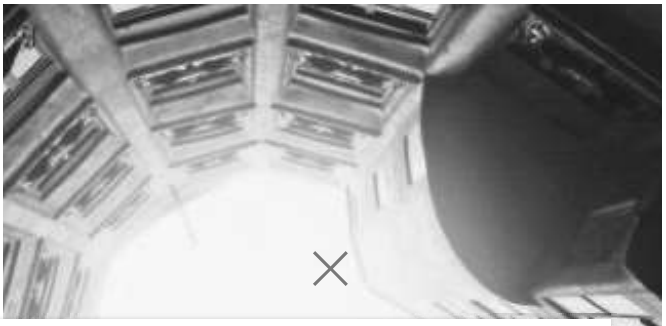
# Parsing in Java: Tools and L



W
G
in
P

If you need to parse a language, or document, from Java there are fundamentally three ways to solve the problem:

- use an existing library supporting that specific language: for example a library to parse XML
- building your own custom parser by hand
- a tool or library to generate a parser: for example ANTLR, that you can use to build parsers for any language

## Use An Existing Library



parser generators, or you have specific requirements

that you cannot satisfy using a typical parser generator. For instance, because you need the best possible performance or a deep integration between different components.



## A Tool Or Library To Generate A Parser

In all other cases the third option should be the default one, because is the one that is most flexible and has the shorter development time. That is why on this article we concentrate on the tools and libraries that correspond to this option.

*Note: text in blockquote describing a program comes from the respective documentation*

# Tools To Create Parsers



We are going to see:

- tools that can generate parsers usable from Java (and possibly from other languages)
- Java libraries to build parsers

Tools that can be used to generate the code for a parser are called **parser generators** or **compiler compiler**. Libraries that create parsers are known as **parser combinators**.

Parser generators (or parser combinators) are not trivial: you need some time to learn how to use them

# Useful Things To Know About Parsers



To make sure that these list is accessible to all programmers we have prepared a short explanation for terms and concepts that you may encounter searching for a parser. We are not trying to give you formal explanations, but practical ones.

## Structure Of A Parser

A parser is usually composed of two parts: a *lexer*, also known as *scanner* or *tokenizer*, and the proper parser. Not all parsers adopt this two-steps schema:

The lexer scans the text and find ‘4’, ‘3’, ‘7’ and then the space ‘ ‘. The job of the lexer is to recognize that the first characters constitute one token of type *NUM.* Then the lexer finds a ‘+’ symbol, which corresponds to a second token of type *PLUS*, and lastly it finds another token of type *NUM*.

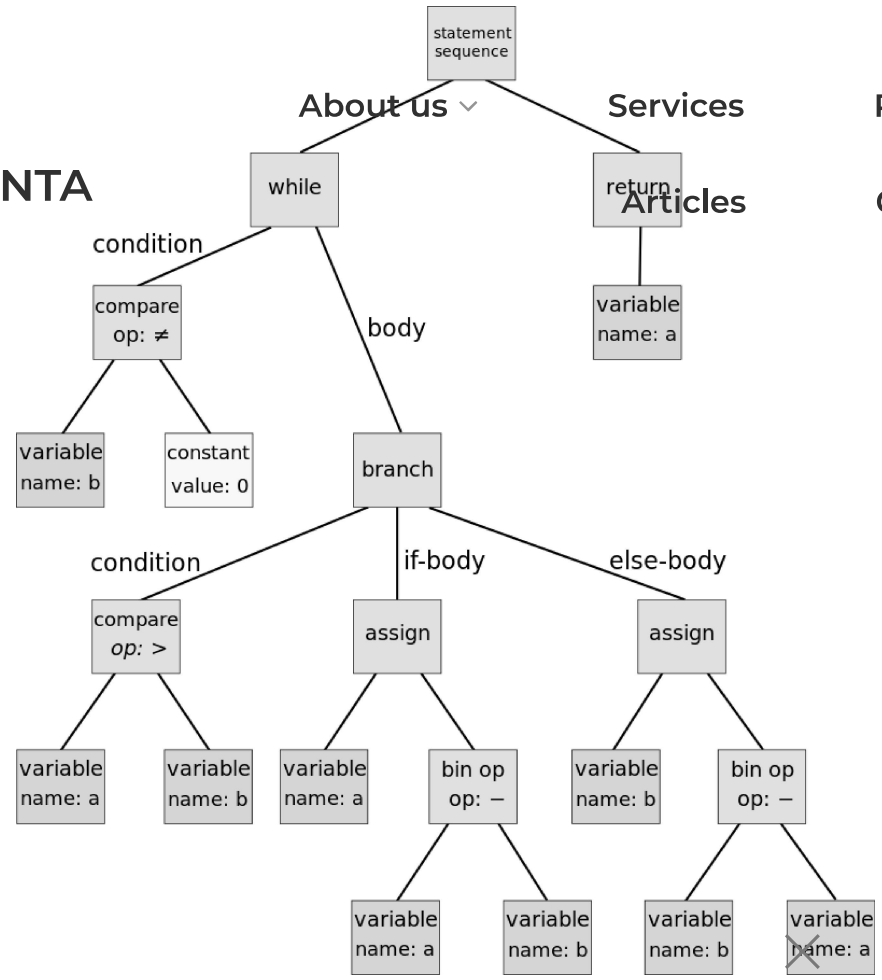
The parser will typically combine the tokens produced by the lexer and group them.

