Arpeggio: A flexible PEG parser for Python

Article *in* Knowledge-Based Systems · December 2015

DOI: 10.1016/j.knosys.2015.12.004

CITATIONS
9

READS
996

3 authors:

Igor Dejanovic
University of Novi Sad
49 PUBLICATIONS **159** CITATIONS
SEE PROFILE

Gordana Milosavljevic
University of Novi Sad
57 PUBLICATIONS **256** CITATIONS
SEE PROFILE

Renata Vaderna
University of Novi Sad
17 PUBLICATIONS **47** CITATIONS
SEE PROFILE

Arpeggio: A Flexible PEG Parser for Python

I. Dejanović, G. Milosavljević, R. Vaderna

Faculty of Technical Sciences, University of Novi Sad

Abstract

Arpeggio is a recursive descent parser with full backtracking and memoization based on PEG (Parsing Expression Grammar) grammars. This category of parsers is known as packrat parsers. It is implemented in the Python programming language and works as a grammar interpreter.

Arpeggio has a very good support for error reporting, debugging, and grammar and parse tree visualization. It is used in industrial environments and teaching *Domain-Specific Languages* course at the Faculty of Technical Sciences in Novi Sad. Arpeggio is a foundation of a high-level DSL meta-language and tool - textX.

It is a free and open-source software available at GitHub under MIT license.

Keywords: PEG, packrat, parser, Python, DSL, textX

1. Introduction

A parser is a software component that takes input (usually textual) and produces a data structure. This transformation is often based on a formal description of the input language syntax - a grammar. A traditional way to define the syntax of a programming language is Chomsky's generative system of grammars[1], in particular, Context-Free Grammars and Regular Expressions. The main problem with this approach is that it was meant to be used to describe natural languages where the possibility to define ambiguity is a desirable feature. But, the very same feature is a source of serious problems when describing machine-oriented syntaxes.

Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntaxes, which solves the ambiguity problem by not introducing ambiguity in the first place [2].

Arpeggio is an implementation of a PEG-based recursive descent parser with backtracking and memoization implemented in the Python programming language. This class of parsers is known as *packrat parsers*[3]. Full backtracking enables an unlimited lookahead while linear parse time is still preserved using memoization technique where intermediate results are cached.

The main motivation to design and implement Arpeggio was to provide a parsing infrastructure for a Domain-Specific Languages (DSL)[4] development tool textX[5]. Nevertheless, as parsers are important parts of many software tools and libraries (e.g. [6]), Arpeggio is

built to be suitable for all sorts of general purpose parsing. It is used in data extraction from various textual formats, parsing of different languages, analysis of legacy source code, etc.

2. Problems and Background

The development of DSLs usually requires a lot of experimentation through trial and error. Furthermore, DSLs are much more prone to change than General-Purpose Languages (GPL). Thus, tools for DSL development should be built in such a way that the grammar is readable, simple to change and extend, and to enable fast round-trip.

From the start, Arpeggio is designed to work as a grammar interpreter as opposed to grammar compiler (i.e. parser generator). Furthermore, various grammar syntaxes are supported, both for people that have extensive knowledge in traditional grammar specification (e.g. EBNF) and for those that know the host language well (Python) and feel more natural when coding grammars using Python language and IDE support.

Packrat parsers are recursive descent parsers with full backtracking[3]. This approach yields an unlimited lookahead but could lead to exponential parse time in the worst-case scenarios. Memoization is used to keep track of the results of already parsed parts of the input. This technique ensures that parse time remains linear.

One of the most popular parsing tools for Python in the past was Pyparsing[7]. It is a mature parsing library based on PEG with a large set of examples and good documentation. But the only way to describe a grammar in Pyparsing is by using Python language. There is no external DSL variant that is more suitable for seasoned EBNF grammar writers. Besides, the parse tree transformation is supported by "parse actions" that get triggered during parsing which hampers backtracking and prohibits the transformation of the same parse tree to multiple representations.

A promising project that has emerged in the last several years is Parsimonious[8]. The goal of this parser is to be fast and frugal on RAM usage while maintaining usage simplicity. Currently, Parsimonious is in development and lacks some features already implemented in Arpeggio (e.g. multiple syntaxes, whitespace handling, visualization). Moreover, there is no documentation nor any examples at the moment.

3. Software Framework

From the given grammar Arpeggio builds, in runtime, an instance of the parser, which is a graph of Python objects whose classes inherit `ParsingExpression` class (Figure 1).

We call this graph of objects *the parser model*. The parser model for the simple grammar given in Figure 3 is given in Figure 2.

A grammar may be specified using different syntaxes. A canonical form of the grammar specification is the *internal* DSL form [4], i.e. the grammar is defined using Python language elements (Figure 3).

