Science of Computer Programming 214 (2022) 102734

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Original software publication

Parglare: A LR/GLR parser for Python

Igor Dejanović

Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia

ARTICLE INFO

Article history:
Received 4 February 2021
Received in revised form 23 September 2021
Accepted 28 September 2021
Available online 8 October 2021

Keywords:
Parsing
LR
GLR
Python
Visualization

ABSTRACT

Parglare is a Python parsing library that implements deterministic LR and its generalized extension GLR algorithms. Parglare strives to be easy to use by providing rich error messages, visualization, a CLI tool for grammar development, and good documentation. The same grammar format is used for both algorithms. It is easy to choose either LR parsing if performance is more important or GLR in case a grammar cannot fit into the constraints of deterministic LR parsing and a more powerful parsing is needed. Parglare has been used in data extraction from various textual formats, analysis of legacy source code, and developing and teaching DSL development. The GitHub repository of the project contains multiple examples and a comprehensive functionality and performance test suite.

Software metadata

(executable) Software metadata description	
Current software version	0.14.0
Permanent link to executables of this version	https://github.com/igordejanovic/parglare/releases/download/0.14.0/parglare-0.14.0-py2.py3-none-any.whl
Legal Software License	MIT
Computing platform / Operating System	Python-compatible platforms
Installation requirements & dependencies	Python 3.6–3.9
If available Link to user manual - if formally published include a reference to the publication in the reference list	https://www.igordejanovic.net/parglare/0.14.0/
Support email for questions	igord@uns.ac.rs

Code metadata

Code metadata description	
Current Code version	0.14.0
Permanent link to code / repository used of this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-21-00025
Legal Code License	MIT
Code Versioning system used	git
Software Code Language used	Python
Compilation requirements, Operating environments & dependencies	Python 3.6–3.9
If available Link to developer documentation / manual	https://www.igordejanovic.net/parglare/0.14.0/
Support email for questions	igord@uns.ac.rs

E-mail address: igord@uns.ac.rs.

https://doi.org/10.1016/j.scico.2021.102734
0167-6423/

1. Introduction

Parsing, or syntax analysis, is an essential activity in computing. It is the process of structuring a linear representation following a given formal grammar. The "linear representation" may be a linear sequence in which the preceding elements in some way restrict the next element [1].

Parsing is considered by some to be a solved problem as there are efficient deterministic parsing techniques. These parsing algorithms typically use one token of lookahead and a pre-calculated table to decide the next operation to perform deterministically. However, context-free grammars (CFGs) of most languages, when written in their most natural form, are often ambiguous and do not fit into deterministic classes of context-free grammars, even when special care is taken to design a deterministic grammar [2]. As an example, the Java grammar provided in the parglare repository is created by taking the official grammar from the language specification [3] and adapting for parglare syntax. This grammar produces LALR(1) tables with multiple conflicting states and would require significant work to make it suitable for deterministic parsing. It is known that even expert users of linear-time deterministic parsers are unlikely to produce deterministic grammar on their first attempt [4]. Furthermore, fitting the grammar into a particular grammar class required by the parsing algorithm through "massaging" and transformations leads to hidden logical errors and loses in its naturalness and readability [5]. Maintaining such non-natural grammars is a pain as adding new rules or modifying existing rules can lead to new conflicts [6]. Further problems with non-general grammar classes are difficulties in modularization and extensibility as they are not closed under composition, e.g. if two LR(1) grammars are combined the resulting grammar may not be LR(1).

Generalised parsing techniques accept the full set of CFGs. Allowing for arbitrary context-free grammar enables language designers freedom to write grammars in the way that most naturally conveys the high-level language structure. The set of context-free grammars is closed under composition, which enables easy modularization and embedding. However, the grammars usually end up being ambiguous, which might be problematic in domains where each input string should yield a single interpretation. We have a multitude of mechanisms to specify disambiguation rules [7] but the problem of grammar ambiguity is undecidable [8] so we cannot tell for sure if after applying disambiguation rules we have resolved all ambiguities in the grammar. A usual approach to deal with the problem is either to run some of the existing heuristics, which are unreliable, or to test the grammar by parsing a large number of input strings. Either way, we have no guarantee that we will not run into an input string that will trigger the ambiguity and produce multiple derivations.

Parsing Expression Grammars (PEGs) [9] are another type of formal grammar for the description of formal languages that has gained in popularity in the last two decades. It provides an alternative, recognition-based formal foundation for describing machine-oriented syntaxes, that solves the ambiguity problem by making the choice operator ordered. Although this seems appealing at first, it can cause trouble as a slight change in the grammar, or the order of alternative productions can catch a naive user off guard by introducing unforeseen changes in the recognized language. For example, a simple composition `S -> a / ab` can trip unwary user because, if `a` matches, then `ab` is never tried, even in situations where `ab` could have match the full input sentence [10]. Furthermore, as the most straightforward approach to implementing parsers based on PEG is in a recursive-descent style, parsers do not accept left recursive rules[1] and do not provide any support for ambiguity analysis as PEGs are, by design, never ambiguous. PEGs are a formal system for language recognition based on an old theory of formal grammars called *Top-Down Parsing Language* (TDPL) with an emphasis on the parser and not a language description. Thus, its style is more imperative and it is generally quite difficult to determine what language a given PEG defines [13].

In general, parsing approaches are either too limited (deterministic parsing), allow ambiguity (generalized parsing), or are hard to reason about (PEG) [10].

