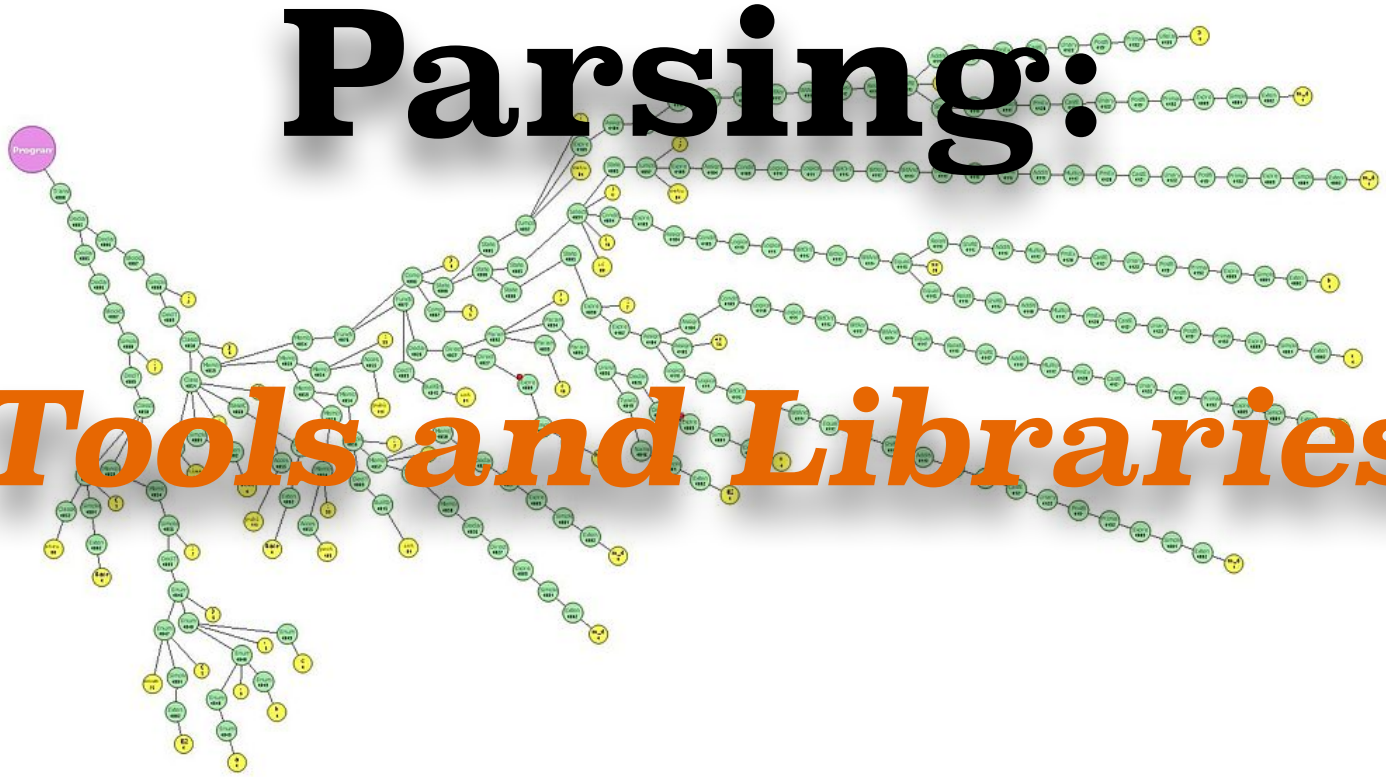
Parsing:



Tools and Libraries

Specific resources for Java, Python, C#, and JavaScript

FEDERICO TOMASSETTI

Software Architect

Learn more at 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

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*.

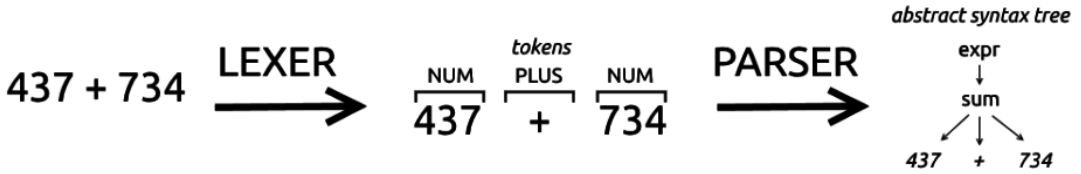
A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser scans the tokens and produces the parsing result.

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).

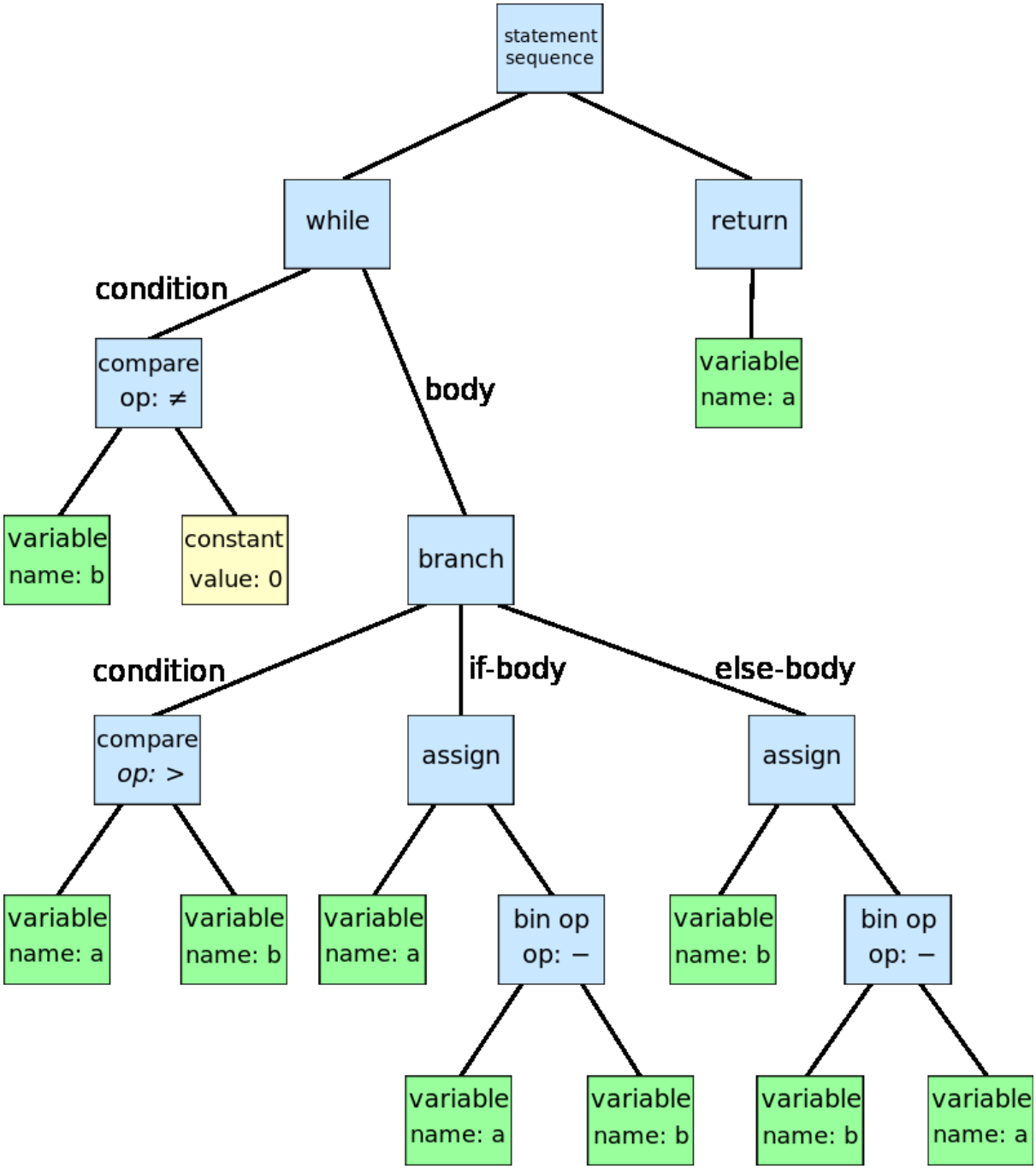
Conceptually they are very similar:

- 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 matches 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 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 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 detail. 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 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.

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.

C#

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 C# 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.

Gardens Point LEX

Gardens Point LEX (GPLEX) is a lexical analyzer (lexer) generator based upon finite state automata. The input language is similar to the original LEX, but it also implements some extensions of FLEX. Obviously, it has better Unicode support. GPLEX can generate a C# lexer from a grammar file .lex.