The definitions used by lexers or parser are called *rules* or *productions*. A lexer rule will specify that a sequence of digits correspond to a token of type *NUM*, while a parser rule will specify that a sequence of tokens of type *NUM, PLUS, NUM* corresponds to an expression.

There are two terms that are related and sometimes they are used interchangeably: parse tree and Abstract SyntaxTree (AST).

Conceptually they are very similar:

- they are both **trees**: there is a root representing the whole piece of code parsed. Then there are smaller subtrees representing portions of code that become smaller until single tokens appear in the tree
- the difference is the level of abstraction: the parse tree contains all the tokens which appeared in the program and possibly a set of intermediate rules. The AST instead is a polished version of the parse tree where the



In simple terms is a list of rules that define how each construct can be composed. For example, a rule for an if statement could specify that it must starts with the “if” keyword, followed by a left parenthesis, an expression, a right parenthesis and a statement.

A rule could reference other rules or token types. In the example of the if statement, the keyword “if”, the left and the right parenthesis were token types, while expression and statement were references to other rules.

The most used format to describe grammars is the **Backus-Naur Form (BNF)**, which also has many variants, including the **Extended Backus-Naur Form**. The Extended variant has the advantage of including a simple way to denote repetitions. A typical rule in a

# Left-recursive Rules



In the context of parsers an important feature is the support for left-recursive rules. This means that a rule could start with a reference to itself. This reference could be also indirect.

Consider for example arithmetic operations. An addition could be described as two expression(s) separated by the plus (+) symbol, but an expression could also contain other additions.

```
1.  addition       ::= expression '+' expression
2.  multiplication ::= expression '*' expression
3.  // an expression could be an addition or
```

# Types Of Languages And Grammars



We care mostly about two types of languages that can be parsed with a parser generator: *regular languages* and *context-free languages*. We could give you the formal definition according to the Chomsky hierarchy of languages, but it would not be that useful. Let's look at some practical aspects instead.

A regular language can be defined by a series of regular expressions, while a context-free one need something more. A simple rule of thumb is that if a grammar of a language has recursive elements it is not a regular language. For instance, as we said

# The Differences Between PEG and CFG



The main difference between PEG and CFG is that the ordering of choices is meaningful in PEG, but not in CFG. If there are many possible valid ways to parse an input, a CFG will be ambiguous and thus wrong. Instead with PEG the first applicable choice will be chosen, and this automatically solve some ambiguities.

Another differnce is that PEG use scannerless parsers: they do not need a separate lexer, or lexical analysis phase.

Traditionally both PEG and some CFG have been unable to deal with left-recursive rules, but some

The basic workflow of a parser generator tool is quite simple: you write a grammar that defines the language, or document, and you run the tool to generate a parser usable from your Java code.

