Performance Evaluation of the Arpeggio Parser

Igor Dejanovic
University of Novi Sad
31 PUBLICATIONS 37 CITATIONS
SEE PROFILE

Gordana Milosavljevic
University of Novi Sad
40 PUBLICATIONS 103 CITATIONS
SEE PROFILE

PERFORMANCE EVALUATION OF THE ARPEGGIO PARSER

Igor Dejanović, Gordana Milosavljević
Faculty of Technical Sciences, University of Novi Sad

Abstract *- PEGs (Parsing Expression Grammar) is a type of formal grammar which is gaining significant traction in the industry and academia. This paper presents a performance evaluation of our Arpeggio parser - PEG interpreter in Python programming language. The aim of the Arpeggio parser is to be used in our DSL framework for both compiling (transformation) and code editing support (e.g. syntax checking and highlighting, code completion/navigation, tree outline, code visualization). The evaluation is performed by measuring parsing time with different grammars and inputs and comparing parsing times with two other popular python PEG parser interpreters (pyparsing and pyPEG).*

1. INTRODUCTION

The Arpeggio parser [1,2] is a recursive descent parser with full backtracking and memoization based on PEG (*Parsing Expression Grammar*) grammars. This kind of parsers is called packrat parser. Arpeggio's purpose is to be used in our DSL framework[13,14]. Although we are mainly concerned with the functionality of the Arpeggio parser, its performance is of no less importance to us. We strive to make it reasonably fast to be used during editing session for various feedback and features such as syntax highlighting, code outline, code completion and navigation. For those purposes parser and semantic analysis should be executed as background process during editing session. For a good user experience this background tasks should introduce as little overhead as possible.

In this paper we investigate the performance of the Arpeggio parser by performing various run-time benchmarks and comparison with some popular Python parsers (pyparsing[3] and pyPEG[4]). Tests has been performed using various language grammars and input strings.

The purpose of this work is to assess the current state of the Arpeggio parser in terms of performance and to guide its further development. We also make some notes on the lessons learned along the way regarding PEG parsing performance.

The paper is structured as follows: In the Section 2 we give an overview of PEG parsers in general; Sections 3, 4 and 5 give a descriptions of Arpeggio, pyparsing and pyPEG respectively; Section 6 describes test setup, grammars and input files; Section 7 presents results; In Section 8 related work has been presented. In the Section 9 we conclude the paper.

2. PEG

Parsing Expression Grammar (PEG) is a type of formal grammar introduced by Bryan Ford [5] which specifies a set of rules used to recognize strings in the given language. At the same time PEG is a declarative description of the top-down recursive-descent parser. In contrast to context-free grammars (CFG) which are generative in nature, PEGs are oriented toward string recognition. One of the often cited property of PEG is its unambiguity, i.e. if the input string parses there is exactly one valid parse tree. This property is a direct consequence of using prioritized choice operator which instructs the parser to match alternatives in the strict order – from left to right.

The other distinctive feature of PEGs are syntax predicates: *not* (!) and *and* (&). Syntax predicates describe rules that must not match (*not*) or must match (*and*) at the current position in the string without actually consuming any input.

Since PEG is a declarative description of the top-down recursive-descent parser, it is straightforward to make an interpreter of PEG grammars. Arpeggio, as well as pyparsing and pyPEG, are PEG interpreters. This means that no code generation takes place. PEGs can be regarded as "programs" for parsing strings in the defined languages. PEG interpreters run this programs. Of course, this programs can be transformed to other programming languages (e.g. Python, Java, Ruby) using parser generators.

PEG parsers use unlimited lookahead. In some cases this could yield exponential parsing time since parser needs to backtrack on failed matches. To remedy this, Bryan Ford has introduced recursive-descent parser with backtracking that guarantee linear parse time through the use of memoization[6]. Memoization is a technique where the result of each match at each position is remembered and will be used on subsequent matching of the same rule on the same position in the input string. Using this technique, parse time will be linear at the expense of more memory consumption. PEG parser that employs memoization is called *packrat parser*.

3. ARPEGGIO

Arpeggio is a packrat parser written in Python programming language. Its purpose is to be used in our DSL framework. An early design decision was to use interpreting approach instead of code generation. We strongly believe that interpreter would give us more flexibility and faster round-tripping from language description to functional editor in comparison to parser code generation. Furthermore, we argue that for recursive-descent top-down parser written in Python there would be no significant difference in terms of performance between the interpreter and the generated parser. The only difference would be parser startup time. In case of the Arpeggio interpreter, grammar needs to be analyzed and so called “parser model” needs to be created. For generated PEG parsers this initial setup is non-existing. But for our DSL framework this setup will be done during framework startup and upon changes in the language description, where the generated parser would also be regenerated.