The typical grammar is divided in three parts, separated by '%%':

1. options and C# definitions
2. rules with embedded actions
3. usercode

As you can see in the following example in standalone programs the usercode section contains the Main function, so a .lex file can generate a complete functioning program. Although this make it always quite messy and hard to read for the untrained reader.

```
// example from the documentation
%namespace LexScanner
%option noparser nofiles

alpha [a-zA-Z]

%%
foo         |
bar         Console.WriteLine("keyword " + yytext);
{alpha}{3}  Console.WriteLine("TLA " + yytext);
{alpha}+    Console.WriteLine("ident: " + yytext);
```

```
%%

    public static void Main(string[] argp) {
        Scanner scnr = new Scanner();
        for (int i = 0; i < argp.Length; i++) {
            Console.WriteLine("Scanning \"" + argp[i] + "\"");
            scnr.SetSource(argp[i], 0);
            scnr.yylex();
        }
    }
```

It can be used as a standalone tool but, being a lexer generator, can also work with parser generators: it is designed to work with Gardens Point Parser Generator, however it has also been used with COCO/R and custom parsers.

Context Free

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

ANTLR

ANTLR is a great parser generator written in Java that can also generate parsers for C# and many other languages. A particularity of the C# target is that there are actually two versions: the original by sharwell and the new standard runtime. The original defined itself as *C# optimized*, while the standard one is included in the general distribution of the tool. Neither is a fork, since the authors work together, and both are mentioned in the official website. It is more of a divergent path. The grammars are compatible, but the generated parsers are not.

If you are unsure which one to pick I suggest the standard one, since it is slightly more updated. In fact, the standard has a release version supporting .NET Core, while the original only a pre-release.

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 is quite popular for its many useful features: for instance, version 4 supports direct left-recursive rules. However, a real added value of a vast community it is the large amount of grammars available.

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.

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++.

Gardens Point Parser Generator

> *Gardens Point Parser Generator (GPPG) is a parser generator that produces parsers written in the C# V2 or higher. The input language is YACC-like, and the parsers are LALR(1), with the usual automatic disambiguations. Designed to work with GPLEX.*

There are some adaptation to make it work with C# and its tools (e.g. MPF). However, a particular feature of GPPG is the possibility of generating also an HTML report of the structure of the generated parser. This is meant to simplify understanding and analysis of the parser and grammar.

It is designed to work with its brother GPLEX, however it has also been used with COCO/R and custom lexers. The structure of the grammar is similar to the one of the brother, but instead of .lex it has the extension .y.

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 ;
```

Hime Parser Generator

Hime Parser Generator is a parser generator for .NET and Java, with a modern implementation of GLR with the RNGLR algorithm. It is a pragmatic tool with everything you need and nothing more. It does not reinvent the wheel, but it does improve it.

The grammar uses a custom language based on BNF with some enhancement. For instance, it supports context-sensitive lexing (useful for soft keywords), template rules to avoid repetition of similar rules and feature to transform the parse tree in an AST. These features are called tree actions: drop and promote. One drops the node from the tree, the other substitute the node with its children.

Instead of embedding code in a Hime grammar you can annotate a rule with something called *semantic action* in the documentation. 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 grammar is put in a file with .gram extension. The structure of the file resembles the classic one with the three sections: options, terminals and rules. But it is much cleaner and looks like a C# class.

```
grammar MathExp
{
	options
	{
		Axiom = "exp";
		Separator = "SEPARATOR";
	}
	terminals
	{
		WHITE_SPACE  -> U+0020 | U+0009 | U+000B | U+000C ;
		SEPARATOR    -> WHITE_SPACE+;

		INTEGER      -> [1-9] [0-9]* | '0' ;
		REAL   -> INTEGER? '.' INTEGER  (('e' | 'E') ('+' | '-')? INTEGER)?
		                    |  INTEGER ('e' | 'E') ('+' | '-')? INTEGER ;
		NUMBER -> INTEGER | REAL ;
	}
	rules
```

```
	{
		exp_atom-> NUMBER^ @OnNumber // @ is a semantic action
				| '('! exp^ ')'! ; // the final ! drop the node from
the final tree
		exp_op0	-> exp_atom^
				|  exp_op0 '*'^ exp_atom // the ^ drop the node and
changes with its children
				|  exp_op0 '/'^ exp_atom ;
		exp_op1	-> exp_op0^
				|  exp_op1 '+'^ exp_op0
				|  exp_op1 '-'^ exp_op0 ;
		exp	-> exp_op1^ ;
	}
}
```

The documentation is concise, but complete: there are tutorials and recipes to explain practical usage of the tool. There is an equally good enough grammar repository. It has debug features, like the generation of DOT files.

LLLPG

> *LLLPG is not a dedicated tool the way ANTLR is. Instead, LLLPG is designed to be embedded inside another programming language*

So LLLPG is a LL(k) parser generator, that is not actually a standalone parser generator, but a part of a larger project called Enhanced C#. It is not really usable standalone, because it does not even generate a complete class, but the tool only translates the parts of the input file that it recognizes.

It also bizarrely claims to be better than ANTLR2 (released in 2006), despite being updated until recently. But we mentioned it because for the very narrow objective of building a custom language on the .NET it is a good tool, designed just for that objective.

PEG

After the CFG parsers is time to see the PEG parsers available for C#.

IronMeta

> *The IronMeta parser generator provides a programming language and application for generating pattern matchers on arbitrary streams of objects. It is an implementation of Alessandro Warth's OMeta system in C#.*

Despite the potentially perplexing reference to be a programming language IronMeta is a PEG parser generator that works just like any other one.

IronMeta improve upon base OMeta allowing direct and indirect left recursion. On the other hand, error reporting and documentation are limited.

An IronMeta grammar can contain embedded actions and conditions. A condition is a boolean expression that controls the activation of the following rule. If the condition is true the rule activates.

```
// from the documentation
// IronMeta Calculator Example

using System;
using System.Linq;

ironmeta Calc<char, int> : IronMeta.Matcher.CharMatcher<int>
{
    Expression = Additive;

    Additive = Add | Sub | Multiplicative;

    Add = BinaryOp(Additive, '+', Multiplicative) -> { return
_IM_Result.Results.Aggregate((total, n) => total + n); };
    Sub = BinaryOp(Additive, '-', Multiplicative) -> { return
_IM_Result.Results.Aggregate((total, n) => total - n); };

    Multiplicative = Multiply | Divide;
    Multiplicative = Number(DecimalDigit);

    Multiply = BinaryOp(Multiplicative, "*", Number, DecimalDigit) -> { return
_IM_Result.Results.Aggregate((p, n) => p * n); };
    Divide = BinaryOp(Multiplicative, "/", Number, DecimalDigit) -> { return
_IM_Result.Results.Aggregate((q, n) => q / n); };

    BinaryOp :first :op :second .?:type = first:a KW(op) second(type):b -> {
return new List<int> { a, b }; };

    Number :type = Digits(type):n WS* -> { return n; };

    Digits :type = Digits(type):a type:b -> { return a*10 + b; };
    Digits :type = type;

    DecimalDigit = .:c ?( (char)c >= '0' && (char)c <= '9' ) -> { return (char)c -
'0'; };
    KW :str = str WS*;
    WS = ' ' | '\n' | '\r' | '\t';
}
```

Pegasus

> *[Pegasus] is a PEG (Parsing Expression Grammar) parser generator for .NET that integrates with MSBuild and Visual Studio.*

Pegasus is a simple parser generator with an equally sparse documentation. It supports the formal definition of PEG and does have basic features to simplify the management of indentation and debugging.

A Pegasus grammar is written in a .peg file that is quite clean, but can also include embedded action.

```
@namespace MyProject
@classname ExpressionParser

additive <double> -memoize
    = left:additive "+" right:multiplicative { left + right }
    / left:additive "-" right:multiplicative { left - right }
    / multiplicative

multiplicative <double> -memoize
    = left:multiplicative "*" right:power { left * right }
    / left:multiplicative "/" right:power { left / right }
    / power

power <double>
    = left:primary "^" right:power { Math.Pow(left, right) }
    / primary

primary <double> -memoize
    = decimal
    / "-" primary:primary { -primary }
    / "(" additive:additive ")" { additive }

decimal <double>
    = value:([0-9]+ ("." [0-9]+)?) { double.Parse(value)
```

Parser Combinators

They allow you to create a parser simply with C# 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 C# 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 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.

Sprache And Superpower

> *Sprache is a simple, lightweight library for constructing parsers directly in C# code.*
>
> *It doesn't compete with "industrial strength" language workbenches - it fits somewhere in between regular expressions and a full-featured toolset like ANTLR.*

The documentation says it all, except how to use it. You can understand how to use it mostly reading tutorials, including one [we have written for Sprache]. However, it is quite popular and cited in the credits for ReSharper.

There is no grammar, you use the functions it provides as you would for normal code.