The parser might produce the AST, that you may have to traverse yourself or you can traverse with additional ready-to-use classes, such Listeners or Visitors. Some tools instead offer the chance to embed code inside the grammar to be executed every time the specific rule is matched.

Usually you need a runtime library and/or program to use the generated parser.

## Regular (Lexer)

AnnoFlex is an annotation-based tool, but it does not use proper Java annotations. Instead it relies on custom Javadoc tags, that are interepreted by AnnoFlex. The reason for this design choice is to avoid the need to double escape regular expressions inside strings, which would have been necessary with strings inside annotations.

This is an interesting compromise: on one hand is easier to create regular expressions, but it looks a bit inelegant. It is pragmatism over formality and we think that most people would like it.

The following example shows a simple AnnoFlex program.



```
18.             MyClass scanner = new MyClass();
19.
```

```
20.             // this function and the
following will be generated by AnnoFlex
21.             scanner.setString("Hello
WorldGoodbye World");
22.
23.             String curToken =
scanner.getNextToken();
24.             while (curToken != null) {
25.                 // it prints I am arriving or
leaving: "Hello World"
26.                 // and I am arriving or
leaving: "Goodbye World"
27.
System.out.println(curToken+":
""+scanner.getMatchText()+"""");
28.                 curToken =
scanner.getNextToken();
29.             }
30.         }
31.  }
```

AnnoFlex requires Java 7 or later.



## JFlex

JFlex is a lexical analyzer (lexer) generator based upon deterministic finite automata (DFA). A JFlex lexer matches the input according to the defined grammar (called spec) and executes the corresponding action (embedded in the grammar).

It can be used as a standalone tool, but being a lexer generator is designed to work with parser generators: typically it is used with CUP or BYacc/J. It can also work with ANTLR.

The typical grammar (spec) is divided three parts,



```
11.
12.   [..]
```

```
13.
14.  LineTerminator = r|n|rn
15.
16.  %%
17.  // third section
18.
19.  /* keywords */
20.  <YYINITIAL> "abstract"          { return
     symbol(sym.ABSTRACT); }
21.  <YYINITIAL> "boolean"           { return
     symbol(sym.BOOLEAN); }
22.  <YYINITIAL> "break"             { return
     symbol(sym.BREAK); }
23.
24.  <STRING> {
25.    "                                  {
     yybegin(YYINITIAL);
26.                                       return
     symbol(sym.STRING_LITERAL,
27.                                              
     string.toString()); }
28.   [..]
29.  }
```





developed by the author and described in this paper.

Adaptive LL(*) Parsing: The Power of Dynamic Analysis (PDF)



It can output parsers in many languages. But the real added value of a vast community it is the large amount of grammars available. The version 4 supports direct left-recursive rules.

It provides two ways to walk the AST, instead of embedding actions in the grammar: visitors and listeners. The first one is suited when you have to manipulate or interact with the elements of the tree, while the second is useful when you just have to do something when a rule is matched.

The typical grammar is divided in two parts: lexer



written. If you are ready to become a professional

ANTLR developer, you can buy our video course to **Build professional parsers and languages using ANTLR**. *Note: the course is taught using Python.*



## APG

APG is a recursive-descent parser using a variation of **Augmented BNF**, that they call Superset Augmented BNF. ABNF is a particular variant of BNF designed to better support bidirectional communications protocol. APG also support additional operators, like syntactic predicates and custom user defined matching functions.

It can generate parsers in C/C++, Java e JavaScript

```
// example from a tutorial of the author of the tool available here
// https://www.sitepoint.com/alternative-to-regular-expressions/
phone-number = ["("] area-code sep office-code sep subscriber
area-code    = 3digit
; 3 digits
office-code  = 3digit
; 3 digits
subscriber   = 4digit
; 4 digits
sep          = *3(%d32-47 / %d58-126 / %d9) ; 0-3 ASCII non-digits
digit        = %d48-57
; 0-9
```



## BYACC/J



separated by '%%': DECLARATIONS, ACTIONS and

CODE. The second one contains the grammar rules and the third one the custom user code.



```
1.  // from the documentation
2.  %{
3.  import java.lang.Math;
4.  import java.io.*;
5.  import java.util.StringTokenizer;
6.  %}
7.
8.  /* YACC Declarations */
9.  %token NUM
10. %left '-' '+'
11. %left '*' '/'
12. %left NEG /* negation--unary minus */
13. %right '^' /* exponentiation */
14.
15. /* Grammar follows */
16. %%
```

recursive descent parser. Attributed grammar means that the rules, that are written in an EBNF variant, can be annotated in several ways to change the methods of the generated parser.