Arpeggio has two equivalent ways for grammar definition: canonical (internal DSL style) and textual PEG notation (external DSL style). The canonical description uses Python language constructs (functions, lists, tuples, objects). This approach is inspired by the previous version of pyPEG. Support for textual PEG notation has been built using canonical description of the textual PEG language[1].

```
def number():     return _(r'\d*\.\d*|\d+')
def factor():     return (Optional(["+","-"]), [number,
                            ("(", expression, ")")])
def term():       return factor, ZeroOrMore(["*","/"], factor)
def expression(): return term, ZeroOrMore(["+", "-"], term)
def calc():       return OneOrMore(expression), EndOfFile
```

Figure 1. Grammar for arithmetic expressions given in the Arpeggio’s internal DSL form.

Figure 1 shows a grammar for simple arithmetic expressions with five basic operations (negation, addition, subtraction, multiplication and division). Each PEG rule is given in the form of Python function. Rule functions accept no parameters and return Python object describing PEG rule. Prioritized (ordered) choice is given as a Python list while a sequence is specified using tuple objects. Rule *number* is a regular expression match of unsigned real number.

```
number <- r'\d*\.\d*|\d+';
factor <- ("+" / "-")?
          (number / "(" expression ")");
term <- factor (( "*" / "/") factor)*;
expression <- term (("+" / "-") term)*;
calc <- expression EndOfFile;
```

Figure 2. Grammar for arithmetic expressions given in the Arpeggio’s external DSL form.

The same grammar, in the form of PEG textual notation, is given in Figure 2. Arpeggio parses this description using canonical parser for PEG language and, using semantic actions, instantiates the parser for the specified grammar. This definition will construct the same parser as the canonical definition from Figure 1[2].

From the grammar definition given in either form, Arpeggio will instantiate the parser which can be visualized using Graphviz tool[3]. Figure 5. shows a parser instance for grammar for simple arithmetic expression. This can be handy for grammar debugging.

4. PYPARSING

Pyparsing [3] is parser combinator written in Python programming language. It is one of the most popular parsers in Python with the long history and strong community. Grammar is specified by combining simple parsers given in the form of Python class instances. Combination is performed through overloaded operators (+, |, <<, ~, etc.). This gives pyparsing a great flexibility and composition properties.

Figure 3 presents the same grammar for simple arithmetic expressions from the previous section implemented with pyparsing..

```
plus = Literal("+")
minus = Literal("-")
mul = Literal("*")
div = Literal("/")
number = Regex(r'\d*\.\d*|\d+')
expression = Forward()
factor = Optional(plus | minus) + (number |
         (Literal("(") + expression + Literal(")")))
term = factor + ZeroOrMore((mul | div) + factor)
expression << term + ZeroOrMore((plus | minus) + term)
calc = OneOrMore(expression)
```

Figure 3. Grammar for arithmetic expressions in pyparsing

Pyparsing uses semantic actions to perform semantic analysis during parsing. Without using semantic actions, parse result is a flat list of tokens. *Group* class may be used to construct a data structure similar to the Arpeggio parse tree. The addition of *Group* instances in the grammar introduces an additional overhead. To examine how this grammar alteration influences parsing speed we have performed measurements for both cases (with and without *Group*).

5. PYPEG

PyPEG [4] is another interesting python PEG grammar interpreter. In version 2.x it uses Python classes for grammar definition. It is oriented towards automatic AST (Abstract Syntax Tree) construction and pretty-printing (code generation). Each PEG rule in pyPEG is specified as a Python class with *grammar* class attribute. This class will get instantiated where match succeeds.

A part of the grammar for arithmetic expression in pyPEG is given in Figure 4.

```
class Term(List):
    grammar = Factor, \
              maybe_some([u"*", u"/"], Factor)
class Expression(List):
    grammar = Term, \
              maybe_some([u"+", u"-"], Term)
class Calc(List):
    grammar = some(Expression)
```

Figure 4. A part of expression grammar in pyPEG.

