may have to build a parser that only deal with preprocessor directives. In that case, you have to find a way to distinguish proper code from directives, which obeys different rules. In such an example the preprocessor directives might be considered an island language, a separate language surrounded by meaningless (for parsing purposes) text.

In any case, the problem for parsing such languages is that there is a lot of text that we don't actually have to parse, but we cannot ignore or discard, because the text contains useful information for the user and it is a structural part of the document. The solution is *lexical modes*, a way to parse structured content inside a larger sea of free text.

21. Lexical Modes

We are going to see how to use lexical modes, by starting with a new grammar.

```
1  lexer grammar MarkupLexer;
2
3  OPEN            : '[' -> pushMode(BBCODE) ;
4  TEXT            : ~('[')+ ;
5
6  // Parsing content inside tags
7  mode BBCODE;
8
9  CLOSE             : ']' -> popMode ;
10 SLASH             : '/' ;
11 EQUALS          : '=' ;
12 STRING          : '"' .*? '"' ;
13 ID            : LETTERS+ ;
14 WS              : [ \t\r\n] -> skip ;
15
```

```
16  fragment LETTERS    : [a-zA-Z] ;
```

Looking at the first line you could notice a difference: we are defining a `lexer grammar`, instead of the usual (combined) `grammar`. **You simply cannot define a lexical mode together with a parser grammar**. You can use lexical modes only in a lexer grammar, not in a combined grammar. The rest is not surprising, as you can see, we are defining a sort of BBCode markup, with tags delimited by square brackets.

On lines 3, 7 and 9 you will find basically all that you need to know about lexical modes. You define one or more tokens that can delimit the different modes and activate them.

The default mode is already implicitly defined, if you need to define yours you simply use mode followed by a name. Other than for markup languages, *lexical modes* are typically used to deal with string interpolation. That is when a string literal can contain more than simple text, for instance arbitrary expressions.

When we used a combined grammar we could define tokens implicitly: that is what happened when we used a string like '=' in a parser rule. Now that we are using separate lexer and parser grammars we cannot do that. That means that every single token has to be defined explicitly. So we have definitions like SLASH or EQUALS which typically could just be directly used in a parser rule. The concept is simple: **in the lexer grammar we need to define all tokens, because they cannot be defined later in the parser grammar.**

22. Parser Grammars

We look at the other side of a lexer grammar, so to speak.

```
1   parser grammar MarkupParser;
2
3   options { tokenVocab=MarkupLexer; }
4
5   file        : element* ;
6
7   attribute   : ID '=' STRING ;
8
9   content  : TEXT ;
10
11  element             : (content | tag) ;
12
13  tag         : '[' ID attribute? ']' element* '[' '/' ID ']' ;
```

On the first line we define a `parser grammar`. Since the tokens we need are defined in the lexer grammar, we need to use an option to say to ANTLR where it can find them. This is not necessary in combined grammars, since the tokens are defined in the same file.

There are many other options available, in the documentation.

There is almost nothing else to add, except that we define a **content** rule so that we can manage more easily the text that we find later in the program.

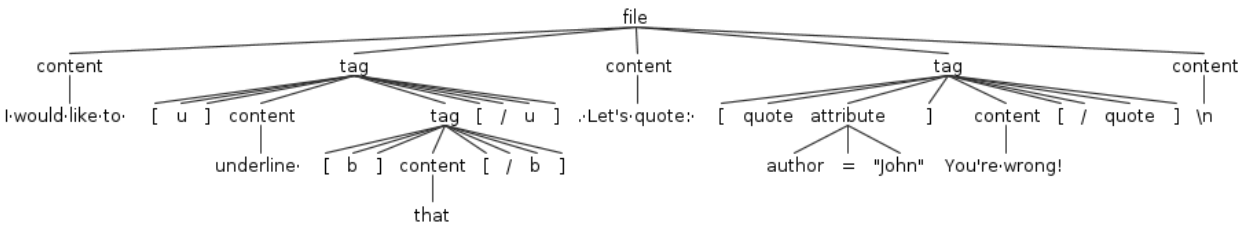
I just want to say that, as you can see, we do not need to explicitly use the tokens every time (e.g., SLASH), but instead we can use the corresponding text (e.g., '/').

ANTLR will automatically transform the text in the corresponding token, but this can happen only if they are already defined. In short, it is as if we had written:

```
1  tag : OPEN ID attribute? CLOSE element* OPEN SLASH ID CLOSE ;
```

But we could not have used the implicit way, if we had not already explicitly defined them in the lexer grammar. Another way to look at this is: when we define a combined grammar, ANTLR defines for us all the tokens that we have not explicitly defined ourselves. When we need to use a separate lexer and a parser grammar, we have to define explicitly every token ourselves. Once we have done that, we can use them in every way we want.