The scanner includes support for dealing with things like compiler directives, called pragmas. They can be ignored by the parser and handled by custom code. The scanner can also be suppressed and substituted with one built by hand.

Technically all the grammars must be LL(1), that is to say the parser must be able to choose the correct rule only looking one symbol ahead. But Coco/R provides several methods to bypass this limitation, including semantic checks, which are basically custom functions that must return a boolean value. The



```
14.     [..]
```

```
15.
16.   TOKENS
17.     ident     = letter { letter | digit
    }.
18.   [..]
19.   // ParserSpecification
20.   PRODUCTIONS
21.   // just a rule is shown
22.   IdentList =
23.     ident <out int x>  (. int n = 1; .)
24.     {',' ident         (. n++; .)
25.     }                  (.
    Console.WriteLine("n = " + n); .)
26.     .
27.   // end
28.   "END" ident '.'
```



Coco/R has a good documentation, with several examples grammars. It supports several languages including Java, C# and C++.



compatibility.

It requires Java 7 to generate the parser, but it can run on earlier versions.

A typical parser defined with annotations will look like this.

```
1.  // required import
2.  import org.yuanheng.cookcc.*;
3.
4.  @CookCCOption (lexerTable = "compressed",
    parserTable = "compressed")
5.  // the generated parser class will be a
    parent of the one you define
6.  // in this case it will be "Parser"
7.  public class Calculator extends Parser
8.  {
9.      // code
10.
11.     // a lexer rule
```

For the standard of parser generators, using Java annotations it is a peculiar choice. Compared to an alternative like ANTLR there is certainly a less clear division between the grammar and the actions. This could make the parser harder to maintain for complex languages. Also porting to another language could require a complete rewrite.

On the other hand this approach permit to mix grammar rules with the actions to perform when you match them. Furthermore it has the advantage of being integrated in the IDE of your choice, since it is just Java code.

## CUP



```
1.  // example from the documentation
```

```
2.  // CUP specification for a simple
    expression evaluator (w/ actions)
3.
4.  import java_cup.runtime.*;
5.
6.  /* Preliminaries to set up and use the
    scanner.  */
7.  init with {: scanner.init();
    :};
8.  scan with {: return scanner.next_token();
    :};
9.
10. /* Terminals (tokens returned by the
    scanner). */
11. terminal           SEMI, PLUS, MINUS,
    TIMES, DIVIDE, MOD;
12. terminal           UMINUS, LPAREN,
    RPAREN;
13. terminal Integer   NUMBER;
14.
15. /* Non-terminals */
16. non terminal            expr_list,
    expr_part;
```



## Grammatica

> *Grammatica is a C# and Java parser generator (compiler compiler). It reads a grammar file (in an EBNF format) and creates well- commented and readable C# or Java source code for the parser. It supports LL(k) grammars, automatic error recovery, readable error messages and a clean separation between the grammar and the source code.*

The description on the Grammatica website is itself a good representation of Grammatica: simple to use, well-documented, with a good amount of features. You can build a listener by subclassing the generated classes, but not a visitor. There is a good reference,



```
11.  [...]
12.  NUMBER                    = <<[0-9]+>>
```

```
13.  WHITESPACE                    = <<[
     tnr]+>> %ignore%
14.
15.  %productions%
16.
17.  Expression = Term [ExpressionTail] ;
18.
19.  ExpressionTail = "+" Expression
20.                 | "-" Expression ;
21.
22.  Term = Factor [TermTail] ;
23.
24.  [..]
25.
26.  Atom = NUMBER
27.       | IDENTIFIER ;
```

Compared to ANTLR the grammar file is much less clean and include a lot of Java source code



```
javacc_options
// "PARSER_BEGIN" "(" <IDENTIFIER> ")"
PARSER_BEGIN(SimpleParser)
public final class SimpleParser { // Standard parser class setup...

   public static void main(String args[]) {
      SimpleParser parser;
      java.io.InputStream input;

   }
   PARSER_END(SimpleParser)

   // the rules of the grammar
   // token rules
   TOKEN : {
```



projects, like JavaParser. This has left some quirks in

the documentation and usage. For instance, technically JavaCC itself does not build an AST, but it comes with a tool that does it, JTree, so for practical purposes it does.