1 See ParserPEG python class from https://github.com/igordejanovic/arpeggio/blob/master/arpeggio/peg.py
2 The details can be found in the arpeggio source code.
3 Graphviz - http://www.graphviz.org/

6. TEST SETUP

Testing has been performed for 5 different grammars and 6 different input string sizes for each grammar. For each input and each grammar 1000 parser runs have been performed. Measurement has been performed on the conventional laptop computer (Core 2 Duo 2.26GHz, 8GB RAM). We did our best to reduce additional overhead while running tests (no other applications were running, cron/at service was disabled etc.). Benchmarking is performed using python *timeit* module. Input loading is done in the setup code to avoid I/O overhead. For each grammar and parser we tried to find the fastest way to perform parsing (e.g. for pyparsing we used regular expression match instead of *Combine* class).

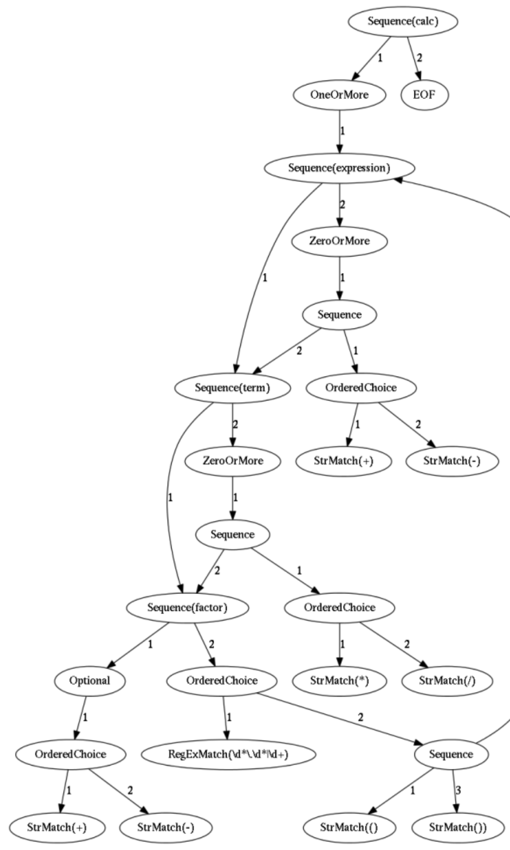

Figure 5. Parser structure for simple arithmetic expressions grammar.

Testing has been performed using the following grammars:

1. The simple language for arithmetic expressions given in Figure 1.
2. BibTeX [7] – language for defining bibliography references.
3. CSV (Comma-Separated Values) [8] – a simple format for storing tabular data in textual form.
4. The language for arithmetic expressions altered to induce backtracking.
5. A classical non context-free language $\{a^n b^n c^n, n>=1\}$ [5]

The first three languages are widely known and we think that they do not require further explanation so we focus on the fourth and the fifth.

In the fourth grammar we have made modification given in Figure 6 to induce backtracking on the always failing *expression2* rule. The input was the same as in the case of grammar 1 so *expression2* kept failing while trying to match '#' at the end of the *expression*. Memoizing parser should not introduce significant overhead because after failing match on *expression2* the parser will try *expression* rule but this time it would use match results from the cache.

```
def expression2():
    return term, ZeroOrMore(["+", "-"], term), "#"
def expression():
    return term, ZeroOrMore(["+", "-"], term)
def calcfile():
    return OneOrMore([expression2, expression]),\
           EndOfFile
```

Figure 6. The change in the grammar to induce backtracking.

The fifth grammar is given in Figure 7. Syntactic predicates *And*, *Not* and *Empty* do not consume any input. *And* will succeed if the given PEG expression is matched, *Not* will fail if given expression is matched and will succeed otherwise, *Empty* will always succeed. It is clear that this grammar is highly recursive so it is interesting to see how this recursion influences parsing performance with increasing input size.

7. RESULTS

In this section we present test results and give a brief discussion.

The results for simple expression language are given in Table 1.

Expressions	input1	input2	input3	input4	input5	input6
arpeggio	56.28	113.95	173.47	232.45	296.96	354.73
pyparsing	52.51	105.41	158.14	212.06	264.90	314.33
pyparsing – Group	61.39	122.68	184.43	245.70	304.67	371.78
pyPEG	250.30	518.77	796.52	1084.58	1388.04	1698.11

Table 1. Results for simple arihtmetic expression grammar.