In this version of Arpeggio, there are two more grammar specification languages both, of them implemented as *external* DSLs. These two languages differ slightly, and both are

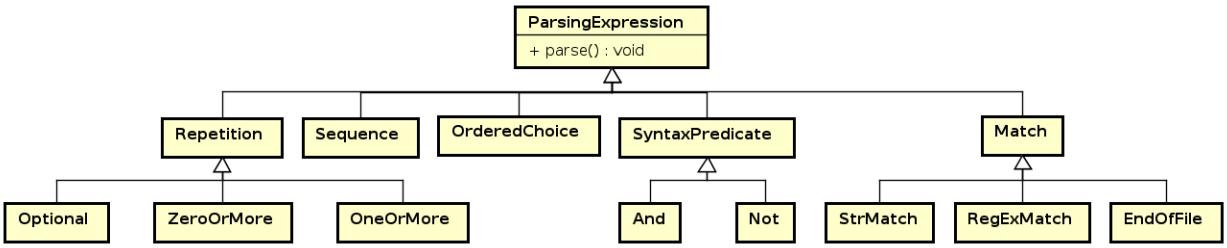

Figure 1: The hierarchy of Arpeggio's PEG classes.

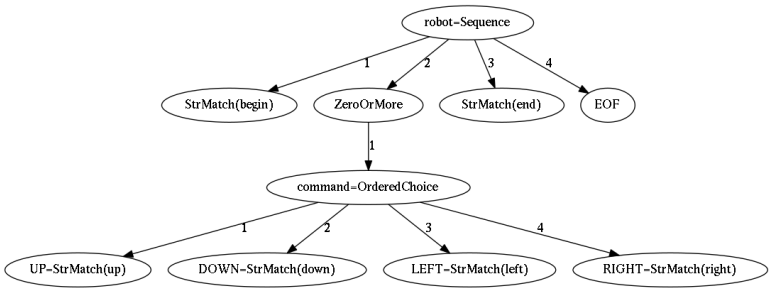

Figure 2: The parser model of the robot language.

implemented using Arpeggio itself [1]. Thus, their implementation is a good example of Arpeggio's capabilities.

Other features of Arpeggio are: (1) Case sensitive/insensitive parsing (configurable per parser), (2) Whitespace handling control, (3) Direct support for language code comments (comments are treated as whitespaces), (4) Keywords handling, (5) Newline termination for repetition operators (available for internal syntax form only), (6) Parse tree navigation operators, (7) visitor pattern for semantic analysis, (8) extensive error reporting and debugging support, (9) parse tree and parse model visualization (using `GraphViz dot` tool[2]).

All of these features are thoroughly explained in the online docs [3].

```
def robot():        return 'begin', ZeroOrMore(command), 'end', EOF
def command():      return [UP, DOWN, LEFT, RIGHT]
def UP():           return 'up'
def DOWN():         return 'down'
def LEFT():         return 'left'
def RIGHT():        return 'right'
```

Figure 3: A simple robot language grammar in Arpeggio's internal Python-based form.

[1] We eat our own dog food - https://en.wikipedia.org/wiki/Eating_your_own_dog_food
[2] http://www.graphviz.org/
[3] http://igordejanovic.net/Arpeggio/

```
robot = 'begin' (command)* 'end' EOF
command = UP/DOWN/LEFT/RIGHT
UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'
```

Figure 4: Robot language grammar in Arpeggio's clean PEG external form.

4. Implementation and Empirical Results

Arpeggio is written in the pure Python programming language without any dependencies[4]. It can be installed from PyPI [5] using the standard Python installer - `pip` [6]. The details of the installation and usage can be found in the project documentation.

Arpeggio has been validated in various academic and industrial projects. It is covered with extensive unit tests. Our previous work shows that Arpeggio's speed is comparable to or in some cases outperforms some popular Python parsers [9].

Arpeggio has been used in many projects as a part of a higher level tool for DSL construction textX[5]. textX uses Arpeggio as a core parsing technology. Some open-source projects that use Arpeggio through textX are listed on the textX project page.

textX and Arpeggio are used in the tool-chain of Typhoon-HIL inc. for the definition and parsing of power electronics models, component description, toolbox specification, and custom component definitions.

5. Illustrative Examples

The Arpeggio code repository hosts 11 different examples in the `examples` directory. Each example comes with a README file which contains its description and instructions on how to run it[7]. Additionally, we provide three full-length tutorials (CSV, BibTeX and Calc) in the documentation[8]. Here we will briefly describe each example.

The BibTeX example demonstrates parsing of the BibTeX format[9] which is used for specification of bibliographic information. This example accepts a file name from the command line. The given file should be properly formatted BibTeX file that is parsed and transformed to a list of BibTeX entries given as Python dictionaries. Transformation is done using an instance of visitor class `BibtexVisitor`. Test data is given in the `bibtex_example.bib` file.