Before moving to actual Java code, let's see the AST for a sample input.



You can easily notice that the **element** rule is sort of transparent: where you would expect to find it there is always going to be a **tag** or **content**. So why did we define it? There are two advantages: avoid repetition in our grammar and simplify managing the results of the parsing.

We avoid repetition because if we did not have the element rule, we should repeat `(content|tag)` everywhere it is used. What if one day we add a new type of element? In addition to that, it simplifies the processing of the AST, because it makes easy to act upon both tag and content, given that you can use their common ancestor `(element)`.

Advanced

In this section we deepen our understanding of ANTLR. We will look at more complex examples and situations we may have to handle in our parsing adventures. We will learn how to perform more adavanced testing, to catch more bugs and ensure a better quality for our code. We will see

what a visitor is and how to use it. Finally, we will see how to deal with expressions and the complexity they bring.

You can come back to this section when you need to deal with complex parsing problems.

23. The Markup Project in Java

You can follow the instructions in Java Setup or just copy the `antlr-java` folder of the companion repository. Once the file pom.xml is properly configured, this is how you build and execute the application.

```
# use gradle to build the project
./gradlew compileJava
# if you are not using an IDE
# and you have defined the fatJar task as in the repository
./gradlew fatJar
java -jar .\build\libs\markup-example-gradle-all.jar
```

As you can see, it is not any different from any typical Gradle project, although it is indeed more complicated than a typical JavasSript or Python project. Of course, if you use an IDE you do not need to do anything different from your typical workflow.

24. The Main App.java

We are going to see how to write a typical ANTLR application in Java.

```
package me.tomassetti.examples.MarkupParser;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class App
{
   public static void main( String[] args )
   {
      CharStream inputStream = CharStreams.fromString(
         "I would like to [b][i]emphasize[/i][/b] this and [u]underline [b]that[/b][/u] ." +
         "Let's not forget to quote: [quote author=\"John\"]You're wrong![/quote]");
      MarkupLexer markupLexer = new MarkupLexer(inputStream);
```

```
13        CommonTokenStream commonTokenStream = new CommonTokenStream(markupLexer);
14        MarkupParser markupParser = new MarkupParser(commonTokenStream);
15
16        MarkupParser.FileContext fileContext = markupParser.file();
17        MarkupVisitor visitor = new MarkupVisitor();
18        visitor.visit(fileContext);
19    }
20 }
```

There is a small surprise regarding the `inputStream` variable. Instead of using an `[ANTLR]InputStream` class we are using a `CharStream` one. This difference applies to Java and C# and it is due to Unicode support. The gist is that previous version of the runtimes for these languages supported only partially Unicode, so to avoid any surprising change in behavior the old [ANTLR]InputStream was deprecated. You can read more on the official ANTLR documentation.

Apart from this change, at this point the main Java file should not come as a surprise, the only new development is the visitor. Of course, there are the obvious little differences in the names of the ANTLR classes and such. This time we are building a visitor, whose main advantage is the chance to control the flow of the program. While we are still dealing with text, we do not want to display it, we want to transform it from pseudo-BBCode to pseudo-Markdown.

25. Transforming Code with ANTLR

The first issue to deal with our translation from pseudo-BBCode to pseudo-Markdown is a design decision. Our two languages are different and frankly neither of the two original one is that well designed.

BBCode was created as a safety precaution, to make possible to disallow the use of HTML but give some of its power to users. Markdown was created to be an easy to read and write format, that could be translated into HTML. So they both mimic HTML, and you can actually use HTML in a Markdown document. Let's start to look into how messy a real conversion would be.

```
1  package me.tomassetti.examples.MarkupParser;
2
3  import org.antlr.v4.runtime.*;
4  import org.antlr.v4.runtime.misc.*;
5  import org.antlr.v4.runtime.tree.*;
6
7  public class MarkupVisitor extends MarkupParserBaseVisitor<String>
```

```
8   {
9       @Override
10      public String visitFile(MarkupParser.FileContext context)
11      {
12              visitChildren(context);
13
14              System.out.println("");
15
16              return null;
17      }
18
19      @Override
20      public String visitContent(MarkupParser.ContentContext context)
21      {
22          System.out.print(context.TEXT().getText());
23
24          return visitChildren(context);
25      }
26  }
```

The first version of our visitor prints all the text and ignore all the tags.