Nevertheless, generalized parsing offers an appealing approach if the grammar readability, understandability, composability, and ease of maintenance is important and one is willing to accept super-linear performance, explicit dealing with ambiguity and possible ambiguous parses.

In the last decade, with the uptake of data science and natural language processing, parsing tools are becoming more and more important outside the domain of the classic compiler construction. With this shift, generalized parsing techniques seem to be getting more traction.

These days memory and processing power are not such a limiting factor as they were five decades ago. Thus, in an increasing number of applications, the ease of use and naturalness of language grammar description is of more importance than the raw execution performance.

On the other hand, as the Python programming language is becoming the *lingua franca* of scientific computing, it is of great importance to have a set of good quality, easy to use, well documented and maintained generalized parsing tools for Python.

2. Problems and background

Extracting data from textual sources can be difficult without proper tools. Although programming languages come equipped with a regular expressions engine, which can help, the language of the regular expressions is not powerful enough

[1] Although, extensions to PEG based parsers do exist that can cope with left recursion [11,12].

to describe more complex nested patterns. For these tasks, descriptions based on more powerful grammar formalisms are needed.

Even though it is quite possible to write parsers manually, and many programmers do so, developing a parser by hand is very hard for some approaches.[2] Furthermore, manual parser development misses the opportunity to reuse the battle-tested tooling support in grammar analysis, ambiguity detection, debugging, error reporting and recovery, etc.

One of the most researched and used deterministic parsing algorithms, and probably still one of the best we have today is LR. The LR algorithm is conceived by Donald Knuth in his seminal paper [14]. The approach was impractical at first, due to the size of LR(1) automata, until DeRemer [15] described the practical algorithm for constructing LR(1) tables whose size was of LR(0).

In LR parsing, the parser operates as a state machine that keeps a stack of (non-)terminals that have been seen[3] so far and have access to zero or more next tokens of the input.[4] In each state, the parser decides which of two actions to perform, shift – taking the next token from the input and moving it to the stack, or reduce – replacing elements from the top of the stack with a non-terminal based on the matching grammar production. The decision is deterministic. There can be only one action that can be performed in each state. If more than one action is possible we have a conflict and the grammar is not accepted. All possible actions are pre-calculated statically and stored in the parsing table before the parser runs. This up-front table calculation allows the parser to achieve good run-time performance.

LR variants are widely used in practice. There are a variety of libraries for Python. PLY,[5] and its successor SLY,[6] are pure Python implementations of the LR parsing algorithm, using LALR(1) tables. They are modeled after traditional lex/yacc tools. A grammar in both tools is specified in Python, using functions, docstrings, decorators, and other Python language constructs. Another interesting LR parsing library for Python is Lark.[7] It is relatively new Python library that has gained significant popularity lately. Interesting features are: support for generalized parsing by implementing Earley [16] and CYK [17] algorithms besides deterministic LALR(1), EBNF-like language for grammar specification, and automatic building of AST based on the grammar structure and in-grammar rules.

GLR is a generalized extension of the LR parsing algorithm proposed by Tomita in [18]. Its development was motivated by natural language parsing. GLR allows for conflicts in LR tables. At each state where multiple possible actions are possible GLR algorithm forks the parser stack to investigate all possible paths. At some point, different parser stacks may again re-merge. To maintain the parser state, GLR uses a multiply-rooted directed acyclic graph, which Tomita called graph-structured stack (GSS). An original Tomita's GLR algorithm is not in fact general. It fails to terminate on grammars with hidden left recursion. This flaw was fixed by Farshi in [19] although at a great cost as in order to take all ϵ reduction into account his correction implements a constant re-processing of already processed parser stacks when a new path is introduced to take into account potentially missed reductions [20,21]. Nevertheless, Farshi's correction gained in popularity and is later incorporated into Rekers' [22] and Visser's work [23].

A favorable property of GLR is that its performance approaches linear when the grammar approaches LR(1) and gracefully degrades to polynomial for non-LR(1) grammars. However, Johnson [24] reported that any parser which returns a Tomita style Shared Packed Parse Forest (SPPF) will be inherently of $O(n^{k+1})$ complexity where k is the length of the longest rule. Thus, transforming the grammar to Chomsky normal form would yield $O(n^3)$ but would destroy its structure. Kipps suggested an approach to achieve $O(n^3)$ but with high constant costs [25]. Later research introduced an improved version of GLR. Aycock et al. [26] introduced a GLR version with fewer stack operations if the grammar does not contain right and hidden left recursion. Scott and Johnstone developed a variant of GLR called Right Nulled GLR (RNGLR) [27] based on short-circuiting of right nulled productions, which corrects the original Tomita's algorithm without the cost of searching required by Farshi's fix. This approach yields much better performance. However, it requires the introduction of new entries in the LR tables and has the same worst-case complexity. In the later work, a more efficient version called Binary Right Nulled GLR (BRNGLR) is introduced, which achieves worst-case cubic run time on all grammars and a cubic size binary SPPF representation of all derivations [28].

More recent work in generalized parsing is focused on a new top-down generalized approach called GLL [29,30]. This approach is promising as it accepts a full set of CFGs, runs in cubic time while retaining the close relationship with the grammar, that is attributed to recursive descent-like approaches, leading to easier debugging and understandability. Afroozeh and Izmaylova [31] showed how a GLL can be further optimized by the implementation of a more efficient Graph-Structured Stack. Since GLL is a top-down approach it can be applied in the context of parser combinators which is investigated in [32,33].