```
// from our tutorial. Command is just a class of our program
public static Parser<Command> Command = Parse.Char('<').Then(_ => Parse.Char('>'))

.Return(SpracheGameCore.Command.Between)
                                        .Or(Parse.Char('<')

.Return(SpracheGameCore.Command.Less))
                                        .Or(Parse.Char('>')

.Return(SpracheGameCore.Command.Greater))
                                        .Or(Parse.Char('=')

.Return(SpracheGameCore.Command.Equal));

// another example of the nice LINQ-like syntax for combining parser functions
public static Parser<Play> Play =
        (from action in Command
        from value in Number
        select new Play(action, value, null))
    .Or(from firstValue in Number
        from action in Command
        from secondValue in Number
        select new Play(action, firstValue, secondValue));
```

> *A parser combinator library based on Sprache. Superpower generates friendlier error messages through its support for token-based parsers.*

Superpower comes from the same author and it is a slightly more advanced tool with an equal lack of documentation. Being newer there are also no tutorials.

Parseq, Parsley and LanguageExt.Parsec

These are three ports of the famous Parsec Library in Haskell. They all have some reasons to choose one over the other.

> *Parseq is a monadic parser combinator library written for C#, It can parse context-sensitive, infinite-lookahead grammers.*

Parseq seems to be a straight port of Haskell. But there is no documentation, so if you know how to use Parsec it might be a good choice, otherwise you are on your own.

> *Parsley is a monadic parser combinator library inspired by Haskell's Parsec and F#'s FParsec. It can parse context-sensitive, infinite look-ahead grammars but it performs best on predictive (LL[1]) grammars.*

Parsley is a parser combinator, but it has a separate lexer and parser phase. In practical terms it means that is simple to use, but it also familiar to experienced creators of parsers. There is a limited amount of documentation, but a complete JSON example used as an integration test.

```
// an example from the documentation
var text = new Text("1 2 3 a b c");
var lexer = new Lexer(new Pattern("letter", @"[a-z]"),
                      new Pattern("number", @"[0-9]+"),
                      new Pattern("whitespace", @"\s+", skippable: true));

// in real usage you are probably going to use a LINQ-like syntax to get the
tokens
Token[] tokens = lexer.ToArray();
```

> *The Parsec library is almost an exact replica of the Haskell Parsec library, it can be used to parse very simple blocks of text up to entire language parsers.*

LanguageExt.Parsec is a port of the Haskell library in a larger library designed to bring functional features in C# 6. There is a minimum amount of documentation to get you started.

```
// example from the documentation
var spaces = many(satisfy(Char.IsWhiteSpace));
var word = from w in many1(letter)   // letter = satisfy(Char.IsLetter)
           from s in spaces
           select w;
var parser = many1(word);
var result = parse(parser, "two words");
```

It is obviously the best choice if you also need a bit of F# in your C#, but is quite good on its own.

Pidgin

> *Pidgin is a parser combinator library, a lightweight, high-level, declarative tool for constructing parsers.*

Pidgin is a new parser combinator library that is already quite mature and useful. Like Sprache, it is easy to use and supports a nice LINQ-like syntax. It also has a few advantages over Sprache: it is more actively maintained, is faster, consumes less memory, supports binary input and include support for advanced features such as recursive structures or operator precedence.

Recursive structures are made possible by a specific operator that allows to defer execution of a parser to another section of the code. The operator precedence is managed with a class made to deal with expressions.

The following is a partial JSON example from the repository.

```
public static class JsonParser
{
    private static readonly Parser<char, char> LBrace = Char('{');
    [..]
    private static readonly Parser<char, char> ColonWhitespace =
        Colon.Between(SkipWhitespaces);

    [..]
```

```
    private static readonly Parser<char, IJson> Json =
        JsonString.Or(Rec(() => JsonArray)).Or(Rec(() => JsonObject));

    private static readonly Parser<char, IJson> JsonArray =
        Json.Between(SkipWhitespaces)
            .Separated(Comma)
            .Between(LBracket, RBracket)
            .Select<IJson>(els => new JsonArray(els.ToImmutableArray()));

    [..]
}
```

The documentation is good and cover many aspects: a tutorial/reference, suggestions to speed up your code and a comparison with other parser combinator libraries. The repository also contains examples on JSON and XML. The tutorial/reference is not as deep as one would like, but it gets you started. The author also gave a talk at NDC that includes a tutorial about Pidgin.

Best Way to Parse C#: Roslyn

> *Roslyn provides open-source C# and Visual Basic compilers with rich code analysis APIs. It enables building code analysis tools with the same APIs that are used by Visual Studio.*

There is one special case that could be managed in more specific way: the case in which you want to parse C# code in C#. In such cases you should use the .NET Compiler Platform which it is a *compiler as a service* better known as Roslyn. It is open source and also the official C# parser, so there is no better choice.

In practical terms it works as a library that you can use to parse C#, but also to generate C# and do everything a compiler can do. The only weak point may be the abundant, but somewhat badly organized documentation. Luckily you can read a few tutorials we have written for Roslyn.

Tools That We Cannot Recommend

We want to also list some tools that people usually mention and are interesting, but we could not include in this analysis for several reasons.

Irony

> *Irony is a development kit for implementing languages on .NET platform.*

Irony is a parser generator that does not rely on a grammar, but on overloading operators in C# to express grammar constructs. It also includes an interpreter. It has not been updated since a 2013 beta release and it does not seem it ever had a stable version. Although there is a recent modified version that supports .NET Core.

GOLD

> *GOLD is a free parsing system that is designed to support multiple programming languages.*

In practical terms it is an IDE that supports the creation of BNF grammars to generate parsers in many languages, including Assembly, C, C#, D, Java, Pascal, Python, Visual Basic.NET and Visual C++. It has been relevant enough to have its own wikipedia article, but it is not updated since 2012.

TinyPG

Tiny Parser Generator is an interesting tool presented in a popular CodeProject article that also spawn a fork. It is a tool with a simple IDE that can generate lexer, scanner and parse trees representation. But it can also generate a syntax highlighter for a text box. It is neat, but we cannot recommend it because it was never really meant for professional use and it is not update anymore.

Summary

As we said in the sister article about parsing in Java, the world of parsers is a bit different from the usual world of programmers. That is because a lot of good tools come directly from academia and, in that sector, Java is more popular than C#. So, there are fewer parsing tool for C# than for Java. Also some, like ANTLR, are written in Java, but can produce C# code. This does not mean that there are not good options, but there are fewer of them.

On the other hand, if you need to parse C# you have the chance to use the official compiler very easily, so that is a plus.

We cannot really say 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. So, we wanted to share what we have learned on the best options for parsing in C#.

Thanks to Lee Humphries for his feedback on this article and *Benjamin Hodgson for having signalled to us Pidgin.*

Python

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 Python 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.

Context Free

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

ANTLR

ANTLR is a great parser generator written in Java that can also generate parsers for Python and many other languages. There is also a beta version for TypeScript from the same guy that makes the *optimized C#* version. 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 is quite popular for its many useful features: for instance, version 4 supports direct left-recursive rules. However, a real added value of a vast community it is the large amount of grammars available.

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.

Lark

> *A modern parsing library for Python, implementing Earley & LALR(1) and an easy interface*

Lark is a parser generator that works as a library. You write the grammar in a string or a file and then use it as an argument to dynamically generate the parser. Lark can use two algorithms: Earley is used when you need to parse all grammars and LALR when you need speed. Earley can parse also ambiguous grammars. Lark offers the chance to automatically solve the ambiguity by choosing the simplest option or reporting all options.

Lark grammars are written in an EBNF format. They cannot include actions. This means that they are clean and readable, but also that you have to traverse the resulting tree yourself. Although there is a function that can help with that if you use the LALR algorithm. On the positive side you can also use specific notations in the grammar to automatically generate an AST. You can do that by dropping certain nodes, merging or transforming them.

The following example grammar shows a useful feature of Lark: it includes rules for common things, like whitespace or numbers.