You can see how to control the flow, either by calling visitChildren, or any other visit* function, and deciding what to return. We just need to override the methods that we want to change. Otherwise, the default implementation would just do like `visitContent`, on line 24, it will visit the children nodes and allows the visitor to continue. Just like for a listener, the argument is the proper context type. If you want to stop the visitor just return null as on line 16.

26. Joy and Pain of Transforming Code

Transforming code, even at a very simple level, comes with some complications. Let's start easy with some basic visitor methods.

```
1   @Override
2   public String visitContent(MarkupParser.ContentContext context)
3   {
```

```
4     return context.getText();
5  }
6
7  @Override
8  public String visitElement(MarkupParser.ElementContext context)
9  {
10    if(context.parent instanceof MarkupParser.FileContext)
11    {
12       if(context.content() != null)
13          System.out.print(visitContent(context.content()));
14       if(context.tag() != null)
15          System.out.print(visitTag(context.tag()));
16    }
17
18    return null;
19 }
```

Before looking at the main method, let's look at the supporting ones. Foremost we have changed `visitContent` by making it return its text instead of printing it. Second, we have overridden the `visitElement` so that it prints the text of its child, but only if it is a top element, and not inside a **tag**. In both cases, it achieves this by calling the proper `visit*` method. It knows which one to call because it checks if it actually has a **tag** or **content** node.

```
1  @Override
2  public String visitTag(MarkupParser.TagContext context)
3  {
4     String text = "";
5     String startDelimiter = "", endDelimiter = "";
6
7     String id = context.ID(0).getText();
8
9     switch(id)
10    {
```

```
11          case "b":
12              startDelimiter = endDelimiter = "**";
13          break;
14          case "u":
15              startDelimiter = endDelimiter = "*";
16          break;
17          case "quote":
18              String attribute = context.attribute().STRING().getText();
19              attribute = attribute.substring(1,attribute.length()-1);
20              startDelimiter = System.lineSeparator() + "> ";
21              endDelimiter = System.lineSeparator() + "> " + System.lineSeparator() + "> – "
22                          + attribute + System.lineSeparator();
23          break;
24      }
25
26      text += startDelimiter;
27
28      for (MarkupParser.ElementContext node: context.element())
29      {
30          if(node.tag() != null)
31              text += visitTag(node.tag());
32          if(node.content() != null)
33              text += visitContent(node.content());
34      }
35
36      text += endDelimiter;
37
38      return text;
39  }
```

`VisitTag` contains more code than every other method, because it can also contain other elements, including other tags that have to be managed themselves, and thus they cannot be

simply printed. We save the content of the **ID** on line 7, of course we don't need to check that the corresponding end tag matches, because the parser will ensure that, as long as the input is well formed.

The first complication starts with at lines 14-15: as it often happens when transforming a language in a different one, there isn't a perfect correspondence between the two. While BBCode tries to be a smarter and safer replacement for HTML, Markdown wants to accomplish the same objective of HTML, to create a structured document. So BBCode has an underline tag, while Markdown does not.

> So we have to make a decision

Do we want to discard the information, or directly print HTML, or something else? We choose something else and instead convert the underline to an italic. That might seem completely arbitrary, and indeed there is an element of choice in this decision. But the conversion forces us to lose some information, and both are used for emphasis, so we choose the closer thing in the new language.

The following case, on lines 18-22, force us to make another choice. We can't maintain the information about the author of the quote in a structured way, so we choose to print the information in a way that will make sense to a human reader.

On lines 28-34 we do our *magic*: we visit the children and gather their text, then we close with the **endDelimiter**. Finally we return the text that we have created.

> That's how the visitor works

1. every top **element** visits each child
    - if it's a **content** node, it directly returns the text
    - if it's a **tag**, it setups the correct delimiters and then it checks its children. It repeats step 2 for each child and then it returns the gathered text
2. it prints the returned text

It's obviously a simple example, but it shows how you can have great freedom in managing the visitor once you have launched it. Together with the patterns that we have seen at the beginning of this section you can see all of the options: to return null to stop the visit, to return children to continue, to return something to perform an action ordered at a higher level of the tree.

27. Advanced Testing

The use of lexical modes permits handling the parsing of island languages, but it complicates testing.

We are not going to show `MarkupErrorListener.java` because we did not change it; if you need you can see it on the repository.