In the Java ecosystem, there are several notable implementations of GLR. ASF+SDF implements a scannerless GLR (SGLR) based on Syntax Definition Formalism (SDF) meta-syntax [34]. The main feature of SDF that it is highly modular and declarative. It enables easy language composition and embedding. Ambiguity resolution is specified using highly expressive filters

[2] For example, LR parsing is based on a state-machine that does not have an apparent correspondence with the language grammar.
[3] Actually, the parser keeps automaton states on the stack, but that implementation detail is not relevant for the discussion.
[4] Usually just one, thus LR(1).
[5] https://github.com/dabeaz/ply.
[6] https://github.com/dabeaz/sly.
[7] https://github.com/lark-parser/lark.

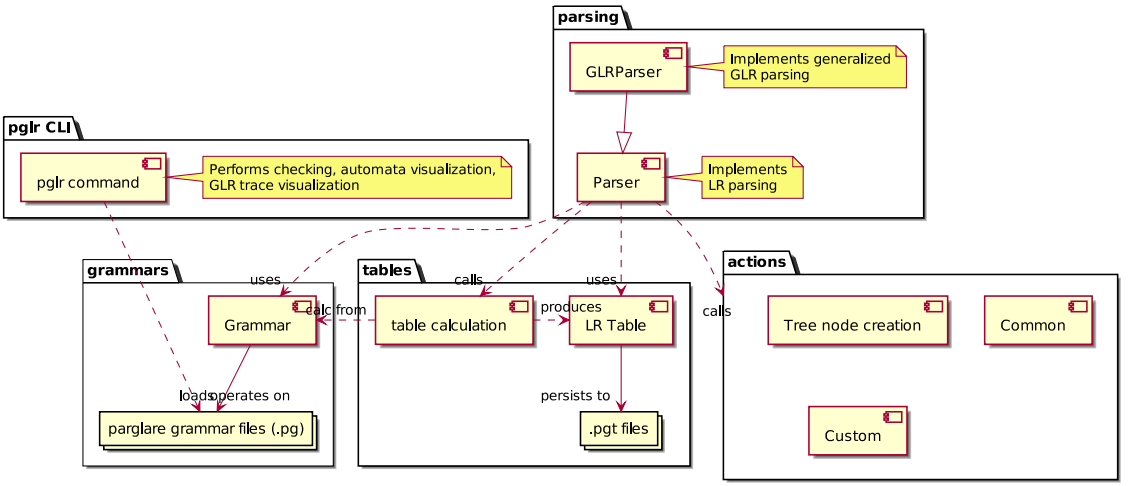

Fig. 1. The parglare architecture.

[6]. SDF has found usage in several other toolsets, most notably Stratego/XT, a DSL and a toolset for program transformation [35]. ASF+SDF is not maintained anymore and the focus of the team has shifted to the Rascal project [36].

Elkhound [37] is a GLR parser generator that can emit the parser code in C++ and OCaml languages. During parsing, Elkhound dynamically switches to LALR parsing on parts of inputs that are non-ambiguous thus providing better performance for languages that are mostly deterministic while still handling ambiguity with GLR.

For Python, there is the GLRParser project[8] which is oriented towards natural language processing. It has an engine for syntax-based translation of natural languages. The project seems to be actively maintained as of this writing. However, according to the project tests and documented usage patterns, it seems that it is specifically targeted towards the translation of natural languages. Another GLR implementation is jupyLR.[9] The algorithm for automata construction is SLR, which is known to be less powerful than LALR(1). It uses a separate scanner and a grammar given in a BNF-like notation. It seems that the project is not maintained anymore.

As PEG-based parsers are easy to write and debug, they attracted a lot of attention spurring a multitude of parsing tools. PyParsing [38] is a well-known Python parsing library based on PEGs. The grammar is defined using Python language constructs by overloading of Python operators. Arpeggio [39] is another PEG library where grammar is defined by either using Python language or by a textual PEG syntax. TextX [40] is a library that builds on top of Arpeggio and provides additional facilities geared towards Domain-Specific Language development. All these libraries provide good error reporting and are easy to debug being a top-down recursive-descent parsers.

In the rest of the paper, we describe parglare – a pure Python parsing library based on LR and GLR algorithms.

3. Software framework

3.1. Software architecture

The parglare library consists of several components (Fig. 1). The `tables` module deals with LR parser tables calculation and persistence. The `grammars` module deals with grammars definition and persistence. The `actions` module defines common semantic actions and provides means for custom actions definition. The `parsing` module implements parsing algorithms where the `Parser` class implements LR and inheriting `GLRParser` class implements GLR algorithms respectively. And finally, `pglr` is a CLI command supporting various grammar development tasks (Section 3.2.3).

3.2. Software functionalities

3.2.1. Integrated scanner

Having a scanner (a.k.a. lexer) operating as a separate stage to syntax analysis poses a number of restrictions on syntax definitions. A separate scanner cannot consider the context in which a token occurs which may lead to a problem where lexer cannot decide which token to create. A common solution to this problem is to make a feedback loop from the parser to the lexer introducing a lexer state.[10]

[8] https://github.com/mdolgun/GLRParser.
[9] https://github.com/bl0b/jupyLR.
[10] This technique is often referred to as a “lexer hack”.

In parglare, the scanning phase is not implemented as a separate step. It is integrated into parsing and the parser calls "recognizers" when the next token from the input stream is needed. Recognizers are Python callables[11] which are in charge of recognizing tokens from the input stream. The parser knows which tokens may appear at the given location so a problem described in the previous paragraph is a non-issue for this approach.[12] Built-in recognizers are `string` and `regex`, recognizing plain strings and regular expressions respectively. Recognizers are provided for grammar terminal definitions.

