Parsing in Java

Tools and Libraries

Learn more at https://tomassetti.me

If you need to parse a language, or document, 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

The first option is the best for well-known and supported languages, like XML or HTML. A good library usually includes also API to programmatically build and modify documents in that language. This is typically more of what you get from a basic parser. The problem is that such libraries are not so common, and they support only the most common languages. In other cases, you are out of luck.

Building Your Own Custom Parser by Hand

You may need to pick the second option if you have particular needs. Both in the sense that the language you need to parse cannot be parsed with traditional 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: paragraphs indented and in italics describing a program comes from the respective documentation

Table Of Contents

Introduction to Parsing

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 and not all types of parser generators are suitable for all kinds of languages. That is why we have prepared a list of the best known of them, with a short introduction for each of them. We are also concentrating each time on one target language. This also means that (usually) the parser itself will be written in that target language.

To list all possible tools and libraries parser for all languages would be kind of interesting, but not that useful. That is because there will be simple too many options and we would all get lost in them. By concentrating on one programming language we can provide an apples-to-apples comparison and help you choose one option for your project.

Useful Things to Know About Parsers

To make sure that this 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: some parsers do not depend on a lexer. They are called *scannerless parsers*.

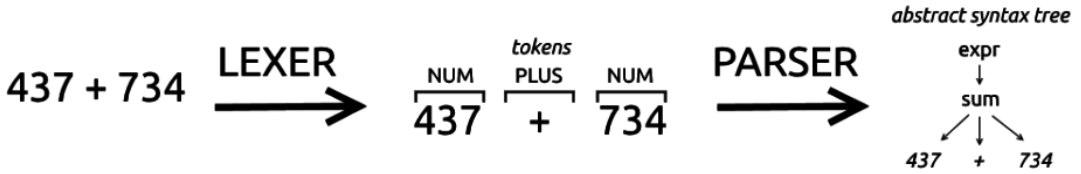
A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser scans the tokens and produces the parsing result.

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

```
437 + 734
```

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.

Scannerless parsers are different because they process directly the original text, instead of processing a list of tokens produced by a lexer.

It is now typical to find suites that can generate both a lexer and parser. In the past it was instead more common to combine two different tools: one to produce the lexer and one to produce the parser. This was for example the case of the venerable lex & yacc couple: lex produced the lexer, while yacc produced the parser.