You can run the tests by using the following command.

```
./gradlew test
```

Now we are going to look at the tests code. We are skipping the setup part, because that also is obvious, we just copy the process seen on the main file, but we simply add our error listener to intercept the errors.

```
    // private variables inside the class AppTest
    private MarkupErrorListener errorListener;
    private MarkupLexer markupLexer;

    @Test
    public void testText()
    {
        MarkupParser parser = setup("anything in here");

        MarkupParser.ContentContext context = parser.content();

        assertEquals("",this.errorListener.getSymbol());
    }

    @Test
    public void testInvalidText()
    {
        MarkupParser parser = setup("[anything in here");

        MarkupParser.ContentContext context = parser.content();

        // note that this.errorListener.symbol could be empty
        // when ANTLR doesn't recognize the token or there is no error.

        // In such cases check the output of errorListener
```

```
25          assertEquals("[",this.errorListener.getSymbol());
26      }
27
28      @Test
29      public void testWrongMode()
30      {
31          MarkupParser parser = setup("author=\"john\"");
32
33          MarkupParser.AttributeContext context = parser.attribute();
34          TokenStream ts = parser.getTokenStream();
35
36          assertEquals(MarkupLexer.DEFAULT_MODE, markupLexer._mode);
37          assertEquals(MarkupLexer.TEXT,ts.get(0).getType());
38          assertEquals("author=\"john\"",this.errorListener.getSymbol());
39      }
40
41      @Test
42      public void testAttribute()
43      {
44          MarkupParser parser = setup("author=\"john\"");
45          // we have to manually push the correct mode
46          this.markupLexer.pushMode(MarkupLexer.BBCODE);
47
48          MarkupParser.AttributeContext context = parser.attribute();
49          TokenStream ts = parser.getTokenStream();
50
51          assertEquals(MarkupLexer.ID,ts.get(0).getType());
52          assertEquals(MarkupLexer.EQUALS,ts.get(1).getType());
53          assertEquals(MarkupLexer.STRING,ts.get(2).getType());
```

```
54
55          assertEquals("",this.errorListener.getSymbol());
56      }
57
58      @Test
59      public void testInvalidAttribute()
60      {
61          MarkupParser parser = setup("author=/\"john\"");
62          // we have to manually push the correct mode
63          this.markupLexer.pushMode(MarkupLexer.BBCODE);
64
65           MarkupParser.AttributeContext context = parser.attribute();
66
67          assertEquals("/",this.errorListener.getSymbol());
68      }
```

The first two methods are exactly as before, we simply check that there are no errors, or that there is the correct one because the input itself is erroneous. On lines 36-38 things start to get interesting: the issue is that by testing the rules one by one we do not give the chance to the parser to switch automatically to the correct mode. So it remains always on the DEFAULT_MODE, which in our case makes everything looks like **TEXT**. This obviously makes the correct parsing of an **attribute** impossible.

The same lines also show how you can check the current mode that you are in, and the exact type of the tokens that are found by the parser, which we use to confirm that indeed all is wrong in this case.

While we could use a string of text to trigger the correct mode, each time, that would make testing intertwined with several pieces of code, which is a no-no. So the solution is seen on line 46: we trigger the correct mode manually. Once you have done that, you can see that our attribute is recognized correctly.

28. Dealing with Expressions

So far we have written simple parser rules, now we are going to see one of the most challenging parts in analyzing a real (programming) language: expressions. While rules for statements are

usually larger, they are quite simple to deal with: you just need to write a rule that encapsulate the structure with all the different optional parts. For instance, a `for` statement can include all other kinds of statements, but we can simply include them with something like `statement*`. An expression, instead, can be combined in many different ways.

An expression usually contains other expressions. For example, the typical binary expression is composed by an expression on the left, an operator in the middle and another expression on the right. This can lead to ambiguities. Think, for example, at the expression 5 + 3 * 2, for ANTLR this expression is ambiguous because there are two ways to parse it. It could either parse it as 5 + (3 * 2) or (5 +3) * 2.

Until this moment we have avoided the problem simply because markup constructs surround the object on which they are applied. So there is not ambiguity in choosing which one to apply first: it's the most external. Imagine if this expression was written as:

```
1  <add>
2    <int>5</int>
3    <mul>
4      <int>3</int>
5      <int>2</int>
6    </mul>
7  </add>
```

That would make obvious to ANTLR how to parse it.