```
parser = Lark('''?sum: product
                     | sum "+" product   -> add
                     | sum "-" product   -> sub

                ?product: item
                     | product "*" item  -> mul
                     | product "/" item  -> div

                ?item: NUMBER           -> number
                     | "-" item         -> neg
                     | "(" sum ")"

                %import common.NUMBER
                %import common.WS
                %ignore WS
        ''', start='sum')
```

Lark comes with a tool to convert Nearley grammars in its own format. It also includes a useful function to transform the tree generated by the parser in an image.

It has a sufficient documentation, with examples and tutorials available. There is also a small reference.

Lrparsing

> *lrparsing is an LR(1) parser hiding behind a pythonic interface*

Lrparsing is a parser generator whose grammars are defined as Python expressions. These expressions are attribute of a class that corresponds to rule of a traditional grammar. They are usually dynamically generated, but the library provides a function to precompile a parse table beforehand.

Given their format depending on Python, lrparsing grammars can be easy to read for Python developers, but they are harder to read than a traditional grammar.

```
// from the documentation
class ExprParser(lrparsing.Grammar):
    #
    # Put Tokens we don't want to re-type in a TokenRegistry.
    #
    class T(lrparsing.TokenRegistry):
        integer = Token(re="[0-9]+")
        integer["key"] = "I'm a mapping!"
        ident = Token(re="[A-Za-z_][A-Za-z_0-9]*")
    #
    # Grammar rules.
    #
    expr = Ref("expr")                  # Forward reference
    call = T.ident + '(' + List(expr, ',') + ')'
    atom = T.ident | T.integer | Token('(') + expr + ')' | call
    expr = Prio(                        # If ambiguous choose atom 1st, ...
        atom,
        Tokens("+ - ~") >> THIS,        # >> means right associative
        THIS << Tokens("* / // %") << THIS,
        THIS << Tokens("+ -") << THIS,# THIS means "expr" here
        THIS << (Tokens("== !=") | Keyword("is")) << THIS)
    expr["a"] = "I am a mapping too!"
    START = expr                        # Where the grammar must start
    COMMENTS = (                        # Allow C and Python comments
        Token(re="#(?:[^\r\n]*(?:\r\n?|\n\r?))") |
        Token(re="/[*](?:[^*]|[*][^/])*[*]/"))
```

Lrparsing also provide some basic functions to print parsing tree and grammar rules for debugging purposes.

The documentation is really good: it explains everything you need to know about the library and it also provide some guidance on creating good grammars (eg. solving ambiguities). There are also quite complex example grammars, like one for SQLite.

PLY

> *PLY doesn't try to do anything more or less than provide the basic lex/yacc functionality. In other words, it's not a large parsing framework or a component of some larger system.*

PLY is a stable and maintained tool with a long history starting from 2001. It is also quite basic, given that there are no tools for automatic creation of AST, or anything that a C developer of the previous century would define as *fancy stuff*. The tool was primarily created as instructional tool. This explains its simplicity, but it also the reason because it offers great support for diagnostics or catching mistakes in the grammar.

A PLY grammar is written in Python code in a BNF-like format. Lexer and parser functions can be used separately. The following example shows only the lexer, but the parser works in the same way.

```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

The documentation is extensive, clear, with abundant examples and explanations of parsing concepts. All that you need, if you can get pass the '90 looks.

PlyPlus

> *Plyplus is a general-purpose parser built on top of PLY (LALR(1)), and written in Python. Plyplus features a modern design, and focuses on simplicity without losing power.*

PlyPlus is a tool that is built on top of PLY, but it is very different from it. The authors and the way the names are written are different. Compared to its father the documentation is lacking, but the features are many.

You can write a grammar in a .g file or in a string, but it is always generated dynamically. The format is based on EBNF, but a grammar can also include special notations to simplify the creation of an AST. This notation allows to exclude or drop certain rules from the generated tree.

```
// from the documentation
start: add;

// Rules
?add: (add add_symbol)? mul;
?mul: (mul mul_symbol)? atom;
// rules preceded by @ will not appear in the tree
@atom: neg | number | '\(' add '\)';
neg: '-' atom;

// Tokens
number: '[\d.]+';
mul_symbol: '\*' | '/';
add_symbol: '\+' | '-';

WS: '[ \t]+' (%ignore);
```

PlyPlus include a function to draw an image of a parse tree based upon pydot and graphviz. PlyPlus has unique features, too. It allows you to select nodes in the AST using selectors similar to the CSS selectors used in web development. For instance, if you want to fill all terminal nodes that contain the letter 'n', you can find them like this:

```
// from the documentation
>>> x.select('/.*n.*/:is-leaf')
['Popen', 'isinstance', 'basestring', 'stdin']
```

This is a unique feature that can be useful, for example, if you are developing a static analysis or refactoring tool.

Pyleri

Python Left-Right Parser (pyleri) is part of a family of similar parser generators for JavaScript, Python, C and Go.

A grammar for Pyleri must be defined in Python expressions part of a class. Once defined the grammar can be exported as a file defining the grammar in Python or any other supported language. Apart from this peculiarity Pyleri is a simple and easy to use tool.

```
# from the documentation
# Create a Grammar Class to define your language
class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
```

```
    START = Sequence(k_hi, r_name)

# Compile your grammar by creating an instance of the Grammar Class.
my_grammar = MyGrammar()

# Use the compiled grammar to parse 'strings'
print(my_grammar.parse('hi "Iris"').is_valid) # => True
print(my_grammar.parse('bye "Iris"').is_valid) # => False
```

PEG

After the CFG parsers is time to see the PEG parsers available for JavaScript.

Arpeggio

> *Arpeggio is recursive descent parser with backtracking and memoization (a.k.a. pacrat parser). Arpeggio grammars are based on PEG formalism.*

The documentation defines Arpeggio as a parser interpreter, since parser are generated dynamically from a grammar. In any case it does not work any different from many other Python parser generators. A peculiarity of Arpeggio is that you can define a grammar in a textual PEG format or using Python expressions. Actually, there are two dialects of PEGs, one with a cleaner Python-like syntax and the other the traditional PEG one.

Arpeggio generate a simple parse tree, but it supports the use of a visitor. The visitor can also include a second action to perform after all the tree nodes have been processed. This is used for post-processing, for instance it can be used to deal with symbol reference.

An Arpeggio grammar defined with either a PEG notation or the Python one is usually quite readable. The following example uses Python notation.

```
# partial example from the documentation
def record():                   return field, ZeroOrMore(",", field)
def field():                    return [quoted_field, field_content]
def quoted_field():             return '"', field_content_quoted, '"'
def field_content():            return _(r'([^,\n])+')
def field_content_quoted():     return _(r('"")|([^"]))+')
def csvfile():                  return OneOrMore([record, '\n']), EOF

[..]

def main(debug=False):
    # First we will make a parser - an instance of the CVS parser model.
    # Parser model is given in the form of python constructs therefore we
    # are using ParserPython class.
    # Skipping of whitespace will be done only for tabs and spaces. Newlines
    # have semantics in csv files. They are used to separate records.
parser = ParserPython(csvfile, ws='\t ', debug=debug)

[..]
```

There are a couple of options for debugging: verbose and informative ouput and the generation of DOT files of the parser. The DOT files can be used for creating a visualization of the parser, but you will have to call graphviz yourself. The documentation is comprehensive and well-organized.

Arpeggio is the foundation of a more advanced tool for the creation of DSL called textX. TextX is made by the same developer that created Arpeggio and it is inspired by the more famous XText.

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 Python file containing the action code.

```
class Actions(object):
    def make_map(self, input, start, end, elements):
        return {elements[1]: elements[3]}

   [..]
```

Parsimonious

> *Parsimonious aims to be the fastest arbitrary-lookahead parser written in pure Python—and the most usable. It's based on parsing expression grammars (PEGs), which means you feed it a simplified sort of EBNF notation.*

Parsimonious is a no-nonsense tool designed for speed and low usage of RAM. It is also a no-documentation tool, there are not even complete examples. Actually the short README file explain the basics and redirect you to Docstring for more specific information.

In any case Parsimonious is good working tool that allows you dynamically create a grammar defined in a file or a string. You can also define a visitor to traverse and transform the parsing tree. So, if you are already familiar with the PEG format you do not need to know anything else to use it at its fullest.

A Parsimonious grammar is readable like any other PEG grammar.

```
# example from the documentation
my_grammar = Grammar(r"""
    styled_text = bold_text / italic_text
    bold_text   = "((" text "))"
    italic_text = "''" text "''"
    text        = ~"[A-Z 0-9]*"i
    """)
```

pyPEG

> *pyPEG is a plain and simple intrinsic parser interpreter framework for Python version 2.7 and 3.x*

PyPEG is a framework to parse and compose text. Which means that you define a grammar in a syntax as powerful as PEG, but you do it in Python code. And then you use this grammar to parse and/or compose a text based upon that grammar. Obviously if you compose a text you have to provide the data yourself. In this case it works as a template system.