Parse Tree and Abstract Syntax Tree

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

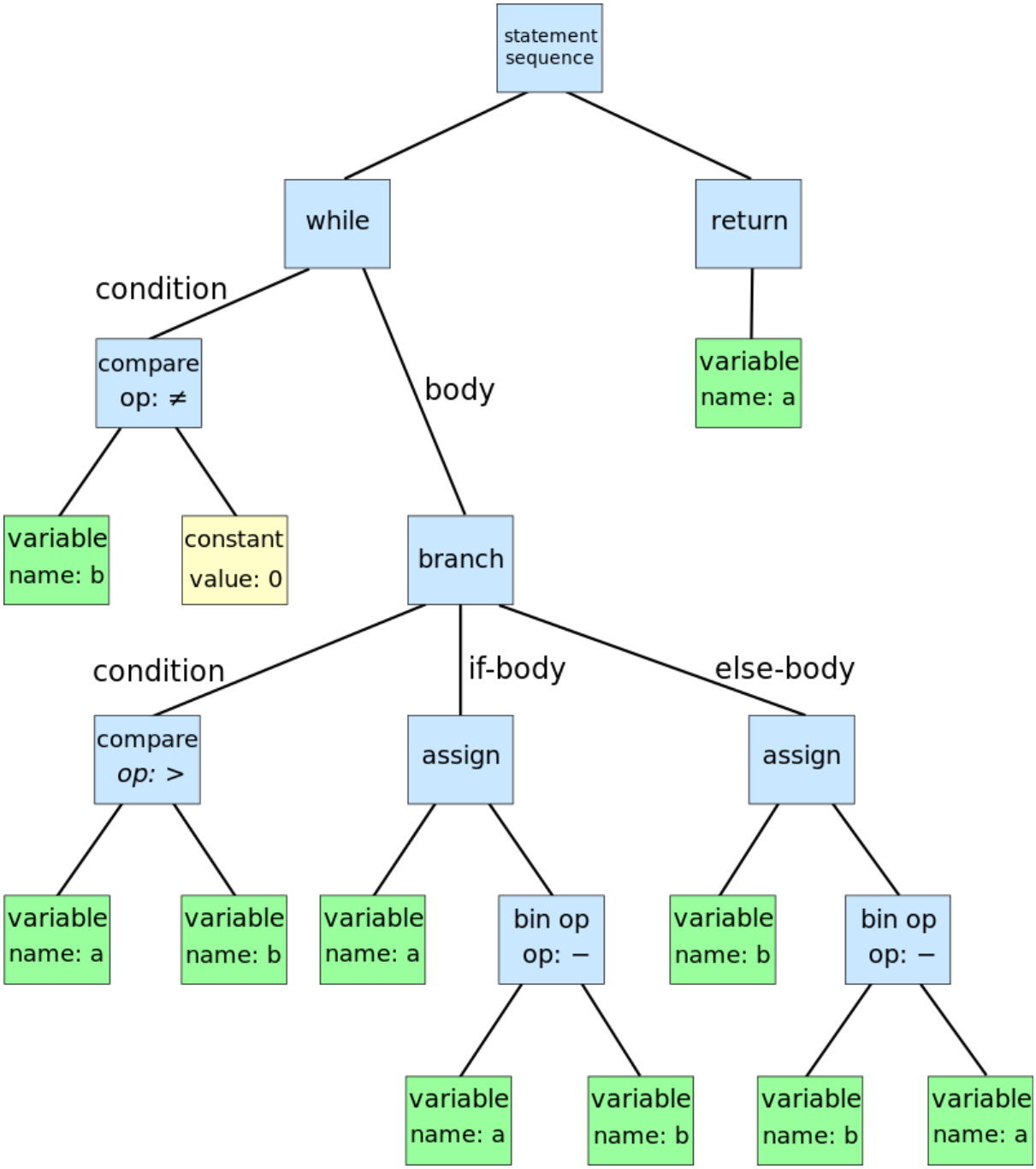
Conceptually they are very similar:

- 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 information that could be derived or is not important to understand the piece of code is removed

In the AST some information is lost, for instance comments and grouping symbols (parentheses) are not represented. Things like comments are superfluous for a program and grouping symbols are implicitly defined by the structure of the tree.

A parse tree is a representation of the code closer to the concrete syntax. It shows many details of the implementation of the parser. For instance, usually a rule corresponds to the type of a node. A parse tree is usually transformed in an AST by the user, possibly with some help from the parser generator.

A graphical representation of an AST looks like this.



Sometimes you may want to start producing a parse tree and then derive from it an AST. This can make sense because the parse tree is easier to produce for the parser (it is a direct representation of the parsing process) but the AST is simpler and easier to process by the following steps. By following steps we mean all the operations that you may want to perform on the tree: code validation, interpretation, compilation, etc.

Grammar

A grammar is a formal description of a language that can be used to recognize its structure.

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 Backus-Naur grammar looks like this:

```
<symbol> ::= __expression__
```

The <simbol> is usually nonterminal, which means that it can be replaced by the group of elements on the right, __expression__. The element __expression__ could contains other nonterminal symbols or terminal ones. Terminal symbols are simply the ones that do not appear as a <symbol> anywhere in the grammar. A typical example of a terminal symbol is a string of characters, like "class".

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.

```
addition       ::= expression '+' expression
multiplication ::= expression '*' expression
// an expression could be an addition or a multiplication or a number
expression     ::= addition | multiplication |// a number
```

This description also match multiple additions like 5 + 4 + 3. That is because it can be interpreted as expression (5) ('+') expression(4+3). And then 4 + 3 itself can be divided in its two components.

The problem is that this kind of rules may not be used with some parser generators. The alternative is a long chain of expressions that takes care also of the precedence of operators.

Some parser generators support direct left-recursive rules, but not indirect one.