There is a grammar repository, but it does not have many grammars in it. It requires Java 5 or later.

## ModelCC

> *ModelCC is a model-based parser generator that decouples language specification from language processing [..]. ModelCC receives a conceptual model as input, along with constraints that*



authors of the tools, but a limited documentation.

Nonetheless there are examples available, including the following model for a calculator partially shown here:



```
1.  public abstract class Expression implements IModel {
2.    public abstract double eval();
3.  }
4.
5.  [..]
6.
7.  public abstract class UnaryOperator implements IModel {
8.    public abstract double eval(Expression e);
9.  }
10.
11.  [..]
12.
29.  }
30.
```

```
31.     [..]
```



# SableCC

SableCC is a parser generator created for a thesis and with the aim to be easy to use and to offer a clean separation between grammar and Java code. Version 3 should also offer an included a ready-to-use way to walk the AST using a visitor. But that is all in theory because there is virtually no documentation and we have no idea how to use any of these things.

Also, a version 4 was started in 2015 and apparently lies abandoned.



```
2.     Letters ::= 'a'..'z', 'A'..'Z';
3.     Digits ::= '0'..'9';
```

```
4.  }
5.
6.  token {
7.     Space ::= [' ', #8, #9]*1;
8.     EOLN ::= [#10, #13];
9.     EOF ::= [#65535];
10.
11.          [..]
12.    Identifier ::= [Letters] [Letters,
    Digits]*;
13.  }
14.
15.  rules {
16.    Variable ::= "var", Identifier;
17.
18.    Element ::= Number | Identifier;
19.
20.    PlusExpression ::= Element, '+',
    Expression;
21.
22.    [..]
23.  }
```

*generated parsers have no runtime dependency on Canopy itself.*



It also provides easy access to the parse tree nodes.

A Canopy grammar has the neat feature of using actions annotation to use custom code in the parser. In practical terms. you just write the name of a function next to a rule and then you implement the function in your source code.

The Java file containing the action code.

```
1.  [..]
2.  import maps.Actions;
3.  [..]
4.
```

*generator with support for runtime grammar rules.*



Laja is a code generator and a parser generator and it is mainly designed to create external DSLs. This means that it have some peculiar features. With Laja you must specify not just the structure of the data, but also how the data should be mapped into Java structures. This structures are usually objects in a hierarchy or flat organization. In short, it makes very easy to parse data files, but it is less suitable for a generic programming language.

Laja options, like output directory or input file, are set in a configuration file.

A Laja grammar is divided in a rules section and the

## Mouse

> *Mouse is a tool to transcribe PEG into an executable parser written in Java.*

It does not use packrat and thus it uses less memory than the typical PEG parser (the manual explicitly compares Mouse to Rats!).

It does not have a grammar repository, but there are grammars for Java 6-8 and C.

A Mouse grammar is quite clean. To include custom code, a feature called semantic predicates, you do something similar to what you do in Canopy. You include a name in the grammar and then later, in a



Compiler). It is based on PEG, but it uses “additional

expressions and operators necessary for generating actual parsers". It supports left-recursive productions. It can automatically generate an AST.



It requires Java 6 or later.

The grammar can be quite clean, but you can embed custom code after each production.

```
1.  // example from Introduction to the Rats! Parser Generator
2.  // http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/rats-intro.pdf
3.  /* module intro */
4.  module Simple;
5.  option parser(SimpleParser);
6.
7.  /* productions for syntax analysis */
8.  public String program = e:expr EOF    {
```

> *Rekex is a new PEG parser generator for Java 17. It unifies grammar definition and AST construction in the most natural and intuitive way, leading to the simplest approach to writing parsers.*

Rekex is a new parser generator with a novel approach that flips writing a parser on its head. With traditional parser generators you write a grammar and then the generated parser produces a parse tree. One issue with this approach is that the parse tree is rarely what you want. So, you need to post-process the parse tree to create a data structure that fits your program. This can be a long process in itself. Particularly if you are dealing with a large grammar



even a few examples in their repository and

explanation of subprojects used by the library. This is crucial given the novel approach of this library, so that every user can understand if it is a good fit for them.