The syntax for a PyPEG is on the verbose side, frankly it is too verbose to be productive if you just want to use it for simple parsing. But it is a cool library if you want to parse and manipulate some document in a specific format. For instance, you could use it to transform documentation in one format to another.

```
# from the documentation
from pypeg2 import *
class Type(Keyword):
grammar = Enum( K("int"), K("long") )

class Parameter:
grammar = attr("typing", Type), name()

class Parameters(Namespace):
grammar = optional(csl(Parameter))

class Instruction(str):
grammar = word, ";"

block = "{", maybe_some(Instruction), "}"
class Function(List):
grammar = attr("typing", Type), name(), \
        "(", attr("parms", Parameters), ")", block

f = parse("int f(int a, long b) { do_this; do_that; }", Function)
```

PyPEG does not produce a standard tree, but a structure based upon the defined grammar. Look at what happens for the previous example.

```
# execute the example
>>> f.name
```

```
Symbol('f')
>>> f.typing
Symbol('int')
>>> f.parms["b"].typing
Symbol('long')
>>> f[0]
'do_this'
>>> f[1]
'do_that'
```

TatSu

> *TatSu (for grammar compiler) is a tool that takes grammars in a variation of EBNF as input, and outputs memoizing (Packrat) PEG parsers in Python.*

TatSu is the successor of Grako, another parser generator tool, and it has a good level of compatibility with it. It can create a parser dynamically from a grammar or compiling into a Python module.

TatSu generate PEG parsers, but grammars are defined in a variant of EBNF. Though the order of rules matters as it is usual for PEG grammars. So it is actually a sort of cross between the two. This variant includes support for dealing with associativity and simplifying the generated tree or model (more on that later). Support for left-recursive rule is present, but experimental.

```
// TatSu example grammar from the tutorial
@@grammar::CALC

start
    =
    expression $
    ;

expression
    =
    | expression '+' term
    | expression '-' term
    | term
    ;

term
    =
    | term '*' factor
    | term '/' factor
    | factor
    ;

factor
    =
    | '(' expression ')'
    | number
    ;
```

```
number
    =
    /\d+/
```

TatSu grammars cannot include actions, that can be defined in a separate Python class. Instead you have to annotate the grammar if you want to use an object model in place of semantic actions. An object model is a way to separate the parsing process from the entity that is parsed. In practical terms instead of doing something when a certain rule is matched you do something when a certain object is defined. This object may be defined by more than one rule.

The following extract example defines an object Multiply that corresponds to the rule multiplication.

```
multiplication::Multiply
    =
    left:factor op:'*' ~ right:term
    ;
```

The object model can then be used for what TatSu calls *walker* (essentially a visitor for the model).

```
from tatsu.walkers import NodeWalker

class CalcWalker(NodeWalker):
    def walk_object(self, node):
        return node

    def walk__add(self, node):
        return self.walk(node.left) + self.walk(node.right)

    def walk__subtract(self, node):
        return self.walk(node.left) - self.walk(node.right)

    def walk__multiply(self, node):
        return self.walk(node.left) * self.walk(node.right)

    def walk__divide(self, node):
        return self.walk(node.left) / self.walk(node.right)

def parse_and_walk_model():
    grammar = open('grammars/calc_model.ebnf').read()

    parser = tatsu.compile(grammar, asmodel=True)
    model = parser.parse('3 + 5 * ( 10 - 20 )')

    print('# WALKER RESULT IS:')
    print(CalcWalker().walk(model))
    print()
```

The same object model can also be used for code generation, for instance to transform one format into another one. But for that you obviously cannot reuse the walker, but you have to define a template class for each object.

TatSu provides also: a tool to translate ANTLR grammars, complex trace output and a graphical representation of the tree using pygraphviz. ANLTR grammar may have to be manually adapted to respect PEG constraints.

The documentation is complete: it shows all the features, provide examples and even has basic introduction to parsing concepts, like AST.

Waxeye

> *Waxeye is a parser generator based on parsing expression grammars (PEGs). It supports C, Java, Javascript, Python, Ruby and Scheme.*

Waxeye can facilitate the creation of an AST by defining nodes in the grammar that will not be included in the generated tree. That is quite useful, but a drawback of Waxeye is that it only generates a AST. In the sense that there is no way to automatically execute an action when you match a node. You have to traverse and execute what you need manually.

One positive side-effect of this limiation is that grammars are easily readable and clean. They are also independent from any language.

```
// from the manual

calc  <- ws sum

sum   <- prod *([+-] ws prod)

prod  <- unary *([*/] ws unary)

unary <= '-' ws unary
       | :'(' ws sum :')' ws
       | num

num   <- +[0-9] ?('.' +[0-9]) ws

ws    <: *[ \t\n\r]
```

A particular feature of Waxeye is that it provides some help to compose different grammars together and then it facilitates modularity. For instance, you could create a common grammar for identifiers, that are usually similar in many languages.

Waxeye has a great documentation in the form of a manual that explains basic concepts and how to use the tool for all the languages it supports. There are a few example grammars.

Parser Combinators

They allow you to create a parser by combining different pattern matching functions, that are equivalent to grammar rules. They are generally considered best suited for simpler parsing needs.

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.

Parsec.py, Parsy and PyParsing

> *A universal Python parser combinator library inspired by Parsec library of Haskell.*

That is basically the extent of the documentation on Parsec.py. Though there are a couple of examples. If you already know how to use the original Parsec library or one of its many clones you can try to use it. It does not look bad, but the lack of documentation is a problem for new users.

> *Parsy is an easy way to combine simple, small parsers into complex, larger parsers. If it means anything to you, it's a monadic parser combinator library for LL(infinity) grammars in the spirit of Parsec, Parsnip, and Parsimmon.*

Parsy is another Parsec-inspired libary which has a mininum of documentation. It also no more update since the end of 2014.

> *The pyparsing module is an alternative approach to creating and executing simple grammars, vs. the traditional lex/yacc approach, or the use of regular expressions. The pyparsing module provides a library of classes that client code uses to construct the grammar directly in Python code.*

Pyparsing is a stable and mature software developed for more than 14 years which has many examples, but still a confusing and lacking documentation.

The following example shows how easy it is to use.

```
# example from the documentation
# define grammar
greet = Word( alphas ) + "," + Word( alphas ) + "!"

# input string
hello = "Hello, World!"

# parse input string
print hello, "->", greet.parseString( hello )
```

Python Libraries Related to Parsing

Python offers also some other libraries or tools related to parsing.

Parsing Python Inside Python

There is one special case that could be managed in more specific way: the case in which you want to parse Python code in Python. When it comes to Python the best choice is to rely on your own Python interpreter.

The standard reference implementation of Python, known as CPython, include a few modules to access its internals for parsing: tokenize, parser and ast. You may also be able to use the parser in the PyPy interpreter.

Parsing with Regular Expressions and The Like

Usually you resort to parsing libraries and tools when regular expression are not enough. However, there is a good library for Python than can extend the life and usefulness of regular expressions or using elements of similar complexity.

> *Regular Expression based parsers for extracting data from natural languages [..]*
>
> *This library basically just gives you a way to combine Regular Expressions together and hook them up to some callback functions in Python.*

Reparse is a simple tool that can nonetheless quite useful in certain scenarios. The author himself says that it is much simpler and with less feature than PyParsing or Parboiled.

The basic idea is that you define regular expressions, the patterns in which they can combine and the functions that are called when an expression or pattern is found. You must define functions in Python, but expressions and pattern can be defined in Yaml, JSON or Python.