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 language*s. 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 elsewhere, HTML is not a regular language. In fact, most programming languages are context-free languages.

Usually to a kind of language correspond the same kind of grammar. That is to say there are regular grammars and context-free grammars that corresponds respectively to regular and context-free languages. But to complicate matters, there is a relatively new (created in 2004) kind of grammar, called Parsing Expression Grammar (PEG). These grammars are as powerful as Context-free grammars, but according to their authors they describe programming languages more naturally.

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 difference 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 tools have found workarounds for this. Either by modifying the basic parsing algorithm, or by having the tool automatically rewrite a left-recursive rule in a non-recursive way. Either of these ways has downsides: either by making the generated parser less intelligible or by worsen its performance. However, in practical terms, the advantages of easier and quicker development outweigh the drawbacks.

Java

Parser Generators

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)

Tools that analyze regular languages are typically called lexers.

We are not going to talk about it, because it is very basic, but Java includes a library to parse data with numbers and simple patterns: java.util.Scanner. It could be defined as a smart library to read streams of data. It might be worth to check it out if you are in need of quickly parse some data.

AnnoFlex

An annotation-based code generator for lexical scanners

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.

```
public class MyClass {
    // notice that Hello/Goodbye/World are not inside a string
    // so we add a space this way [ ]
    // However you could also use a string
    /**
     * @expr (Hello | Goodbye)[ ]World
     */
    String greeting()
    {
        return "I am arriving or leaving";
    }
```

```
    //%%LEX-MAIN-START%%
    // This is the area in which AnnoFlex will put the generated code
    //%%LEX-MAIN-END%%

    public static void main(String args[]) {
        MyClass scanner = new MyClass();

        // this function and the following will be generated by AnnoFlex
        scanner.setString("Hello WorldGoodbye World");

        String curToken = scanner.getNextToken();
        while (curToken != null) {
            // it prints I am arriving or leaving: "Hello World"
            // and I am arriving or leaving: "Goodbye World"
            System.out.println(curToken+": \""+scanner.getMatchText()+"\"");
            curToken = scanner.getNextToken();
        }
    }
}
```

There is an integration for IDEs, but only up to a point. The tool comes with a script to easily call it from an IDE, but since the tool uses non-standard Javadoc tags the IDE itself might complain about errors in the Javadoc comments.

The support for regular expression is complete and include everything you need: from quantifiers (e.g., *) to POSIX character classes (e.g., [[:alnum:]]). The syntax also supports lookahead tokens (i.e., you can match an expression based on what follows it) and macros.

The documentation explains the syntax in details. At the moment it is available as a PDF manual, but the author is working also on a website. There are a few examples, that work as a tutorial.

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, separated by '%%':

1. usercode, that will be included in the generated class,
2. options/macros,
3. and finally the lexer rules.

```
// taken from the documentation
/* JFlex example: partial Java language lexer specification */
```

```
import java_cup.runtime.*;

%%
// second section

%class Lexer
%unicode
%cup

[..]

LineTerminator = \r|\n|\r\n

%%
// third section

/* keywords */
<YYINITIAL> "abstract"          { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"           { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break"             { return symbol(sym.BREAK); }

<STRING> {
  \"                            { yybegin(YYINITIAL);
                                  return symbol(sym.STRING_LITERAL,
                                  string.toString()); }

 [..]
}

/* error fallback */
[^]                             { throw new Error("Illegal character <"+
                                                  yytext()+">"); }
```

Context Free

Let's see the tools that generate Context Free parsers.

ANTLR

ANTLR is probably the most used parser generator for Java. ANTLR is based on an new LL algorithm 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 rules and parser rules. The division is implicit, since all the rules starting with an uppercase letter are lexer rules, while the ones starting with a lowercase letter are parser rules. Alternatively lexer and parser grammars can be defined in separate files.

```
grammar simple;

basic   : NAME ':' NAME ;

NAME    : [a-zA-Z]* ;

COMMENT : '/*' .*? '*/' -> skip ;
```

If you are interested in ANTLR you can look into this giant ANTLR tutorial we have written.