These results show that Arpeggio and pyparsing have similar speed for this language. Arpeggio is a little slower in comparison to pyparsing with plain expression grammar but when pyparsing *Group* is used it becomes a little faster. Those differences are small enough to have marginal impact in practical use. These results were consistent with the increasing input size.

What was surprising to us was the low performance of pyPEG. It can be seen that pyPEG is considerably slower in comparison to both Arpeggio and pyparsing. In our opinion pyPEG in version 2.x is considerably redesigned and oriented towards language design and code generation so the speed is probably not the current focus of the project. pyPEG was significantly slower in all tests so we decided to remove its result from graphs in the rest of the paper.

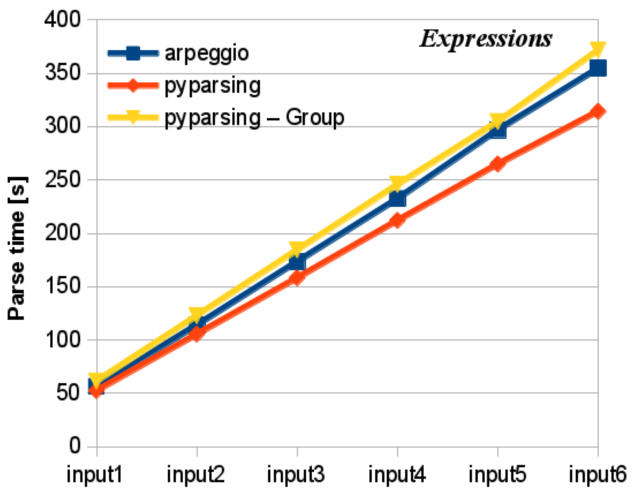

Figure 7. Graph results for simple arihtmetic expression grammar

The graph in Figure 7 shows differences between Arpeggio and pyparsing for the simple expression language and various input sizes.

Table 2 and Figure 8 present results for BibTeX grammar. This time Arpeggio is slightly faster than pyparsing in both *Group* and non-*Group* versions, but again there was not much difference so it should not matter much in the real-world use-cases.

Bibtex	input1	input2	input3	input4	input5	input6
arpeggio	4.74	9.53	14.38	19.50	24.92	30.25
pyparsing	5.74	11.60	17.34	23.28	29.08	34.64
pyparsing – Group	6.60	13.19	19.70	26.16	32.82	39.28
pyPEG	30.76	66.50	108.12	155.14	207.04	263.81

Table 2. Results for BibTeX grammar.

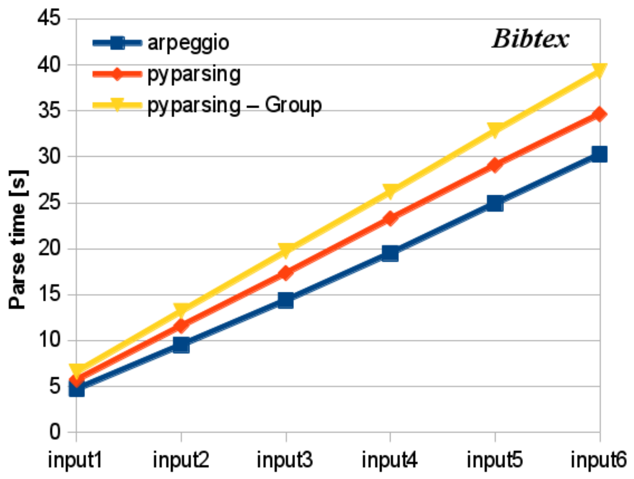

Figure 8. Graph results for BibTeX grammar.

For CSV grammar results given in Table 3 and Figure 9 pyparsing is faster in both *Group* and non-*Group* version. The difference is again marginal, especially for the *Group* version which is slower for smaller input but gets faster as the input increases (the slope of the Arpeggio time increase is bigger).

CSV	input1	input2	input3	input4	input5	input 6
arpeggio	21.35	43.39	65.71	89.41	114.09	138.49
pyparsing	19.24	38.38	57.20	76.47	95.28	114.14
pyparsing – Group	22.11	44.04	65.68	87.21	109.59	131.35
pyPEG	99.23	214.04	344.53	487.01	645.78	819.92

Table 3. Results for CSV grammar.