In this example from the documentation expressions and patterns are defined in Yaml.

```
Color:
    Basic Color:
        Expression: (Red|Orange|Yellow|Green|Blue|Violet|Brown|Black)
        Matches: Orange | Green
        Non-Matches: White
        Groups:
          - Color

Time:
    Basic Time:
        Expression: ([0-9]|[1][0-2]) \s? (am|pm)
        Matches: 8am | 3 pm
        Non-Matches: 8a | 8:00 am | 13pm
        Groups:
          - Hour
          - AMPM
```

Fields like Matches are there for humans, but can be used for testing by Reparse.

```
BasicColorTime:
  Order: 1
  Pattern: |
    <Color> \s? at \s? <Time>
```

```
  # Angle brackets detonate expression groups
  # Multiple expressions in one group are combined together
```

An example function in Python for the pattern.

```
from datetime import time
def color_time(Color=None, Time=None):
    Color, Hour, Period = Color[0], int(Time[0]), Time[1]
    if Period == 'pm':
        Hour += 12
    Time = time(hour=Hour)

    return Color, Time

functions = {
   'BasicColorTime' : color_time,
}
```

The file that puts everything together.

```
from reparse_functions import functions
import reparse

colortime_parser = reparse.parser(
    parser_type=reparse.basic_parser,
    expressions_yaml_path=path + "expressions.yaml",
    patterns_yaml_path=path + "patterns.yaml",
    functions=functions
)

print(colortime_parser("~ ~ ~ go to the store ~ buy green at 11pm! ~ ~"))
```

Parsing Binary Data: Construct

Instead of writing *imperative code* to parse a piece of data, you declaratively define a *data structure* that describes your data. As this data structure is not code, you can use it in one direction to *parse* data into Pythonic objects, and in the other direction, to *build* objects into binary data.

And that is it: Construct. You could parse binary data even with some parser generators (e.g. ANTLR), but Constuct make it much easier. It is a sort of DSL combined with a parser combinator to parse binary formats. It gives you a bunch of fields to manage binary data: apart from the obvious ones (e.g. float, string, bytes etc.), there are a few specialized to manage sequences of fields (sequence), group of them (struct) and a few conditional statements.

It also makes available functions to adapt or validate (test) the data and debug any problem you found.

As you can see in the following example, it is quite easy to use.

```
# from the documentation
```

```
gif_logical_screen = Struct("logical_screen",
    ULInt16("width"),
    ULInt16("height"),
    [..]
    If(lambda ctx: ctx["flags"]["global_color_table"],
        Array(lambda ctx: 2**(ctx["flags"]["global_color_table_bpp"] + 1),
            Struct("palette",
                ULInt8("R"),
                ULInt8("G"),
                ULInt8("B")
            )
        )
    )
)

gif_header = Struct("gif_header",
    Const("signature", b"GIF"),
    Const("version", b"89a"),
)

[..]

gif_file = Struct("gif_file",
    gif_header,
    gif_logical_screen,
    [..]
)

if __name__ == "__main__":
    f = open("../../../tests/sample.gif", "rb")
    s = f.read()
    f.close()
    # if you want to build the file, you just have to provide the data
    # to the build() function
    print(gif_file.parse(s))
```

There is a nice amount of documentation and even many example grammars for different kinds of format, such as filesystems or graphics files.

Summary

Any programming language has a different community with its peculiarities. These differences remain even when we compare the same interests across the languages. For instance, when we compare parsers tools we can see how Java and Python developers live in a different world.

The parsing tools and libraries for Python for the most part use very readable grammars and are simple to use. But the most interesting thing is that they cover a very wide spectrum of competence and use cases. There seems to be an uninterrupted line of tools available from regular expression, passing through Reparse to end with TatSu and ANTLR.

Sometimes this means that it can be confusing, if you are a parsing expert coming from a different language. Because few parser generators actually generate parsers, but they mostly interpret them at runtime. On the other hand, with Python you can really find the perfect library,

or tools, for your needs. And to help you with that we hope that this comparison has been useful for you.

JavaScript

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 JavaScript 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.

Lexer

Lexer is a lexer that claims to be modelled after flex. Though a fairer description would be a short lexer based upon RegExp. The documentation seems minimal, with just a few examples, but the whole thing is 147 lines of code, so it is actually comprehensive.

However, in a few lines manages to support a few interesting things and it appears to be quite popular and easy to use. One thing is its supports RingoJS, a JavaScript platform on top of the JVM. Another one is the integration with Jison, the Bison clone in JavaScript. If you temper your expectations it can be a useful tool.

There is no grammar, you just use a function to define the RegExp pattern and the action that should be executed when the pattern is matched. So, it is a cross between a lexer generator and a lexer combinator. You cannot combine different lexer functions, like in a lexer combinator, but the lexer it is only created dynamically at runtime, so it is not a proper lexer generator either.

```
var lexer = new Lexer;

var chars = lines = 0;

lexer.addRule(/\n/, function () {
    lines++;
    chars++;
});

lexer.addRule(/./, function () {
    chars++;
});

lexer.setInput("Hello World!\n Hello Stars!\n!")
```

```
lexer.lex();
```

Context Free

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

ANTLR

ANTLR is a great parser generator written in Java that can also generate parsers for JavaScript and many other languages. There is also a beta version for TypeScript from the same guy that makes the *optimized C#* version. 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 is quite popular for its many useful features: for instance, version 4 supports direct left-recursive rules. However, a real added value of a vast community it is the large amount of grammars available.

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
```

Jison

> *Jison generates bottom-up parsers in JavaScript. Its API is similar to Bison's, hence the name.*

Despite the name Jison can also replace flex, so you do not need a separate lexer. Although you can use one or build your own custom lexer. All you need is an object with the functions setInput and lex. It can best recognize languages described by LALR(1) grammars, though it also has modes for LR(0), SLR(1).

The generated parser does not require a runtime component, you can use it as a standalone software.

It has a good enough documentation with a few examples and even a section to try your grammars online. Although at times it relies on the Bison manual to cover lack of its own documentation.

A Jison grammar can be inputted using a custom JSON format or a Bison-styled one. Both requires you to use embedded actions if you want to do something when a rule is matched. The following example is in the custom JSON format.

```
// example from the documentation
{
   "comment": "JSON Math Parser",
   // JavaScript comments also work

   "lex": {
      "rules": [
         ["\\s+",                    "/* skip whitespace */"],
         ["[0-9]+(?:\\.[0-9]+)?\\b", "return 'NUMBER'"],
         ["\\*",                     "return '*'"],
         ["\\/",                     "return '/'"],
         ["-",                       "return '-'"],
         ["\\+",                     "return '+'"],
         [..] // skipping some parts
      ]
   },
```

```
   "operators": [
      ["left", "+", "-"],
      ["left", "*", "/"],
      ["left", "UMINUS"]
      [..] // skipping some parts
   ],

   "bnf": {
      "expressions": [["e EOF",   "return $1"]],

      "e" :[
         ["e + e",  "$$ = $1+$3"],
         ["e - e",  "$$ = $1-$3"],
         ["- e",    "$$ = -$2", {"prec": "UMINUS"}],
         ["NUMBER", "$$ = Number(yytext)"],
         [..] // skipping some parts
      ]
   }
}
```

It is very popular and used by many project including CoffeeScript and Handlebars.js.

Nearley

> *nearley uses the Earley parsing algorithm augmented with Joop Leo's optimizations to parse complex data structures easily. nearley is über-fast and really powerful. It can parse literally anything you throw at it.*

The Earley algorithm is designed to easily handle all grammars, including left-recursive and ambiguous ones. Essentially its main advantage it is that it should never catastrophically fail. On the other hand, it could be slower than other parsing algorithms. Nearly itself also is able to detect some ambiguous grammars. It can also and reports multiple results in the case of an ambiguous input.

A Nearley grammar is a written in a .ne file that can include custom code. A rule can include an embedded action, which the documentation calls a *postprocessing function*. You can also use a custom lexer.

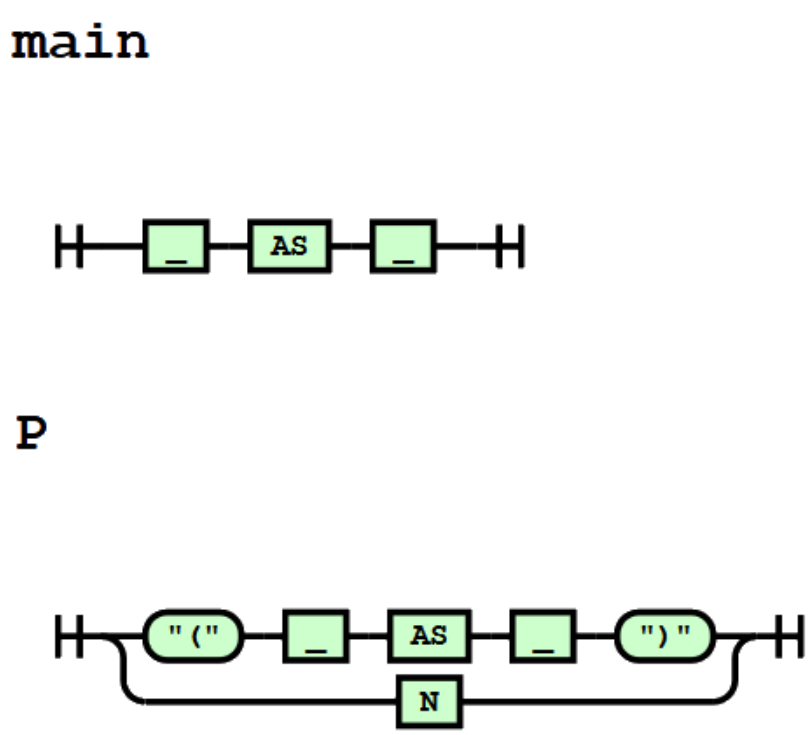
```
# from the examples in Nearly
# csv.ne