Okay, now we can see what a Rekex parser looks like.

```
1.  [..]
2.
3.  public interface ExampleParser_Calculator1
4.  {
5.      // in this example:
6.      //   - whitespaces: optional Space/Tab between tokens
7.      //   - token datatypes are extracted
8.      //   - eval() defined inside datatypes
9.
27.
        // tokens --------------------------
```

```
28. ----------------------------
29.
30.     enum Op1
31.     {
32.         @Word("+") plus,
33.         @Word("-") minus;
34.
35.         public int eval(int x, int y){
36.             return this==plus ? x+y : x-
    y;
37.         }
38.     }
39.     enum Op2
40.     {
41.         @Word({"/","÷"}) div,
42.         @Word({"*","×"}) mul,
43.         @Word("") mul_implicit; // a b ==
    a * b
44.
45.         public int eval(int x, int y){
46.             return this==div ? x/y : x*y;
47.         }
```



```
66.     // [..] other records with eval
    methods
```

```
67.
68.         // test ----------
            ----------
69.
70.         public static PegParser<Input>
            parser()
71.         {
72.             return PegParser.of(Input.class);
73.         }
74.         public static Function<Input,Integer>
            eval()
75.         {
76.             return (input)->input.e0.eval();
77.         }
78.     }
```

In practice, you write the parser using a well-defined structure and conventions (e.g., `eval` classes) in Java. In other words, the grammar is Java (17) code but a

In practice this means that they are very useful for all the little parsing problems you find. If the typical developer encounters a problem, that is too complex for a simple regular expression, these libraries are usually the solution. In short, if you need to build a parser, but you do not actually want to, a parser combinator may be your best option.

# Jparsec

Jparsec is the port of the parsec library of Haskell.

Parser combinators are usually used in one phase, that is to say they are without lexer. This is simply because it can quickly become too complex to



```
7.     private static final Terminals
```

```
OPERATORS =
        Terminals.operators("+", "-", "*",
"/", "(", ")");

    [..]

    static final Parser<?> TOKENIZER =
Parsers.or(Terminals.DecimalLiteral.TOKENI
ZER, OPERATORS.tokenizer());

  [..]

    static Parser<Double>
calculator(Parser<Double> atom) {
      Parser.Reference<Double> ref =
Parser.newReference();
      Parser<Double> unit =
ref.lazy().between(term("("),
term(")")).or(atom);
      Parser<Double> parser = new
OperatorTable<Double>()
          .infixl(op("+", (l, r) -> l + r),
```



beginning of 2017).

> *Parboiled provides a recursive descent PEG parser implementation that operates on PEG rules you specify.*

The objective of parboiled is to provide an easy to use and understand way to create small DSLs in Java. It puts itself in the space between a simple bunch of regular expressions and an industrial-strength parser generator like ANTLR. A parboiled grammar can include actions with custom code, included directly into the grammar code or through an interface.



```
19.
20.
```

```
21.       [..]
22.       public Rule OperatorRule(Rule subRule,
Rule operatorRule) {
23.           Var<Character> op = new
Var<Character>();
24.           return Sequence(
25.                   subRule,
26.                   ZeroOrMore(
27.                           operatorRule,
op.set(matchedChar()),
28.                           subRule,
29.                           push(new
CalcNode(op.get(), pop(1), pop()))
30.                   )
31.           );
32.       }
33.
34.       [..]
35.
36.       public Rule Number() {
37.           return Sequence(
38.                   Sequence(
```

Parboiled works a bit like a cross between a parser combinator and a parser generator. You create rules in code, with ready-to-use methods like `Sequence` or `Optional`, just like a parser combinator. However, the end result is parser class that you are supposed to use like a generated parser. Parboiled is not suited to create individually used rules, i.e., to parse bits and pieces, the way a parser combinator can. You use it to parse a coherent language.