The Calc example is an implementation of a simple expression evaluator. This example is done in both internal (`calc.py`) and external (`calc_peg.py` and `calc_cleanpeg.py`) DSL forms. There are two external variants. One is based on a clean PEG syntax (`calc_cleanpeg.py`),

[4] Besides optional `GraphViz` library for visualization.
[5] https://pypi.python.org/pypi
[6] https://docs.python.org/3/installing/
[7] https://github.com/igordejanovic/Arpeggio/tree/v1.2/examples
[8] http://igordejanovic.net/Arpeggio/
[9] http://www.bibtex.org/

while the other is based on the traditional PEG syntax (`calc_peg.py`). Visitor object performs the evaluation of the expression by transforming the parse tree to an expression value. In the external PEG versions, the same visitor from the internal DSL version is used. The only difference is that the grammars are in different forms, and thus `ParserPEG` class is used to instantiate parsers instead of `ParserPython`.

The CSV example shows how to build a simple parser for the CSV (*Comma Separated Values*) file format. A test data is given in the `test_data.csv` file. In this example, we see how to change whitespace handling by replacing standard whitespaces with tab and space characters. This is done because newline has semantics in the CSV file format. It is used to separate records. To handle newlines properly, we have to instruct Arpeggio not to treat newlines as whitespaces. This configuration is given in the call to `ParserPython`. Parameter `ws` is set to `'\t '`. This example is done in both internal(`csv.py`) and external clean PEG form (`csv_peg.py` and `csv.peg`).

The JSON parser example (`json.py`) demonstrates how to parse the popular JSON format for data exchange. There are excellent Python packages for working with the JSON format though. Test data is given in the `test.json` file.

The robot example is another example given in both internal (`robot.py`) and external (`robot_peg.py`) DSL forms. This is a simple imperative language for "robot control". The grammar for the external variant is given in the `robot.peg` file, while the example program is given in the `program.rbt` file. This example is used for Figures 3 and 4 as well as the parser model in Figure 2.

The simple language example (`simple.py`) is an example of a very simple C-like language. The `program.simple` file contains an example of a small program written in this language.

And last but not least, there is an example of PEG language specification in PEG itself (`peg_peg.py`). In this example, a grammar of the PEG language is specified in PEG language itself (file `peg.peg`). `PEGVisitor` from `arpeggio.peg` module is used for transforming a parse tree to a fully working parser. We demonstrate that the PEG parser constructed by the visitor is fully capable of parsing PEG grammars. Thus, parsing and transformation of PEG grammar definition could be reapplied indefinitely.

6. Conclusions

Arpeggio is an implementation of a packrat parser that brings unambiguous parsing with unlimited lookahead while still preserving linear parse times. Its performance has been tested and our results show that its speed is comparable to other popular solutions and in some cases outperforms them. It has been used for several years in both academic and industrial environments and is covered with extensive unit tests. Because of its usage on the DSL course at the Faculty of Technical Sciences we have developed an extensive error reporting and debugging support.

Grammars can be written in the internal Python-based form, and two external DSL forms. Furthermore, additional grammar syntaxes can be provided. Arpeggio works as a grammar interpreter. No code generation is performed.

It comes with 11 examples and complete documentation. The full source code is available at GitHub under a permissive MIT license. The project employs continuous integration to ensure the stability of the development branch.

7. References

[1] N. Chomsky, Three models for the description of language, Information Theory, IRE Transactions on 2 (3) (1956) 113–124.

[2] B. Ford, Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, ACM SIGPLAN Notices 39 (1) (2004) 111–122.

[3] B. Ford, Packrat Parsing: Simple, Powerful, Lazy, Linear Time, in: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, ACM New York, NY, USA, 2002, pp. 36–47.

[4] M. Fowler, Domain-Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[5] I. Dejanović, textX, https://github.com/igordejanovic/textX, [Online; Accessed: 2015-09-02].

[6] G. Petrović, H. Fujita, Soner: Social network ranker, Neurocomputing (2015) –doi:http://dx.doi.org/10.1016/j.neucom.2015.10.021.
URL http://www.sciencedirect.com/science/article/pii/S0925231215014800

[7] P. McGuire, Pyparsing, https://pyparsing.wikispaces.com/, [Online; Accessed: 2015-09-02].

[8] E. Rose, Parsimonious, https://github.com/erikrose/parsimonious, [Online; Accessed: 2015-09-02].

[9] I. Dejanović, G. Milosavljević, Performance Evaluation of the Arpeggio Parser, in: 4rd International Conference on Information Society Technology and Management (ICIST 2014), Vol. 1, Kopaonik, Serbia, 2014, pp. 229–234.

Required Metadata

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.2
C2	Permanent link to code/repository used of this code version	https://github.com/igordejanovic/Arpeggio/tree/v1.2
C3	Legal Code License	MIT
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Python 2.7, 3.2 - 3.5
C6	Compilation requirements, operating environments & dependencies	Optional dependency: GraphViz v2.2x for visualization
C7	If available Link to developer documentation/manual	http://igordejanovic.net/Arpeggio/
C8	Support email for questions	igor.dejanovic@gmail.com

Table 1: Code metadata