These types of rules are called *left-recursive rules.* You might say: just parse whatever comes first. The problem with that is semantic: the addition comes first, but we know that multiplications have a precedence over additions. Traditionally the way to solve this problem was to create a complex cascade of specific expressions like this:

```
1  expression       : addition;
2  addition         : multiplication ('+' multiplication)* ;
3  multiplication : atom ('*' atom)* ;
4  atom             : NUMBER ;
```

This way ANTLR would have known to search first for a number, then for multiplications and finally for additions. This is cumbersome and also counterintuitive, because the last expression is the first to be actually recognized. Luckily **ANTLR4 can create a similar structure automatically, so we can use a much more natural syntax**.

```
1  expression : expression '*' expression
2             | expression '+' expression
3             | NUMBER
4             ;
```

In practice ANTLR consider the order in which we defined the alternatives to decide the precedence. By writing the rule in this way we are telling to ANTLR that the multiplication takes precedence over the addition.

29. Parsing Spreadsheets

Now we are prepared to create our last application, in C#. We are going to build the parser of an Excel-like application. In practice, we want to manage the expressions you write in the cells of a spreadsheet.

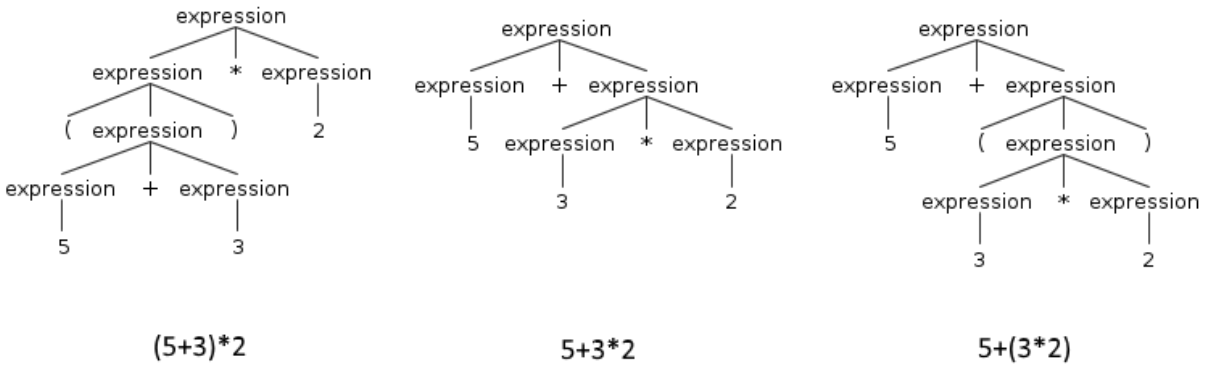
```
1  grammar Spreadsheet;
2
3  expression        : '(' expression ')'                  #parenthesisExp
4                    | expression (ASTERISK|SLASH) expression   #mulDivExp
5                    | expression (PLUS|MINUS) expression      #addSubExp
6                    | <assoc=right> expression '^' expression  #powerExp
7                    | NAME '(' expression ')'             #functionExp
8                    | NUMBER                              #numericAtomExp
9                    | ID                                  #idAtomExp
10                   ;
11
12 fragment LETTER : [a-zA-Z] ;
13 fragment DIGIT    : [0-9] ;
14
15 ASTERISK         : '*' ;
16 SLASH            : '/' ;
17 PLUS             : '+' ;
18 MINUS            : '-' ;
19
```

```
20  ID              : LETTER DIGIT ;
21
22  NAME            : LETTER+ ;
23
24  NUMBER          : DIGIT+ ('.' DIGIT+)? ;
25
26  WHITESPACE      : ' ' -> skip;
```

With all the knowledge you have acquired so far everything should be clear, except for possibly three things:

1. why the parentheses are there,
2. what's the stuff on the right,
3. that thing on line 6.

The parentheses comes first because their only role is to give the user a way to override the precedence of operator, if he needs to do so. This graphical representation of the AST should make it clear.

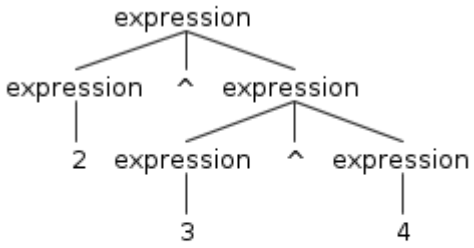


The things on the right are *labels*, they are used to make ANTLR generate specific functions for the visitor or listener. So there will be a `VisitFunctionExp`, a `VisitPowerExp`, etc. This makes possible to avoid the use of a giant visitor for the **expression** rule.