3.2.2. Generalized LR parsing - GLR

Parglare provides information on where non-determinism[13] in grammar lies and gives detailed information on why that happens. In case a language needs non-deterministic parsing, one can resort to the GLR algorithm by instantiating `GLRParser` class instead of `Parser`. The grammar and the calculated table stay the same.

In the case of non-determinism, the GLR parser will fork and investigate each possibility. Eventually, parsers that decided wrong will die, leaving only the right one. In case there are multiple interpretations of your input, you will get all possible trees (a.k.a. "the parse forest").

3.2.3. pglr command

Parglare provides `pglr` CLI command which can be used for various development tasks. Currently, it is used to check and compile the grammar, visualize the LALR automaton, visualize GLR parse trace, parse the given input and present parse forest/trees as an indented string or in a graphical form using `dot`.

3.2.4. Actions and recognizers

The purpose of parsing is to transform input into some usable representation. This is usually done by executing a specific action when a grammar rule matches. There are several options for that:

- by default, the LR parser builds nested lists;
- a parse tree can be built by passing `build_tree=True` parameter to the parser (applicable to LR parsing as GLR always builds SPPF);
- call user-supplied actions – by writing Python functions that are called when rules match. Each function can perform arbitrary computation, and the result returned is used in parent rules/actions. There are some handy built-in actions in the `parglare.actions` module;
- user actions may be postponed and called on the parse tree – this is useful if the tree has to be processed in multiple ways or if the GLR parsing is used.

Actions (a.k.a. *semantic* or *reduction actions*) are Python callables that get called to reduce the recognized pattern to some higher-level concept. The action is of the following form:

```
1  def some_action(context, nodes):
2      # Do some calculation and return result
```

where `context` is the parser context object used to query various parser state information, and `nodes` is the list of results from the lower-level actions. Parglare provides a so called *action collector*, that is used to keep track of and collect all parser actions. It is constructed by a `parglare.get_collector` function and is used as a Python decorator:

```
1  from parglare import get_collector
2
3  action = get_collector()
4
5  @action
6  def number(_, value):
7      return float(value)
8
9  @action('E')
10 def sum_act(_, nodes):
11     return nodes[0] + nodes[2]
```

Module `parglare.actions` provides a library of some commonly used actions.

[11] Objects with the call semantics in Python, most commonly functions or lambdas.

[12] Although, lexical ambiguity still may happen between lexemes that can match at the same place according to the grammar.

[13] Either the parser needs additional lookahead to decide, or the language is inherently ambiguous. It boils down to having conflicts in LR tables, by which the parser may end-up in a state where multiple actions are possible.

Recognizers are callables used to recognize a token in the input. Parglare has two built-in recognizers for textual parsing that can be specified in the grammar directly. Those are usually enough if text is parsed, but if a non-textual content is parsed custom recognizers need to be registered that are able to recognize tokens in the input stream of objects.[14]

Recognizers are Python callables of the following form:

```
def some_recognizer(context, input, pos):
    ...
    ...
    return part of input starting at pos
```

where `context` is the same parser context object as used for actions, `input` is the list of input objects,[15] and `pos` is the current position in the input stream. The recognizer should return the part of the input that matches.

3.2.5. Disambiguation

Parglare offers a standard mechanism for removing ambiguities from the grammar in a declarative style. There are generally two sources of ambiguity. The first is syntactical ambiguity where a string of tokens can derive different trees. The second is lexical ambiguity when multiple tokens can be recognized at the same location.

Some forms of ambiguity can be resolved statically before the parser run, while others must be done dynamically at run-time. Static disambiguation rules in parglare are priorities and associativities. When provided, they will be used during LR table calculations to remove conflicts. Disambiguation rules are specified declaratively using annotations applied inside of curly braces,[16] either at production or a rule level. However, not all ambiguities can be handled with these two rule types [7]. For more complex cases parglare offers a pluggable dynamic filter.[17] Unfortunately, with this approach, we lose declarativity. In future versions, we plan to work on more powerful dynamic disambiguation filters.

Furthermore, parglare offers a "prefer shifts" global strategy for shift-reduce conflict resolution that is utilized by many classic LR parsing tools. This strategy always chooses `shift` operation over `reduce` leading to a greedy behavior. However, it can be disabled on a per-production or per-rule basis using `nops` meta-data flag.

In the following example, we see a lexical ambiguity between integer and float.

```
Stuff: Stuff "+" Stuff | Something;
Something: INT | FLOAT | Object;
Object: INT "." INT;

terminals
INT: /\d+/;
FLOAT: /\d+(\.{\d}+)?/;
```

`INT` and `FLOAT` rules overlap and due to the `Object` rule both interpretations succeed even if `FLOAT` token contains a decimal separator. The `Object` rule can match the same part of the input as the `FLOAT` lexical rule. If no disambiguation is provided, parglare will use GLR machinery to handle lexical ambiguities as well by splitting each frontier by recognized token and re-merging heads on the part of the input where they synchronize.

LR parser will use the default lexical disambiguation strategy. GLR does not use this strategy by default but forks instead. The strategy is to choose the longest matched token (e.g. FLOAT instead of INT). If multiple tokens have the same size, a more specific match is chosen (e.g. string match over regex match). If this does not help then a token marked with `prefer` is used. If we still have a lexical ambiguity the LR parser will throw an exception in this case.

3.2.6. Support for whitespaces

By default, whitespaces are skipped. This is controlled by the `ws` parameter to the parser constructor which is by default set to `"\t\n"`. If set to `None` no whitespace skipping is performed. If there is a rule `LAYOUT` in the grammar this rule is used instead. An additional parser with the grammar defined by the `LAYOUT` rule will be built to handle whitespaces.