It does not build an AST for you, but it provides a parse tree and some classes to make it easier to build it. That is because its authors maintain that the AST is heavily dependent on your exact project needs, so they prefer to offer an “open and flexible approach”. It



*scannerless parsing, parser*

*combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically.*



PetitParser is a also between a parser combinator and a traditional parser generator. All the information is written in the source code, but the source code is divided in two files. In one file you define the grammar, while in the other one you define the actions corresponding to the various elements. The idea is that it should allow you to dynamically redefine grammars. While it is smartly engineered, it is debatable if it is also smartly designed.

Given that departs from the usual design of a parser



```
7.   // setup code not shown
```

```
8.
9.      public JsonGrammarDefinition() {
10.         def("start", ref("value").end());
11.
12.         def("array", of('[').trim()
13.                 .seq(ref("elements").optional())
14.                 .seq(of(']').trim()));
15.         def("elements",
    ref("value").separatedBy(of(',').trim()));
16.         def("members",
    ref("pair").separatedBy(of(',').trim()));
17.
18.         [..]
19.
20.         def("trueToken",
    of("true").flatten().trim());
21.         def("falseToken",
    of("false").flatten().trim());
22.         def("nullToken",
    of("null").flatten().trim());
23.         def("stringToken",
    ref("stringPrimitive").flatten().trim());
24.         def("numberToken",
    Functions.withoutSeparators());
9.          action("members",
```

```
    Functions.withoutSeparators());
10.       action("array", new
    Function<List<List<?>>, List<?>>() {
11.         @Override
12.         public List<?> apply(List<List<?>>
    input) {
13.           return input.get(1) != null ?
    input.get(1) : new ArrayList<>();
14.         }
15.       });
16.
17.       [..]
18.     }
19.   }
```

There is a version written in Java, but there are also versions in Smalltalk, Dart, PHP, and TypeScript.

The documentation is lacking, but there are example grammars available.



framework of functional Java libraries. The library

wants to provide a simple internal Domain Specific Language to express grammar languages.

The library is simple, but offers everything necessary to create simple parsers.

```
1.  Parser<Chr, Integer> sum =
    intr.andL(chr('+')).and(intr).map(Integer:
    :sum);
```

In this example from the documentation you can see how it is possible to combine the parsers for integers (`intr`) and the one for characters (`chr`) to parse a simple sum expression. The expression is also evaluated using the `map` function to call the normal



recommended by Danny van Bruggen, the maintainer

of JavaParser, so you know it is good.



# Java Libraries That Parse Java: JavaParser

There is one special case that requires a specific comment: the case in which you want to parse Java code in Java. In this case we suggest using a library named JavaParser. Incidentally we heavily contribute to JavaParser, but this is not the only reason why we suggest it. The fact is that JavaParser is a project with tens of contributors and thousands of users, so it is pretty robust.

# Summary



Parsing in Java is a broad topic and the world of parsers is a bit different from the usual world of programmers. You will find the best tools coming directly from academia, which is typically not the case with software. Some tools and libraries have been started for a thesis or a research project. The upside is that tools tend to be easily and freely available. The downside is that some authors prefer to have a good explanation of the theory behind what their tools do, rather than a good documentation on how to use them. Also, some tools end up being abandoned as the original authors finish their master or their PhD.



most technically correct solution might not be ideal in

real life with all its constraints. But we have searched and tried many similar tools in our work and something like this article would have helped us save some time. So we wanted to share what we have learned on the best options for parsing in Java.

*We would like to thank Stefan Czaska for having informed us of AnnoFlex.*

*We would like to thank Danny van Bruggen for having informed us of funcj.*

*We would like to thank Zhong Yu for having informed us of rekex.*

Parsing

Software Development

STRUMENTA

**Boot Migrator**

27 September 2022

14 September 2022



**STRUMENTA**
Tools to solve complex problems

Privacy Policy
P.IVA 11817320010
Company Information

Strumenta
Strumenta
Federico Tomassetti
Federico Tomassetti