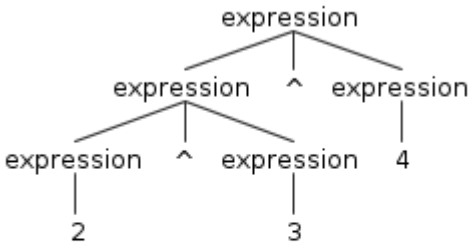
The expression relative to exponentiation is different because there are two possible ways to act, to group them, when you meet two sequential expressions of the same type. The first one is to execute the one on the left first and then the one on the right, the second one is the inverse: this is called *associativity*. Usually the one that you want to use is *left-associativity,* which is the default option. Nonetheless exponentiation is *right-associative*, so we have to signal this to ANTLR.

Another way to look at this is: if there are two expressions of the same type, which one has the precedence: the left one or the right one? Again, an image is worth a thousand words.

We also have



<assoc=right>



(Implicit) <assoc=left>

support for functions, alphanumeric variables that represents cells and real numbers.

30. The Spreadsheet Project in C#

You just need to follow the C# Setup: to install a nuget package for the runtime and an ANTLR4 extension for Visual Studio. The extension will automatically generate everything whenever you build your project: parser, listener and/or visitor.

Notice that there are two small differences between the code for a project using the extension and one using the Java tool. These are noted in the README for the C# project at the repository.

After you have done that, you can also add grammar files just by using the usual menu Add -> New Item. Do exactly that to create a grammar called `Spreadsheet.g4` and put in it the grammar we have just created. Now let's see the main Program.cs.

```
1  using System;
2  using Antlr4.Runtime;
3
4  namespace AntlrTutorial
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             string input = "log(10 + A1 * 35 + (5.4 - 7.4))";
11
12             ICharStream inputStream = CharStreams.fromString (input);
13             SpreadsheetLexer spreadsheetLexer = new SpreadsheetLexer(inputStream);
```

```
14          CommonTokenStream commonTokenStream = new CommonTokenStream(spreadsheetLexer);
15          SpreadsheetParser spreadsheetParser = new SpreadsheetParser(commonTokenStream);
16
17          SpreadsheetParser.ExpressionContext expressionContext = spreadsheetParser.expression();
18          SpreadsheetVisitor visitor = new SpreadsheetVisitor();
19
20          Console.WriteLine(visitor.Visit(expressionContext));
21      }
22  }
23  }
```

There is nothing to say, apart from that, of course, you have to pay attention to yet another slight variation in the naming of things: pay attention to the casing. For instance, `ICharStream`, in the C# program, was `CharStream` in the Java program.

Also you can notice that, this time, we output on the screen the result of our visitor, instead of writing the result on a file.

31. Excel is Doomed

We are going to take a look at our visitor for the *Spreadsheet* project.

```
1   public class SpreadsheetVisitor : SpreadsheetBaseVisitor<double>
2   {
3       private static DataRepository data = new DataRepository();
4
5       public override double VisitNumericAtomExp(SpreadsheetParser.NumericAtomExpContext context)
6       {
7           return double.Parse(context.NUMBER().GetText(), System.Globalization.CultureInfo.InvariantCulture);
8       }
9
10      public override double VisitIdAtomExp(SpreadsheetParser.IdAtomExpContext context)
11      {
12          String id = context.ID().GetText();
13
```

```
14          return data[id];
15      }
16
17      public override double VisitParenthesisExp(SpreadsheetParser.ParenthesisExpContext context)
18      {
19          return Visit(context.expression());
20      }
21
22      public override double VisitMulDivExp(SpreadsheetParser.MulDivExpContext context)
23      {
24          double left = Visit(context.expression(0));
25          double right = Visit(context.expression(1));
26          double result = 0;
27
28          if (context.ASTERISK() != null)
29              result = left * right;
30          if (context.SLASH() != null)
31              result = left / right;
32
33          return result;
34      }
35
36      [..]
37
38      public override double VisitFunctionExp(SpreadsheetParser.FunctionExpContext context)
39      {
40          String name = context.NAME().GetText();
41          double result = 0;
42
43          switch(name)
44          {
```

```
45          case "sqrt":
46              result = Math.Sqrt(Visit(context.expression()));
47              break;
48
49          case "log":
50              result = Math.Log10(Visit(context.expression()));
51              break;
52      }
53
54      return result;
55    }
56  }
```

`VisitNumeric` and `VisitIdAtom` return the actual numbers that are represented either by the literal number or the variable. In a real scenario **DataRepository** would contain methods to access the data in the proper cell, but in our example is just a Dictionary with some keys and numbers. The other methods actually work in the same way: they visit/call the containing expression(s). The only difference is what they do with the results.

Some perform an operation on the result, the binary operations combine two results in the proper way and finally `VisitParenthesisExp` just reports the result higher on the chain. Math is simple, when it is done by a computer.