3.2.7. Error reporting and error recovery

The LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input [41]. There are multiple aspects of the problem. The first aspect is error reporting during grammar parsing and the LALR table

[14] Test `tests/func/recognizers/test_recognizers.py` is an example of parsing Python list of integers.

[15] For example, characters if text is parsed.

[16] Generally, curly braces are used to specify arbitrary meta-data for productions and rules. These can later be used during custom disambiguation and grammar analysis. The details can be found in the "Grammar language/User meta-data" section of the documentation.

[17] See "Disambiguation" section of the documentation.

construction process. The second is error reporting during the parsing of the given input according to the provided grammar. A further consideration is error recovery.

Parglare works like an interpreter, i.e. it does not generate parser code but rather configures itself with the provided grammar. The bootstrapping process is done by parglare grammar description given in the form of Python structures in module `parglare/grammar.py`. The bootstrapping produces a parglare parser capable of parsing parglare grammars. Thus, the same error reporting facilities are used for both grammar parsing and parsing of the user input.

When a syntax error is found, parglare throws `ParseError` exception with the information about tokens expected at the place of the error, symbols seen previously, and the last parsing heads. A naive approach, by reporting all tokens expected according to the LR table, could yield tokens that cannot be found at the given location due to LR automata compression introduced by the LALR algorithm. Thus, during the error reporting phase, the parser will try to do all pending reductions for each possible token from the table. Only those tokens which could be shifted at the end of the reduction process are the tokens that could continue parsing and those tokens are reported.

When GLR parsing is used, multiple parse heads may be active at any given time. The error occurs when no parse head can shift the next token. Although, different parse heads have a different interpretation of already seen input, currently parglare still reports the union of all expected tokens from all parse heads.

When LALR tables are constructed, if debug mode is active, a detailed report on the automaton states, actions, and conflicts is given to the user. Conflict messages are descriptive and give possible ways to resolve. For example:

```
In state 12:E and input symbol '*' can't decide
    whether to shift or reduce by
    production(s) '4: E = E / E'.
There are 16 Shift/Reduce conflicts.
Either use 'prefer_shifts' parser mode,
try to resolve manually, or use GLR parsing.
```

Although this provides information about the conflict, it is often hard to understand when these conflicts trigger and how are they related to the input. A possible improvement would be to implement a generator of counterexamples [42].

Error recovery is something that is often lacking in parsing libraries. It is a desired feature in building IDE support where we need to report all errors in the file in one go. Parglare has a built-in error recovery strategy which is currently a simplistic one – it will skip the current character until it is able to continue.[18] However, the error recovery is extensible and a custom strategy can be registered. A strategy is a Python function that will be called, when an error occurs, with the error information and the parser head (or heads in the GLR mode). Its job is to modify the parser head and bring it into a state where it can continue.[19]

The following example shows both error reporting and the default error recovery. We use here `pglr parse` command to parse erroneous input `"3 /& & & 6 a + 3"` using the grammar of simple algebraic expressions introduced in Section 5).

```
$ pglr parse calc.pg -i "3 *& & & 6 + * 3" -glr -recovery
Solutions:2
Ambiguities:1
Errors: 2
     Error at 1:3:"3 * **> & & & 6 + "
           => Expected: ( or Number
     Error at 1:13:"& & & 6 +  **> * 3"
           => Expected: ( or Number but found <*(*)>
Printing the forest:
E - ambiguity[2]
1:E[0->16]
  E[0->10]
    E[0->1]
      Number[0->1, "3"]
    *[2->3, "*"]
    E[9->10]
      Number[9->10, "6"]
  +[11->12, "+"]
  E[15->16]
    Number[15->16, "3"]
...
```

[18] This strategy is often referred to as the “panic mode”.

[19] The more details are provided in the “Handling errors” section of the documentation.

We can see that two errors are reported (lines 5 and 7 above). The `"**>"` marker denotes the current position that the parser is scanning. The first error is at column 3 at the first occurrence of `"&"`. The default recovery strategy will skip characters of the input until the parser can continue and that happens at the occurrence of character `"6"`. The second error is reported at the possition of character `"*"` as the operation `"+"` has already been found and the parser expects to see a number or an open parenthesis.

4. Implementation

4.1. GLR

The current GLR implementation in parglare is based on the work of Farshi [19] and Rekers [22]. This algorithm is believed to be correct although it has been demonstrated not very efficient [20]. It correctly handles grammars with hidden left recursion and enables handling of cyclic grammars by introducing loops in the produced SPPF.

The run-time cost of the Farshi's approach is due to demand for brute-force GSS re-search when new paths are introduced during the reductions of the current frontier. To prevent unnecessary GSS walk parglare keeps a dictionary where the keys are states of heads traversed by processing head from the set given as a value. The dictionary is consulted for heads to reprocess if a new parent link is introduced in GSS. This guarantees that only those heads that will traverse the given parent link will be reprocessed.[20]

Further extension is the support for lexical ambiguity as there can be many interpretations of the content that lies ahead so the GLR frontier must be split to handle different lexemes. GLR heads are synchronized by position and can again re-merge thus fully utilizing the GSS structure to handle lexical ambiguity as well. These ideas are inspired by Visser's work [23], although his work goes further by defining grammars down to a character level.