@{%
var appendItem = function (a, b) { return function (d) { return
d[a].concat([d[b]]); } };
var appendItemChar = function (a, b) { return function (d) { return
d[a].concat(d[b]); } };
var empty = function (d) { return []; };
var emptyStr = function (d) { return ""; };
%}

file              -> header newline rows             {% function (d) { return {
```

```
header: d[0], rows: d[2] }; } %}
# id is a special preprocessor function that return the first token in the match
header            -> row                              {% id %}

rows              -> row
                   | rows newline row                 {% appendItem(0,2) %}

row               -> field
                   | row "," field                    {% appendItem(0,2) %}

field             -> unquoted_field                   {% id %}
                   | "\"" quoted_field "\""           {% function (d) { return
d[1]; } %}

quoted_field      -> null                             {% emptyStr %}
                   | quoted_field quoted_field_char   {% appendItemChar(0,1) %}

quoted_field_char -> [^"]                             {% id %}
                   | "\"" "\""                        {% function (d) { return
"\""; } %}

unquoted_field    -> null                             {% emptyStr %}
                   | unquoted_field char              {% appendItemChar(0,1) %}

char              -> [^\n\r",]                        {% id %}

newline           -> "\r" "\n"                        {% empty %}ca
                   | "\r" | "\n"                      {% empty %}
```

Another interesting feature is that you could build custom tokens. You can define them using a tokenizing library, a literal or a test function. The test function must return true if the text corresponds to that specific token.

```
@{%
var print_tok  = {literal: "print"};
var number_tok = {test: function(x) {return x.constructor === Number; }}
%}

main -> %print_tok %number_tok
```

Nearley include tools for debugging and understanding your parser. For instance, Unparser can automatically generate random strings that are considered correct by your parser. This is useful to test your parser against random noise or even to generate data from a schema (e.g. a random email address). It also include a tool to generate SVG railroad diagrams: a graphical way to represent a grammar.



Railroad diagrams from the documentation demo

A Nearley parser requires the Nearley runtime.

Nearley documentation is a good overview of what is available and there is also a third-party playground to try a grammar online. But you will not find a complete explanation of all the features.

PEG

After the CFG parsers is time to see the PEG parsers available for JavaScript.

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 JavaScript file containing the action code.

```
var maps = require('./maps');

var actions = {
  make_map: function(input, start, end, elements) {
    var map = {};
    map[elements[1]] = elements[3];
    return map;
  },
  [..]
};
```

Ohm

> *Ohm is a parser generator consisting of a library and a domain-specific language*

Ohm grammars are defined in a custom format that can be put in a separate file or in a string. However, the parser is generated dynamically and not with a separate tool. In that sense it works like a parser library more than a traditional parser generator.

A helper function to create an AST is included among the extras.

> *In Ohm, a grammar defines a language, and semantic actions specify what to do with valid inputs in that language*

A grammar is completely separated from semantic actions. This simplify portability and readability and allows to support different languages with the same grammar. At the moment Ohm only supports JavaScript, but more languages are planned for the future. The typical grammar is then clean and readable.

```
Arithmetic {
  Exp
    = AddExp

  AddExp
    = AddExp "+" PriExp  -- plus
    | AddExp "-" PriExp  -- minus
    | PriExp

  PriExp
    = "(" Exp ")"  -- paren
    | number

  number
    = digit+
}
```

Semantic actions can be implemented using a simple API. In practical terms this ends up working like the visitor pattern with the difference that is easier to define more groups of semantic actions.

```
var semantics = g.createSemantics().addOperation('eval', {
  Exp: function(e) {
    return e.eval();
  },
  AddExp: function(e) {
    return e.eval();
  },
  AddExp_plus: function(left, op, right) {
    return left.eval() + right.eval();
  },
  AddExp_minus: function(left, op, right) {
    return left.eval() - right.eval();
  },
  PriExp: function(e) {
    return e.eval();
  },
  PriExp_paren: function(open, exp, close) {
    return exp.eval();
  },
  number: function(chars) {
    return parseInt(this.sourceString, 10);
  },
});
var match = g.match('1 + (2 - 3) + 4');
console.log(semantics(match).eval());
```

To support debugging Ohm has a text trace and (work in progress) graphical visualizer. You can see the graphical visualizer at work and test a grammar in the interactive editor.

The documentation is not that bad, though you have to go under the doc directory to find it. There is no tutorial, but there are a few examples and a reference. In particular the documentation suggests reading a well commented Math example.

PEG.js

> *PEG.js is a simple parser generator for JavaScript that produces fast parsers with excellent error reporting.*

PEG.js can work as a traditional parser generator and create a parser with a tool or can generate one using a grammar defined in the code. It supports different module loaders (e.g. AMD, CommonJS, etc.).

PEG.js has a neat online editor that allows to write a grammar, test the generated parser and download it.

A PEG.js grammar is usually written in a .pegjs file and can include embedded actions, custom initializing code and semantic predicates. That is to say functions that determine if a specific match is activated or not.

```
// example from the documentation
{
  function makeInteger(o) {
    return parseInt(o.join(""), 10);
  }
}

start
  = additive

additive
  = left:multiplicative "+" right:additive { return left + right; }
  / multiplicative

multiplicative
  = left:primary "*" right:multiplicative { return left * right; }
  / primary

primary
  = integer
  / "(" additive:additive ")" { return additive; }

integer "integer"
  = digits:[0-9]+ { return makeInteger(digits); }
```

The documentation is good enough, there are a few example grammars, but there are no tutorials available. The popularity of the project had led to the development of third-party tools, like one to generate railroad diagrams, and plugins, like one to generate TypeScrypt parsers.

Waxeye

> *Waxeye is a parser generator based on parsing expression grammars (PEGs). It supports C, Java, Javascript, Python, Ruby and Scheme.*

Waxeye can facilitate the creation of an AST by defining nodes in the grammar that will not be included in the generated tree. That is quite useful, but a drawback of Waxeye is that it only generates a AST. In the sense that there is no way to automatically execute an action when you match a node. You have to traverse and execute what you need manually.

One positive side-effect of this limitation is that grammars are easily readable and clean. They are also independent from any language.

```
// from the manual

calc  <- ws sum

sum   <- prod *([+-] ws prod)

prod  <- unary *([*/] ws unary)

unary <= '-' ws unary
       | :'(' ws sum :')' ws
```

```
        | num

num    <- +[0-9] ?('.' +[0-9]) ws

ws     <: *[ \t\n\r]
```

A particular feature of Waxeye is that it provides some help to compose different grammars together and then it facilitates modularity. For instance, you could create a common grammar for identifiers, that are usually similar in many languages.

Waxeye has a great documentation in the form of a manual that explains basic concepts and how to use the tool for all the languages it supports. There are a few example grammars.

Waxeye seems to be maintained, but it is not actively developed.

Parser Combinators

They allow you to create a parser simply with JavaScript 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 JavaScript 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 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.

Bennu, Parjs And Parsimmon

> *Bennu is a Javascript parser combinator library based on Parsec. The Bennu library consists of a core set of parser combinators that implement Fantasy Land interfaces. More advanced functionality such as detailed error messaging, custom parser state, memoization, and running unmodified parsers incrementally is also supported.*

All libraries are inspired by Parsec. Bennu and Parsimmon are the oldest and Bennu the more advanced between the two. They also all have an extensive documentation, but they have is no tutorial. Bennu seems to be maintained, but it is not actively developed.

An example from the documentation.

```
var parse = require('bennu').parse;
var text = require('bennu').text;

var aOrB = parse.either(
    text.character('a'),
    text.character('b'));

parse.run(aOrB, 'b'); // 'b'
```

> *Parsimmon is a small library for writing big parsers made up of lots of little parsers. The API is inspired by parsec and Promises/A+.*

Parsimmon is the most popular among the three, it is stable and updated.

The following is a part of the JSON example.