32. Testing Everything

Up until now we have only tested the parser rules, that is to say we have tested only if we have created the correct rule to parse our input. Now we are also going to test the visitor functions. This is the ideal chance because our visitor returns values that we can check individually. In other occasions, for instance if your visitor prints something to the screen, you may want to rewrite the visitor to write on a stream. Then, at testing time, you can easily capture the output.

We are not going to show `SpreadsheetErrorListener.cs` because it is the same as the previous one we have already seen; if you need it you can see it on the repository.

To perform unit testing on Visual Studio you need to create a specific project inside the solution. You can choose different formats, we opt for the xUnit version. To run them there is an aptly named section “TEST” on the menu bar in Visual Studio or the command dotnet test on the command line.

```
1  [Fact]
```

```
2   public void testExpressionPow()
3   {
4       setup("5^3^2");
5
6       PowerExpContext context = parser.expression() as PowerExpContext;
7
8       CommonTokenStream ts = (CommonTokenStream)parser.InputStream;
9
10      Assert.Equal(SpreadsheetLexer.NUMBER, ts.Get(0).Type);
11      Assert.Equal(SpreadsheetLexer.T__2, ts.Get(1).Type);
12      Assert.Equal(SpreadsheetLexer.NUMBER, ts.Get(2).Type);
13      Assert.Equal(SpreadsheetLexer.T__2, ts.Get(3).Type);
14      Assert.Equal(SpreadsheetLexer.NUMBER, ts.Get(4).Type);
15  }
16
17  [Fact]
18  public void testVisitPowerExp()
19  {
20      setup("4^3^2");
21
22      PowerExpContext context = parser.expression() as PowerExpContext;
23
24      SpreadsheetVisitor visitor = new SpreadsheetVisitor();
25      double result = visitor.VisitPowerExp(context);
26
27      Assert.Equal(double.Parse("262144"), result);
28  }
29
30  [..]
```

```
31
32  [Fact]
33  public void testWrongVisitFunctionExp()
34  {
35      setup("logga(100)");
36
37          FunctionExpContext context = parser.expression() as
    FunctionExpContext;
38
39      SpreadsheetVisitor visitor = new SpreadsheetVisitor();
40      double result = visitor.VisitFunctionExp(context);
41
42      CommonTokenStream ts = (CommonTokenStream)parser.InputStream;
43
44      Assert.Equal(SpreadsheetLexer.NAME, ts.Get(0).Type);
45      Assert.Equal(null, errorListener.Symbol);
46      Assert.Equal(0, result);
47  }
48
49  [Fact]
50  public void testCompleteExp()
51  {
52      setup("log(5+6*7/8)");
53
54      ExpressionContext context = parser.expression();
55
56      SpreadsheetVisitor visitor = new SpreadsheetVisitor();
57      double result = visitor.Visit(context);
58
```

```
59      Assert.Equal("1.0107238653917732",
result.ToString(System.Globalization.CultureInfo.GetCultureInfo("en-
US").NumberFormat));
60  }
```

The first test function is similar to the ones we have already seen; it checks that the corrects tokens are selected. On line 11 and 13 you may be surprised to see that weird token type, this happens because we didn't explicitly created one for the '^' symbol so one got automatically created for us. If you need you can see all the tokens by looking at the *.tokens file generated by ANTLR.

On line 25 we visit our test node and get the results, that we check on line 27. It's all very simple because our visitor is simple, while unit testing should always be easy and made up of small parts it really can't be easier than this.

The only thing to pay attention to is related to the format of the number, it's not a problem here, but look at line 59, where we test the result of a whole expression. There we need to make sure that the correct format is selected, because different countries use different symbols as the decimal mark.

> There are some things that depends on the cultural context

If your computer was already set to the *American English Culture* this wouldn't be necessary, but to guarantee the correct testing results for everybody we have to specify it. Keep that in mind if you are testing things that are culture-dependent: such as grouping of digits, temperatures, etc.

On line 44-46 you see than when we check for the wrong function the parser actually works. That is because indeed `logga` is syntactically valid as a function name, but it is not semantically correct. The function `logga` does not exists, so our program does not know what to do with it. So when we visit it we get 0 as a result. As you recall this was our choice: since we initialize the result to 0 and we do not have a default case in VisitFunctionExp. So if there is no function the result remains 0. A possible alternative could be to throw an exception.

Final Remarks

In this section we see tips and tricks that never came up in our example, but can be useful in your programs. We suggest more resources you may find useful if you want to know more about ANTLR, both the practice and the theory, or you need to deal with the most complex problems.