4.2. Graph structured stack (GSS)

GSS structure is based on Tomita's work. It comprises of instances of `GSSNode` class. `GSSNode` represents parser heads and keeps track of state, position, token ahead and a set of `Parent` instances which models the back-links in the GSS structure. `Parent` instance has a reference to the parent GSS node and a set of possible solutions at that point. These `Parent` instances are a part of the resulting SPPF.

4.3. Shared packed parse forest (SPPF)

During the GLR parsing, a Shared Packed Parse Forest (SPPF) is constructed as proposed by Tomita and later improved by Rekers. SPPF is implemented combining the `Parent` class which models the back-links in GSS with `NodeTerm` and `NodeNonTerm`, which represent terminal and non-terminal nodes in the parse tree, respectively. The `Parent` class has a list of possibilities and thus represents the ambiguity.

SPPF behaves like a standard Python collection. It can be iterated, yielding enumerated trees. The length of the forest is the number of derivations/trees.

```
1  forest = GLRParser(grammar).parse(some_input)
2  print(len(forest))          # number of trees
3  print(forest.ambiguities)   # number of ambiguities
4  print(forest.to_str())      # string representation
5  print(forest[5].to_str())   # index access. Get sixth tree.
6  for tree in forest:         # iteration
7      # each tree is an instance of LazyTree
8      print(tree.to_str())
```

Trees are lazy by default. They are constructed during access/iteration of their nodes. For easier processing of forests and trees, parglare provides generic `visitor` for depth-first traversal and processing of tree-like structures.

4.4. Table calculation

LR parsing is a deterministic table-driven approach. It corresponds to a deterministic push-down automaton (DPDA) where in each state, depending on the next token[21] it has exactly one action to perform.

LR table calculation is done in the `tables` module. The input to the calculation is the `Grammar` object, and the result is `LRTable` which encodes the LR state machine transitions. The table is cached in a file with `.pgt` extension and reloaded

[20] See `_states_traversed` dictionary in the `glr.py` module.

[21] If it uses lookahead.

Table 1
Grammars and LALR(1) automata sizes.

	Productions	Non-terminals	Automaton size
JSON	19	9	26
BibTeX	29	17	51
Java	795	405	1,190

Table 2
Input file sizes (in bytes).

	I_1	I_2	I_3	I_4	I_5	I_6
JSON	12,447	24,892	49,782	99,562	199,122	398,243
BibTeX	1,727	3,455	6,911	13,823	27,647	55,295
Java	2,189	8,018	15,471	27,600	52,802	70,084

in future parser runs. Table caching is transparent for the user. If the `.pgt` file is found it is used, if not the table calculation is performed and the cached file is created. The table calculation can be done in advance by using the `pglr compile` CLI command (Section 3.2.3).

The table calculation can result in states containing multiple transitions. These states are said to have conflicts which can be either: (a) *shift-reduce* where the parser cannot decide if it needs to consume the next token from the input (`shift` operation) or reduce what it saw so far, or (b) *reduce-reduce* where parser cannot decide by which production rule to do the reduction of the top of the stack.

Since the LR parsing is deterministic, conflicting states cannot be handled. On the other hand, the GLR algorithm allows conflicting states. Intuitively, when conflicting operations compete, GLR will fork and investigate all possibilities. Thus, parglare allows for conflicts in LR tables but will allow only GLR parser to run for such tables.

4.5. Performance evaluation

This section provides the result of the performance evaluation.

4.5.1. Test setup

The test is performed on 3 grammars and 6 inputs of increasing sizes. The grammars are JSON, BibTeX, and Java SE 16.

JSON and BibTeX grammars are LALR(1), and thus results for both LR and GLR are reported. The Java grammar is created directly from the language specification without any transformation to eliminate conflicts. LALR table for this grammar has multiple shift-reduce and reduce-reduce conflicts. Thus, it is not LALR(1) and only GLR results are reported. Grammars and LALR(1) automata sizes (number of states) are reported in Table 1.

Input files for JSON and BibTeX are doubling in size. For Java, a set of files of increasing size, from 2.19 KB to 70.08 KB, has been taken from the Spring Boot project.[22] File sizes are given in Table 2.

The test is performed on the following hardware configuration: Lenovo X1 Carbon 16 GB RAM, Intel Core i7-8550U CPU @ 1.80GHz. Operating system is a GNU/Linux distribution, kernel version 5.12.3.

To get more consistent results, Python garbage collector and all unnecessary OS services were disabled during testing. Inputs are preloaded into memory before the parsing begins.

The whole test setup, including test scripts and inputs, are available at the project GitHub repository, folder `tests/perf/`. All tests can be run by the `runall.sh` script. The results are stored in the `tests/perf/reports/` folder.

4.5.2. Results

Table 3 shows the results for all 3 grammars and all 6 inputs. For the first two grammars, the throughput does not change much with the increase in the input size. This result is expected as LR parsing is linear-time and GLR should be linear for deterministic grammars and should degrade gracefully as the grammar degrades from determinism. It can be observed from the results that, for these two grammars, GLR is ~1.8x slower due to the overhead of the more complex algorithm and maintaining more complex data structures.

The Java grammar is ambiguous and more complex, so the lower performance is not unexpected. The number of ambiguities in Java input files are ranging from 53 for the smallest input file to 1987 for the largest input file.

Memory utilization, given in Table 4, is measured using Python built-in `tracemalloc` module. The results show that the GLR has a significantly larger memory consumption due to the maintenance of more complex data structures. This should be taken into account if parglare is used in memory-constrained environments.

[22] The Spring Boot project - https://github.com/spring-projects/spring-boot.