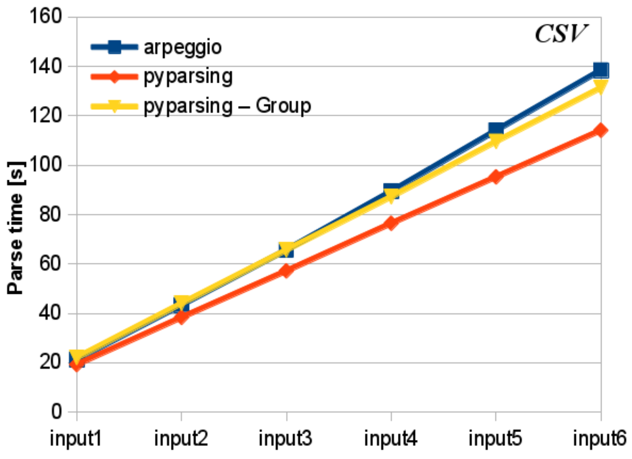

Figure 9. Graph results for CSV grammar.

The results of the fourth test are given in Table 4 and Figure 10. This time there is a noticeable difference in speed in favor of Arpeggio. For grammars and inputs where backtracking is significant Arpeggio performs better. What is also evident is that for the same inputs as in test 1, Arpeggio introduced an overhead of roughly 20% where at the same time pyparsing introduces overhead of approximately 95%. This difference is due to packrat parsing memoization. Packrat machinery is disabled in pyparsing by default while Arpeggio uses packrat parsing always. In the documentation of pyparsing it is stated that it supports packrat parsing, but enabling packrat parsing with `ParserElement.enablePackrat()` (as suggested) has made parsing time even worse. It might be a bug in the pyparsing packrat implementation (we were using the latest 2.0.1 version at the time of this writing) so further investigation should be done.

Expr. backtr.	input1	input2	input3	input4	input5	input6
arpeggio	67.05	135.75	206.00	277.85	348.72	425.91
pyparsing	105.82	209.43	313.10	416.84	526.42	632.45
pyparsing – Group	121.59	243.18	364.05	485.60	607.55	727.92
pyPEG	260.33	538.08	825.08	1137.11	1443.71	1779.21

Table 4. Results for altered expression grammar.

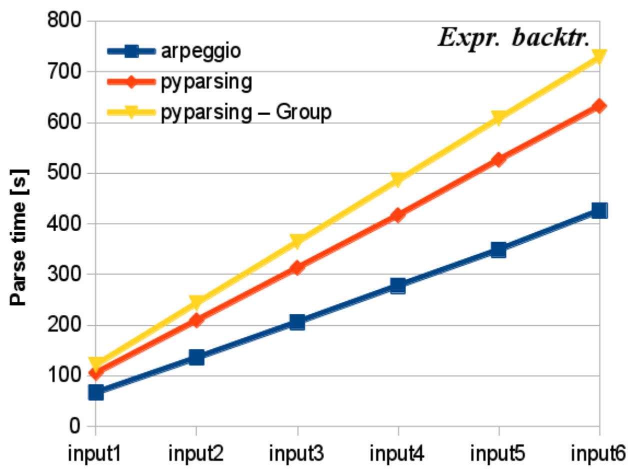

Figure 10. Graph results for altered expressions grammar.

The results of test 5 given in Table 5 and Figure 11 are also interesting. This grammar is highly recursive so, in classical PEG parsing, the maximum recursion depth for Python is eventually reached. Pyparsing reached this limit for 150<n<200 while Arpeggio hit the same limit for 200<n<250. At the same time Arpeggio was faster. For pyPEG we have not been able to define the same grammar as the current version lacks syntactic predicates.

$a^nb^nc^n$	n=50	n=100	n=150	n=200
arpeggio	4.48	9.21	13.90	18.80
pyparsing	5.78	11.50	17.48	NA
pyPEG	NA	NA	NA	NA

Table 5. Results for $a^nb^nc^n$ grammar

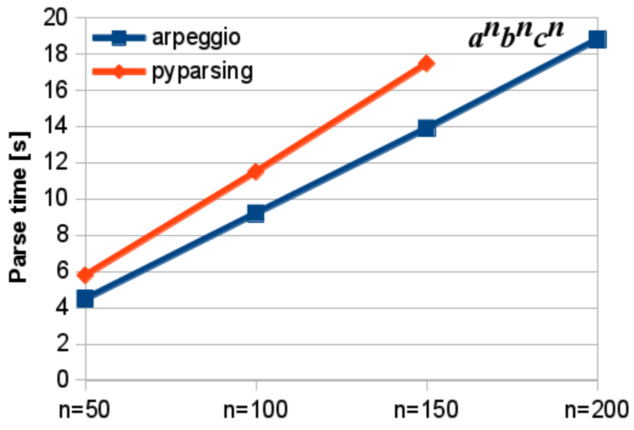

Figure 11. Graph results for $a^nb^nc^n$ grammar.