33. Tips and Tricks

Let's see a few tricks that could be useful from time to time. These were never needed in our examples, but they have been quite useful in other scenarios.

Catchall Rule

The first one is the **ANY** lexer rule. This is simply a rule in the following format.

```
1  ANY : . ;
```

This is a catchall rule that should be put at the end of your grammar. It matches any character that didn't find its place during the parsing. So creating this rule can help you during development, when your grammar has still many holes that could cause distracting error messages. It's even useful during production, when it acts as a canary in the mines. If it shows up in your program you know that something is wrong.

Channels

There is also something that we have not talked about: *channels*. Their use case is usually handling comments. You do not really want to check for comments inside every of your statements or expressions, so you usually throw them away with -> skip. But there are some cases where you may want to preserve them, for instance if you are translating a program in another language. When this happens, you use *channels*. There is already one called HIDDEN that you can use, but you can declare more of them at the top of your lexer grammar.

```
1  channels { UNIQUENAME }
2  // and you use them this way
3  COMMENTS : '//' ~[\r\n]+ -> channel(UNIQUENAME) ;
```

Rule Element Labels

There is another use of labels other than to distinguish among different cases of the same rule. They can be used to give a specific name, usually but not always of semantic value, to a common rule or parts of a rule. The format is label=rule, to be used inside another rule.

```
1  expression : left=expression (ASTERISK|SLASH) right=expression ;
```

This way **left** and **right** would become fields in the `ExpressionContext` nodes. And instead of using `context.expression(0)`, you could refer to the same entity using `context.left`.

Problematic Tokens

In many real languages some symbols are reused in different ways, some of which may lead to ambiguities. A common problematic example are the angle brackets, used both for bitshift expression and to delimit parameterized types.

```
1  // bitshift expression, it assigns to x the value of y shifted by three bits
2  x = y >> 3;
3  // parameterized types, it define x as a list of dictionaries
4  List<Dictionary<string, int>> x;
```

The natural way of defining the bitshift operator token is as a single double angle brackets, '>>'. But this might lead to confusing a nested parameterized definition with the bitshift operator, for instance in the second example shown up here. While a simple way of solving the problem would be using semantic predicates, an excessive number of them would slow down the parsing phase. The solution is to avoid defining the bitshift operator token and instead using the angle brackets twice in the parser rule, so that the parser itself can choose the best candidate for every occasion.

```
1  // from this
2  RIGHT_SHIFT : '>>';
3  expression : ID RIGHT_SHIFT NUMBER;
4  // to this
5  expression : ID SHIFT SHIFT NUMBER;
```

34. Conclusions

We have learned a lot today:

- what are a lexer and a parser
- how to create lexer and parser rules
- how to use ANTLR to generate parsers in Java, C#, Python and JavaScript
- the fundamental kinds of problems you will encounter parsing and how to solve them
- how to understand errors
- how to test your parsers

That's all you need to know to use ANTLR on your own. And I mean literally, you may want to know more, but now you have solid basis to explore on your own.

Where to look if you need more information about ANTLR:

- On this very website there is whole category dedicated to ANTLR.
- The official ANTLR website is a good starting point to know the general status of the project, the specialized development tools and related project, like StringTemplate

- The ANTLR documentation on GitHub; especially useful are the information on targets and how to setup it on different languages.
- The ANTLR API; it's related to the Java version, so there might be some differences in other languages, but it's the best place where to settle your doubts about the inner workings of this tool.
- For the very interested in the science behind ANTLR4, there is an academic paper: *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*
- **The Definitive ANTLR 4 Reference**, by the man itself, *Terence Parr*, the creator of ANTLR. The resource you need if you want to know everything about ANTLR and a good deal about parsing languages in general.

Also the book is only place where you can find an answer to question like these:

> ANTLR v4 is the result of a minor detour (twenty-five years) I took in graduate school. I guess I'm going to have to change my motto slightly.
>
> *Why program by hand in five days what you can spend twenty-five years of your life automating?*

If instead you decide you could use some help with your projects involving ANTLR, you can also use our ANTLR Consulting Services.

We would like to thank Bernard Kaiflin for having revised the document and helped us improving it.

We would like to thank: Brasilio Castilho, Andy Nicholas for having spotted errors and typos in the article.

We worked quite hard to build the largest tutorial on ANTLR: the mega-tutorial! A post over 14.000 words long, or more than 70 pages, to try answering all your questions about ANTLR. Missing something? Contact us and let us now, we are here to help.