Table 3
Parsing run-time performance results. The results are given in throughput (bytes/sec). `Java ambig` row is the number of ambiguities for Java test files.

	I_1	I_2	I_3	I_4	I_5	I_6
JSON LR	746,728	770,605	766,252	771,167	761,376	759,707
JSON GLR	413,838	412,225	411,039	408,479	408,531	401,523
BibTeX LR	456,845	525,634	530,546	517,580	515,125	512,579
BibTeX GLR	302,313	300,401	299,262	292,926	290,300	288,395
Java GLR	51,994	41,574	36,845	53,070	55,404	38,258
Java ambig	53	239	366	629	1,065	1,987

Table 4
Memory utilization (in KB).

	I_1	I_2	I_3	I_4	I_5	I_6
JSON LR	116	210	404	792	1,567	3,118
JSON GLR	2,485	4,945	9,891	19,786	39,593	79,206
BibTeX LR	36	64	120	233	458	909
BibTeX GLR	521	1,044	2,089	4,182	8,367	16,739
Java GLR	1,466	5,919	11,013	13,811	26,031	56,343

5. Illustrative example

In this section, an implementation of a simple language for algebraic expressions with two basic operations is analyzed. The language is well known, so there is no need for much explanation. It is small enough to fit into the constrained space of this publication but complex enough to demonstrate the overall workflow and some important parglare features. More complex examples can be found in the `examples` folder in the source code repository of the project.[23]

Parglare source code is hosted on GitHub, but it is also distributed through Python Package Index (PyPI)[24] so it can be easily installed using the standard Python `pip` package manager. When working with Python it is a good practice to use Python virtual environments to isolate a set of libraries used for a particular project. To create a virtual environment and install parglare the following commands can be used:

```
# Create folder venv with Python virtual enviromnet
$ python -m venv venv
# Activate the environment
$ source venv/bin/activate
# Install parglare from PyPI
(env) $ pip install parglare
```

5.1. Calc grammar

In Listing 1, a grammar for the language is given. We can write the grammar in a separate file or directly inside Python scripts as strings. The advantage of having the grammar written in a separate file is that the LALR tables will be transparently cached in a file with the `.pgt` extension and loaded on successive runs. Another advantage is that `pglr` commands for compiling, parsing, and visualizations can be run on the grammar if written in a separate file.

```
E: E '+' E
 | E '*' E
 | '(' E ')'
 | number;

terminals
number: /\d+/;
```

Listing 1: An ambiguous algebraic expression grammar in parglare.

[23] https://github.com/igordejanovic/parglare/tree/master/examples.
[24] Python Package Index (PyPI) – https://pypi.org/.

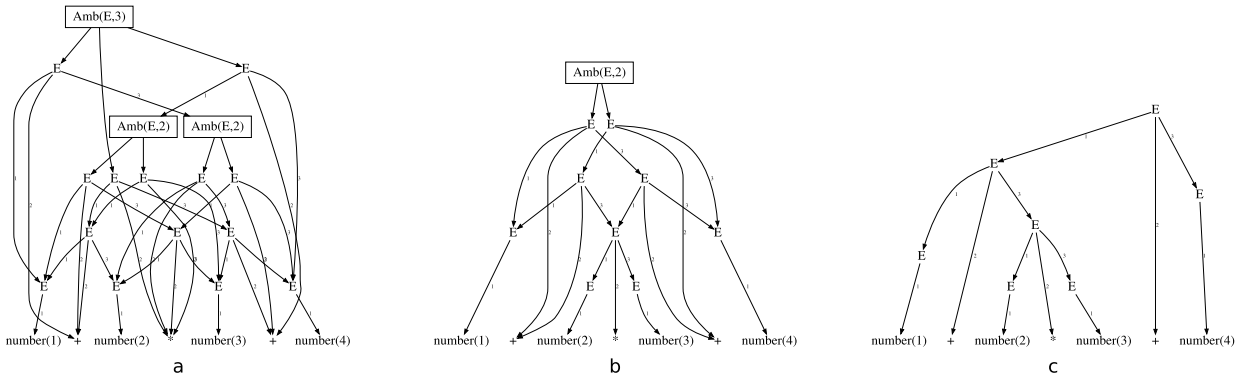

Fig. 2. The parse forest of an expression `1 + 2 * 3 + 4`: a) all 5 derivations when no disambiguation is applied, b) 2 derivations when priority is applied, c) a single derivation when both priorities and associativities are applied.

Grammar is instantiated using the `Grammar` class by calling either `from_file` or `from_string` factory class methods.

```
from parglare import Grammar
grammar = Grammar.from_file('calc.pg')
```

This grammar enables parsing of sentences like `1 + 2 * 3 + 4`. But, even this simple grammar is ambiguous. The given sentence yields the parse forest with three ambiguous nodes and 5 derivations (Fig. 2 (a)). The number of possible trees raises quickly with the length of the input sentence.[25] The grammar is ambiguous, but the language we are trying to describe is not. We know that the operations have two properties that resolve ambiguity: priority and associativity. One approach in handling operations priority is by encoding the priority in the grammar productions, but we lose the declarativity and conciseness of the natural grammar in Listing 1 [5].

The priority is defined as an integer number where productions annotated with a greater number are of greater priority. If we apply the priority rules by specifying a number inside of curly braces we get two possible derivations shown in Fig. 2 (b). We can see that the `*` operation is of higher priority and will be lower in the tree in both derivations. However, since the associativity for `+` operation is not defined we get both solutions.