8. RELATED WORK

PEG, as an alternative to CFG, has gained significant traction in literature in the last decade. Bryan Ford in [5,6] have introduced PEGs and packrat parsing based on the previous research of Alexander Birman [9] and Aho et al. [10]. PEGs represent schematic description of recursive-descent parsers which makes them relatively easy to comprehend and maintain. The construction of PEG parser interpreter is straightforward using dynamic programming languages such as Python or Ruby.

There are many open-source PEG-based parsers in different programming languages. The community maintained list of parser generators and interpreters can be found at Wikipedia [11]. At the time of this writing there were more than 30 parsers for PEG grammars.

For performance optimization there is an interesting project called parsimonious [12] whose main goal is to implement the fastest python PEG parser but at the time of this writing the parsimonious author states in the documentation that the speed optimization has not been finished yet. Nevertheless, ideas employed in this project could be beneficial to the Arpeggio parser.

Another way to increase performance, not only for parser but for the DSL framework as a whole is to use tools like PyPy [13] which could bring noticeable speedups without code changes.

9. CONCLUSION

In this paper we have investigated the current state of our Arpeggio parser in terms of parsing speed. Although speed is not our main goal, we are well aware of the fact that, in a textual DSL framework, parsing is an operation that is executed often for features such as syntax check and highlighting, code navigation, tree outline etc. Thus, for guiding further development, we have compared the speed of Arpeggio with the two popular Python PEG parser interpreters: pyparsing and pyPEG.

The results show that Arpeggio's performance is comparable to pyparsing and outperform it in some cases. Furthermore, Arpeggio's parsing speed is much better than the current pyPEG version.

We have seen that packrat parser implemented in Arpeggio achieves good performance in the event of significant backtracking. The overhead in test 4 was not above 20%.

The performance tests presented here will help us develop new features in Arpeggio while being aware of their implications on the parsing performance.

BIBLIOGRAPHY

[1] Dejanović, I.; Perišić, B., Milosavljević, G. Arpeggio: Pakrat parser interpreter, *Zbornik radova na CD-ROM-u, YUInfo 2010*, 2010.

[2] Arpeggio parser, https://github.com/igordejanovic/arpeggio, online, accessed January 28, 2014.

[3] pyparsing parser, http://pyparsing.wikispaces.com/, online, accessed January 28, 2014.

[4] pyPEG parser, http://fdik.org/pyPEG/, online, accessed January 28, 2014.

[5] Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *ACM SIGPLAN Notices, vol. 39, pp. 111-122, ACM New York, NY, USA*, 2004.

[6] Ford, B., Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pp. 36-47*, 2002.

[7] BibTeX, http://www.bibtex.org/, online, accessed January, 28. 2014.

[8] Comma-separated values (CSV), http://en.wikipedia.org/wiki/Comma-separated_values, online, accessed January, 28. 2014.

[9] Alexander Birman. The TMG Recognition Schema. PhD thesis, Princeton University, February 1970.

[10] Alfred V. Aho and Jeffrey D. Ullman. The Theory of Parsing, Translation and Compiling - Vol. I: Parsing. Prentice Hall, Englewood Cliffs, N.J., 1972.

[11] Comparison of parser generators, http://en.wikipedia.org/wiki/Comparison_of_parser_generators, Wikipedia, The Free Encyclopedia, online, accessed January, 28. 2014.

[12] parsimonious parser, https://github.com/erikrose/parsimonious, online, accessed January, 28. 2014.

[13] PyPy, http://pypy.org/, online, accessed January, 28. 2014.

[13] Dejanović, I.; Perišić, B. & Milosavljević, G. MoRP Meta-metamodel: Towards a Foundation of SLEWorks Language Workbench 2nd International Conference on Information Society Technology (ICIST 2012), pp. 36-40 , 2012.

[14] Dejanović, I.; Milosavljević, G.; Perišić, B.; Vasiljević, I. & Filipović, M. Explicit Support For Languages and Mograms in the SLEWorks Language Workbench 3rd International Conference on Information Society Technology and Management (ICIST 2013), 2013.