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. Support for the last language seems superior and more up to date: it has a few more features and seems more updated. In fact the documentation says it is designed to have the look and feel of JavaScript RegExp.

> *Because it is based on ABNF, it is especially well suited to parsing the languages of many Internet technical specifications and, in fact, is the parser of choice for a number of large Telecom companies.*

An APG grammar is very clean and easy to understand.

```
// 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

BYACC is Yacc that generates Java code. That is the whole idea and it defines its advantages and disadvantages. It is well known, it allows easier conversion of a Yacc and C program to a Java program. Although you obviously still need to convert all the C code embedded in semantic actions into Java code. Another advantage it is that you do not need a separate runtime, the generated parser it is all you need.

On the other hand, it is old and the parsing world has made many improvements. If you are an experienced Yacc developer with a code base to upgrade it is a good choice, otherwise there are many more modern alternatives you should consider.

The typical grammar is divided in three sections, separated by '%%': DECLARATIONS, ACTIONS and CODE. The second one contains the grammar rules and the third one the custom user code.

```
// from the documentation
%{
import java.lang.Math;
import java.io.*;
import java.util.StringTokenizer;
%}

/* YACC Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG /* negation--unary minus */
%right '^' /* exponentiation */

/* Grammar follows */
%%
input: /* empty string */
 | input line
 ;

line: '\n'
 | exp '\n' { System.out.println(" " + $1.dval + " "); }
 ;
%%
public static void main(String args[])
{
 Parser par = new Parser(false);
  [..]
}
```

Coco/R

Coco/R is a compiler generator that takes an attributed grammar and generates a scanner and a 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 manual also provides some suggestions for refactoring your code to respect this limitation.

A Coco/R grammar looks like this.

```
[Imports]
// ident is the name of the grammar
"COMPILER" ident
// this includes arbitrary fields and method in the target language (eg. Java)
[GlobalFieldsAndMethods]
// ScannerSpecification
CHARACTERS
  [..]
  zero          = '0'.
  zeroToThree   = zero + "123" .
  octalDigit    = zero + "1234567" .
  nonZeroDigit  = "123456789".
  digit         = '0' + nonZeroDigit .
  [..]

TOKENS
  ident         = letter { letter | digit }.
[..]
// ParserSpecification
PRODUCTIONS
// just a rule is shown
IdentList =
  ident <out int x>  (. int n = 1; .)
  {',' ident         (. n++; .)
  }                  (. Console.WriteLine("n = " + n); .)
  .
// end
"END" ident '.'
```

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

CookCC

CookCC is a LALR (1) parser generator written in Java. Grammars can be specified in three different ways:

- in Yacc format: it can read grammar defined for Yacc
- in its own XML format
- in Java code, by using specific annotations

A unique feature is that it can also output a Yacc grammar. This can be useful if you need to interact with a tool that support a Yacc grammar. Like some old C program with which you must maintain 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.

```
// required import
import org.yuanheng.cookcc.*;

@CookCCOption (lexerTable = "compressed", parserTable = "compressed")
// the generated parser class will be a parent of the one you define
// in this case it will be "Parser"
public class Calculator extends Parser
{
    // code

    // a lexer rule
    @Shortcuts ( shortcuts = {
    @Shortcut (name="nonws", pattern="[^ \\t\\n]"),
    @Shortcut (name="ws", pattern="[ \\t]")
    })
    @Lex (pattern="{nonws}+", state="INITIAL")
    void matchWord ()
    {
        m_cc += yyLength ();
        ++m_wc;
    }

    // a typical parser rules
    @Rule (lhs = "stmt", rhs = "SEMICOLON")
    protected Node parseStmt ()
    {
        return new SemiColonNode ();
    }
}
```

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 permits 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

CUP is the acronym of Construction of Useful Parsers and it is LALR parser generator for Java. It just generates the proper parser part, but it is well suited to work with JFlex. Although obviously you can also build a lexer by hand to work with CUP. The grammar has a syntax similar to Yacc and it allows to embed code for each rule.

It can automatically generate a parse tree, but not an AST.

It also has an Eclipse plugin to aid you in the creation of a grammar, so effectively it has its own IDE.

The typical grammar is similar to YACC.