```
let whitespace = P.regexp(/\s*/m);

let JSONParser = P.createLanguage({
  // This is the main entry point of the parser: a full JSON value.
  value: r =>
    P.alt(
      r.object,
      r.string,
      [..]
    ).thru(parser => whitespace.then(parser)),

  [..]

  // Regexp based parsers should generally be named for better error reporting.
  string: () =>
    token(P.regexp(/"((?:\\.|.)*?)"/, 1))
      .map(interpretEscapes)
      .desc('string'),

  [..]

  object: r =>
    r.lbrace
      .then(r.pair.sepBy(r.comma))
      .skip(r.rbrace)
      .map(pairs => {
        let object = {};
        pairs.forEach(pair => {
          let [key, value] = pair;
          object[key] = value;
        });
        return object;
      }),
});

//////////////////////////////////////////////////////////////////////

let text = // JSON Object

let ast = JSONParser.value.tryParse(text);
```

> *Parjs is a JavaScript library of parser combinators, similar in principle and in design to the likes of Parsec and in particular its F# adaptation FParsec.*

It's also similar to the parsimmon library, but intends to be superior to it.

Parjs is only a few months old, but it is already quite developed. It also has the advantage of being written int TypeScript.

All the libraries have good documentation, but Parjs is great: it explains how to use the parser and also how to design good parsers with it. There are a few examples, including the following on string formatting.

```
/**
 * Created by lifeg on 04/04/2017.
 */
import "../setup";
import {Parjs, LoudParser} from "../src";
//+ DEFINING THE PARSER

//Parse an identifier, an asciiLetter followed by an asciiLetter or digit, e.g.
a12b but not 1ab.
let ident = Parjs.letter.then(Parjs.letter.or(Parjs.digit).many()).str;

//Parse a format token, an `ident` between `{` and `}`. Return the result as a
Token object.
let formatToken = ident.between(Parjs.string("{"), Parjs.string("}")).map(x =>
({token: x}));

//Parse an escaped character. This parses "`{a}" as the text "{a}" instead of a
token.
//Also escapes the escaped char, parsing "``" as "`".
//Works for arbitrary characters like `a being parsed as a.
let escape = Parjs.string("`").then(Parjs.anyChar).str.map(x => ({text:
x.substr(1)}));

//Parse text which is not an escape character or {.
let text = Parjs.noCharOf("`{").many(1).str.map(x => ({text: x}));

//The parser itself. Parses either a formatToken, e.g. {abc} or an escaped combo
`x, or text that doesn't contain `{.
//Parses many times.
let formatParser = formatToken.or(escape, text).many();

//This prints a sequence of tokens {text: "hello, my name is "}, {token: name},
{text: " and I am "}, {token: " years old}, ...
console.log(formatParser.parse("hello, my name is {name} and I am {age} years old.
This is `{escaped}. This is double escaped: ``{name}."));
```

JavaScript Libraries Related to Parsing

There are also some other interesting libraries related to parsing that are not part of a common category.

JavaScript Libraries That Parse JavaScript

There is one special case that could be managed in more specific way: the case in which you want to parse JavaScript code in JavaScript. Contrary to what we have found for Java and C# there is not a definitive choice: there are many good choices to parse JavaScript.

The three most popular libraries seems to be: Acorn, Esprima and UglifyJS. We are not going to say which one it is best because they all seem to be awesome, updated and well supported.

One important difference is that UglifyJS is also a mangler/compressor/beautifier toolkit, which means that it also has many other uses. On the other hand, it is the only one to support only up to the version ECMAScript 5. Another thing to consider is that only esprima have a documentation worthy of projects of such magnitude.

Interesting Parsing Libraries: Chevrotain

> *Chevrotain is a very fast and feature rich JavaScript LL(k) Parsing DSL. It can be used to build parsers/compilers/interperters for various use cases ranging from simple configuration files, to full fledged programing languages.*

There is another interesting parsing tool that does not really fit in more common categories of tools, like parser generators or combinators: Chevrotain, a parsing DSL. A parsing DSL works as a cross between a parser combinator and a parser generator. You define a grammar in JavaScript code directly, but using the (Chevrotrain) API and not a standard syntax like EBNF or PEG.

Chevotrain supports many advanced features typical of parser generators: like semantic predicates, separate lexer and parser and a grammar definition (optionally) separated from the actions. The actions can be implemented using a visitor and thus you can reuse the same grammar for multiple projects.

The following is a partial JSON example grammar from the documentation. As you can see the syntax is clearer to understand for a developer unexperienced in parsing, but a bit more verbose than a standard grammar.

```
// partial JSON grammar for ES6 from the documentation
"use strict"
const chevrotain = require("chevrotain")

// ----------------- lexer -----------------
const Token = chevrotain.Token
const Lexer = chevrotain.Lexer
const Parser = chevrotain.Parser

// With ES6 we can define Tokens using the class keywords.

class True extends Token {}
True.PATTERN = /true/

class False extends Token {}
False.PATTERN = /false/

class StringLiteral extends Token {}
```

```
StringLiteral.PATTERN = /"(?:[^\\"]|\\(?:[bfnrtv"\\/]|u[0-9a-fA-F]{4}))*"/

class WhiteSpace extends Token {}
WhiteSpace.PATTERN = /\s+/
WhiteSpace.GROUP = Lexer.SKIPPED // marking WhiteSpace as 'SKIPPED' makes the
lexer skip it.
WhiteSpace.LINE_BREAKS = true

const allTokens = [
    WhiteSpace,
    StringLiteral,
    True,
    False,
    [..]
]
const JsonLexer = new Lexer(allTokens)

// ----------------- parser -----------------
// Using ES6 the parser too can be defined as a class
class JsonParserES6 extends chevrotain.Parser {
    constructor(input) {
        super(input, allTokens)

        // not mandatory, using $ (or any other sign) to reduce verbosity (this.
this. this. this. .......)
        const $ = this

        $.RULE("json", () => {
            $.OR([
                {ALT: () => {$.SUBRULE($.object)}},
                {ALT: () => {$.SUBRULE($.array)}}
            ])
        })

        [..]

        $.RULE("array", () => {
            $.CONSUME(LSquare)
            $.MANY_SEP({
                SEP: Comma,
                DEF: () => {
                    $.SUBRULE2($.value)
                }
            })
            $.CONSUME(RSquare)
        })

        $.RULE("value", () => {
            $.OR([
                {ALT: () => {$.CONSUME(StringLiteral)}},
                {ALT: () => {$.SUBRULE($.array)}},
```

```
                {ALT: () => {$.CONSUME(True)}},
                {ALT: () => {$.CONSUME(False)}},
                [..]
            ])
        })

        // very important to call this after all the rules have been defined.
        // otherwise the parser may not work correctly as it will lack information
        // derived during the self analysis phase.
        Parser.performSelfAnalysis(this)
    }
}
```

It is very fast, faster than any other JavaScript library and can compete with a custom parser written by hand, depending on the JavaScript engine on which it run on. You can see the numbers and get more details on the benchmark of parsing libraries developed by the author of the library.

It also supports features useful for debugging like railroad diagram generation and custom error reporting. There are also a few features that are useful for building compiler, interpreters or tools for editors, such as automatic error recovery or syntactic content assist. The last one means that it can suggests the next token given a certain input, so it could be used as the building block for an autocomplete feature.

This is not all, Chevrotain even makes available its own engine to external use. This means that you can build your own parsing library on top of Chevrotain. For instance, you can create your own format for a grammar and then use the Chevrotain engine to power the parsing. It even gives you for free error checking features, such as detecting ambiguous alternatives, left recursion, etc.

Chevrotain has a good documentation, with a tutorial, examples grammars and a reference. It also has a great online editor/playground.

Chevrotrain is written in TypeScript.

Summary

As we said in the sisters article about parsing in Java and C#, the world of parsers is a bit different from the usual world of programmers. In the case of JavaScript also the language lives in a different world from any other programming language. There is such disparate level of competence between its developers that you could find the best ones working with people that just barely know how to put together a script. And both want to parse things.

So, for JavaScript there are tools that a bit all over this spectrum. We have serious tools developed by academics for their courses, or in the course of their degrees, together with much simpler tools. Some of which blur the lines between parser generators and parser combinators. And all of them have their place. A further complication is that while usually parser combinators are reserved for easier uses, with JavaScript it is not always the case. You could find very powerful and complex parser combinators and much easier parser generators.

So, with JavaScript more than ever we cannot definitely suggest one software over the other. 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. So, we wanted to share what we have learned on the best options for parsing in JavaScript.

We would like to thank Shahar Soel for having signalled to us Chevrotain and having suggested some needed corrections.