```
// example from the documentation
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal           SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal           UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part
              |
              expr_part;

expr_part ::= expr:e
             {: System.out.println("= " + e); :}
              SEMI
             ;
[..]
```

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 number of features. You can build a listener by subclassing the generated classes, but not a visitor. There is a good reference, but not many examples.

A typical grammar of Grammatica is divided in three sections: header, tokens and productions. It is also clean, almost as much as an ANTLR one. It is also based on a similar Extended BNF, although the format is slightly different.

```
%header%

GRAMMARTYPE = "LL"

[..]

%tokens%

ADD                             = "+"
SUB                             = "-"
[..]
NUMBER                          = <<[0-9]+>>
WHITESPACE                      = <<[ \t\n\r]+>> %ignore%

%productions%

Expression = Term [ExpressionTail] ;

ExpressionTail = "+" Expression
               | "-" Expression ;

Term = Factor [TermTail] ;

[..]

Atom = NUMBER
     | IDENTIFIER ;
```

Jacc

Jacc is similar to BYACC/J, except that is written in Java and thus it can run wherever your program can run. As a rule of thumb, it is developed as a more modern version of Yacc. The author describes small improvements in areas like error messages, modularity and debug support.

If you know Yacc and you do not have any code base to upgrade, it might be a great choice.

JavaCC

JavaCC is the other widely used parser generator for Java. The grammar file contains actions and all the custom code needed by your parser.

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  :
{
  < #DIGIT : ["0"-"9"] >
| < #LETTER : ["A"-"Z","a"-"z"] >
| < IDENT : <LETTER> (<LETTER> | <DIGIT>)* >
[..]
}

SKIP : {  " " | "\t" | "\n" | "\r" }

// parser rules

[..]

void IdentDef() : {}
{
  <IDENT> ("*" | "-")?
}
```

Thanks to its long history it is used in important 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 annotate it.

In practical terms you define a model of your language, that works as a grammar, in Java, using annotations. Then you feed to ModelCC the model you have created to obtain a parser.

With ModelCC you define your language in a way that is independent from the parsing algorithm used. Instead, it should the best conceptual representation of the language. Although, under the hood, it uses a traditional parsing algorithm. So, the grammar *per se* use a form that is independent from any parsing algorithm, but ModelCC does not use magic and produces a normal parser.

There is a clear description of the intentions of the authors of the tools, but a limited documentation. Nonetheless there are examples available, including the following model for a calculator partially shown here.

```
public abstract class Expression implements IModel {
  public abstract double eval();
}

[..]

public abstract class UnaryOperator implements IModel {
  public abstract double eval(Expression e);
}

[..]

@Pattern(regExp="-")
public class MinusOperator extends UnaryOperator implements IModel {
  @Override public double eval(Expression e) { return -e.eval(); }
}

@Associativity(AssociativityType.LEFT_TO_RIGHT)
public abstract class BinaryOperator implements IModel {
  public abstract double eval(Expression e1,Expression e2);
}

[..]

@Priority(value=2)
@Pattern(regExp="-")
public class SubtractionOperator extends BinaryOperator implements IModel {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()-
e2.eval(); }
}

[..]
```

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.

UrchinCC

Urchin(CC) is a parser generator that allows you to define a grammar, called Urchin parser definition. Then you generate a Java parser from it. Urchin also generate a visitor from the UPD.

There is an exhaustive tutorial that is also used to explain how Urchin works and its limitations, but the manual is limited.

A UPD is divided in three sections: terminals, token and rules.

```
terminals {
	Letters ::= 'a'..'z', 'A'..'Z';
	Digits ::= '0'..'9';
}

token {
	Space ::= [' ', #8, #9]*1;
	EOLN ::= [#10, #13];
	EOF ::= [#65535];

	 [..]
	Identifier ::= [Letters] [Letters, Digits]*;
}

rules {
	Variable ::= "var", Identifier;

	Element ::= Number | Identifier;

	PlusExpression ::= Element, '+', Expression;

	[..]
}
```

PEG

After the CFG parsers is time to see the PEG parsers available in Java.

Canopy

> Canopy is a parser compiler targeting Java, JavaScript, Python and Ruby. It takes a file describing a parsing expression grammar and compiles it into a parser module in the target language. The 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 actions are prepended by %
grammar Maps
  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

The Java file containing the action code.

```
[..]
import maps.Actions;
[..]

class MapsActions implements Actions {
    public Pair make_map(String input, int start, int end, List<TreeNode>
elements) {
        Text string = (Text)elements.get(1);
        Array array = (Array)elements.get(3);
        return new Pair(string.string, array.list);
    }

    [..]
}
```

Laja

> *Laja is a two-phase scannerless, top-down, backtracking parser 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 has 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. These 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 data mapping section. It looks like this.

```
// this example is from the documentation
grammar example {
  s = [" "]+;
  newline = "\r\n" | "\n";

  letter = "a".."z";
  digit = "0".."9";
  label = letter [digit|letter]+;
  row = label ":" s [!(newline|END)+]:value [newline];
  example = row+;

   Row row.setLabel(String label);
   row.setValue(String value);

   Example example.addRow(Row row);
}
```

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 Java file, you actually write the custom code.

```
// example from the manual
// http://mousepeg.sourceforge.net/Manual.pdf
// the semantics are between {}
Sum     = Space Sign Number (AddOp Number)* !_ {sum} ;
Number  = Digits Space {number} ;
Sign    = ("-" Space)? ;
AddOp   = [-+] Space ;
Digits  = [0-9]+ ;
Space   = " "* ;
```

Rats!

Rats! is a parser generator part of xtc (eXTensible 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.

```
// example from Introduction to the Rats! Parser Generator
// http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/rats-intro.pdf
/* module intro */
module Simple;
option parser(SimpleParser);

/* productions for syntax analysis */
public String program = e:expr EOF    { yyValue = e; } ;
String expr  = t:term r:rest          { yyValue = t + r; } ;
String rest = PLUS  t:term r:rest     { yyValue = t + "+" + r; }
              / MINUS t:term r:rest   { yyValue = t + "-" + r; }
              / /*empty*/             { yyValue = ""; } ;
String term  = d:DIGIT                { yyValue = d; } ;

/* productions for lexical analysis */
void    PLUS  = "+";
void    MINUS = "-";
String DIGIT = [0-9];
void    EOF   = ! ;
```

Parser Combinators

They allow you to create a parser simply with Java code, by combining different pattern matching functions, that are equivalent to grammar rules. They are generally considered best suited for simpler parsing needs. Given they are just Java libraries you can easily introduce them into your project: you do not need any specific generation step and you can write all of your code in your favorite Java editor. Their main advantage is the possibility of being integrated in your traditional workflow and IDE.

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 don't 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 manage all the combinators chains directly in the code. Having said that, jparsec has a special class to support lexical analysis.

It does not support left-recursive rules, but it provides a special class for the most common use case: managing the precedence of operators.

A typical parser written with jparsec is similar to this one.

```
// from the documentation
public class Calculator {

  static final Parser<Double> NUMBER =
      Terminals.DecimalLiteral.PARSER.map(Double::valueOf);

  private static final Terminals OPERATORS =
      Terminals.operators("+", "-", "*", "/", "(", ")");

  [..]

  static final Parser<?> TOKENIZER =
      Parsers.or(Terminals.DecimalLiteral.TOKENIZER, 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), 10)
        .infixl(op("-", (l, r) -> l - r), 10)
        .infixl(Parsers.or(term("*"), WHITESPACE_MUL).retn((l, r) -> l * r), 20)
        .infixl(op("/", (l, r) -> l / r), 20)
        .prefix(op("-", v -> -v), 30)
        .build(unit);
```

```
    ref.set(parser);
    return parser;
  }

  public static final Parser<Double> CALCULATOR =
      calculator(NUMBER).from(TOKENIZER, IGNORED);
}
```

Parboiled

> *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 put 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.

```
// example parser from the parboiled repository
// CalculatorParser4.java

package org.parboiled.examples.calculators;

[..]

@BuildParseTree
public class CalculatorParser4 extends CalculatorParser<CalcNode> {

    @Override
    public Rule InputLine() {
        return Sequence(Expression(), EOI);
    }

    public Rule Expression() {
        return OperatorRule(Term(), FirstOf("+ ", "- "));
    }

    [..]

    public Rule OperatorRule(Rule subRule, Rule operatorRule) {
        Var<Character> op = new Var<Character>();
        return Sequence(
                subRule,
                ZeroOrMore(
                        operatorRule, op.set(matchedChar()),
                        subRule,
                        push(new CalcNode(op.get(), pop(1), pop()))
                )
        );
    }
```

```
    [..]

    public Rule Number() {
        return Sequence(
                Sequence(
                        Optional(Ch('-')),
                        OneOrMore(Digit()),
                        Optional(Ch('.'), OneOrMore(Digit()))
                ),
                // the action uses a default string in case it is run during error
recovery (resynchronization)
                push(new CalcNode(Double.parseDouble(matchOrDefault("0")))),
                WhiteSpace()
        );
    }

    //**************** MAIN ****************

    public static void main(String[] args) {
        main(CalculatorParser4.class);
    }
}
```

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 sound quite appropriate to the project objective and one of our readers find the approach better than a straight AST.

The documentation is very good, it explains features, shows example, compare the ideas behind parboiled with the other options. There are some example grammars in the repository, including one for Java.

It is used by several projects, including important ones like neo4j.

PetitParser

> *PetitParser combines ideas from 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 cross 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. You can see that the example JSON grammar it is more lengthy than one expects it to be.

An excerpt from the example grammar file for JSON.

```
package org.petitparser.grammar.json;

[..]

public class JsonGrammarDefinition extends GrammarDefinition {

// setup code not shown

  public JsonGrammarDefinition() {
    def("start", ref("value").end());

    def("array", of('[').trim()
        .seq(ref("elements").optional())
        .seq(of(']').trim()));
    def("elements", ref("value").separatedBy(of(',').trim()));
    def("members", ref("pair").separatedBy(of(',').trim()));

    [..]

    def("trueToken", of("true").flatten().trim());
    def("falseToken", of("false").flatten().trim());
    def("nullToken", of("null").flatten().trim());
    def("stringToken", ref("stringPrimitive").flatten().trim());
    def("numberToken", ref("numberPrimitive").flatten().trim());

    [..]
  }
}
```

An excerpt from the example parser definiton file (that defines the actions for the rules) for JSON .

```
package org.petitparser.grammar.json;

import org.petitparser.utils.Functions;

public class JsonParserDefinition extends JsonGrammarDefinition {

  public JsonParserDefinition() {
    action("elements", Functions.withoutSeparators());
    action("members", Functions.withoutSeparators());
    action("array", new Function<List<List<?>>, List<?>>() {
      @Override
      public List<?> apply(List<List<?>> input) {
        return input.get(1) != null ? input.get(1) : new ArrayList<>();
      }
    });

    [..]
  }
}
```

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.

Java Libraries That Parse Java: JavaParser

There is one special case that requires some more comments: the case in which you want to parse Java code in Java. In this case we have to suggest to use 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.

A quick list of features:

- it supports all versions of Java from 1 to 9
- it supports lexical preservation and pretty printing: it means you can parse Java code, modify it and printing it back either with the original formatting or pretty printed
- it can be used with JavaSymbolSolver, which gives you symbol resolution. I.e., it understands which methods are invoked, to which declarations references are linked to, it calculates the type of expressions, etc.

Convinced? You want still to write your own Java parser for Java?

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.

We tend to use parser generators quite a lot: ANTLR is our favorite one and we use JavaCC extensively in our work on JavaParser. We do not use parser combinators very much. It is not because they are bad, they have their uses and in fact we wrote an article about one in C#. But for the problems we deal with, they typically lead to less maintainable code. However, they could be easier to start with so you may want to consider those. Especially if until now you have hacked something terrible using regular expressions and an half baked parser written by hand.

We cannot really say to you definitely what software you should use. What it is best for a user might not be the best for somebody else. And we all know that the 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.