The same grammar with both disambiguation annotations applied is given in Listing 2. Both operations are left-associative. In terms of the GLR/LR parsing approach, left associativity means that the parsing table will have a `reduce` operation if the given operation production is competing with the same production ahead. Fig. 2 (c) shows that when both associativities and priorities are defined we get a single tree as a result.

```
E: E "+" E {left, 1}
 | E "*" E {left, 2}
 | "(" E ")"
 | Number
;
terminals
Number: /\d+/;
```

Listing 2: The grammar from Listing 1 with declarative disambiguation rules (priorities and associativities).

5.2. Connecting actions

To provide the transformation of the input string into the desired form we attach semantic actions to grammar productions (Section 3.2.4).

In this example, our transformation is calculating the result according to the usual semantics of the given two operations. Thus, as the result of the parsing, we want to get the result of the expression. To do so, we attach actions to the grammar rules and production as given in Listing 3. We see that there are four actions for rule `E` and one action for rule `number`. Each of the four actions under the dictionary key `E` corresponds to `E` grammar productions in the same order.

[25] The number of possible parse trees for the given number of operations can be calculated by an integer series known as the Catalan numbers [43].

```
actions = {
    "E": [lambda _, n: n[0] + n[2],
          lambda _, n: n[0] * n[2],
          lambda _, n: n[1],
          lambda _, n: n[0]],
    "number": lambda _, value: float(value),
}

g = Grammar.from_file('calc.pg')
parser = Parser(g, actions=actions)
result = parser.parse("34 + 4 * 2 + 4")
```

Listing 3: Connecting actions to grammar rules.

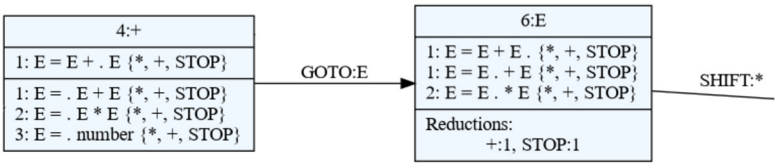

Fig. 3. A part of the state-machine diagram produced by `pglr viz` sub-command for the grammar given in Listing 2.

For example, rule:

```
lambda _, n: n[0] + n[2]
```

corresponds to the production `E: E "+" E`. It has an underscore at the place of `context` as a suggestion that we are not using this parameter, and `n` at the place of `nodes`. Our `nodes` will be the results of the sub-trees lower in the hierarchy. We know that the first production for rule `E` in our grammar defines the `+` operation, so we return the result of Python summation.

Provided actions are applied for each sub-pattern, and at the end of the parsing, the `result` variable from Listing 3 will hold the value of the given expression.

The further information can be found in Section "Actions" in the parglare documentation.

5.3. Visualization

`pglr` offers `viz` sub-command that can be used to produce a *dot* diagram of LR state machine. `dot` is a graph drawing language, and is a part of GraphViz software package.[26]

```
(venv)$ pglr viz calc.pg
Grammar OK.
Generating 'calc.pg.dot' file for the grammar PDA.
Use dot viewer (e.g. xdot) or convert to pdf by running
  'dot -Tpdf -O calc.pg.dot'
```

In Fig. 3 we see a part of the diagram of the automaton visualization constructed from the grammar given in Listing 2. The full diagram shows all parser states with detailed information about each state and the possible transitions, i.e. possible shifts and gotos.

For learning purposes, parglare `pglr trace` command provides visualization of GLR parsing process. For example, executing:

```
$ pglr trace -i "1+2+3" calc.pg
```

where the file `calc.pg` contains the grammar from Listing 2, will produce the trace given in Fig. 4.

[26] Graphviz is an open source graph visualization software – https://graphviz.org/.

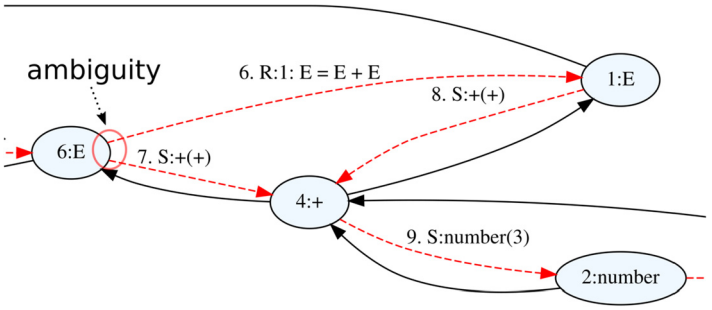

Fig. 4. A part of GLR trace visualization for input `1+2+3` and ambiguous grammar from Listing 1 produced by `pglr trace` command. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

The dashed red lines represent automaton actions that either can be `S - shift` or `R - reduce`. Shift operations specify a token recognized at the current position, while reduce operations specify the production used. The solid black arrows together with the elliptic shapes represent the edges and nodes of the GSS respectively.

We can see in Fig. 4 that at state `6:E` at the left edge of the Figure there is an ambiguity as two actions can be performed: (1) a reduce action `6.R:1: E=E+E`, and (2) a shift action `7.S:+(+)`. This is the place where the GLR graph-structured stack forks investigating both paths. In the first step, a reduction is performed reducing node `6:E` and the previous two nodes[27] using production `E:E+E` to state `1:E`. In the second step, a shift operation is performed taking token `+` from the input and moving automaton to state `4:+`.

The `pglr` command can be used to parse input files or strings and represent the parse forest/trees as either an indented string or as a dot graph. Fig. 2 in this paper is produced using this command. For further information see `pglr parse` sub-command.

6. Conclusions

This paper introduces parglare, a pure Python LR/GLR parser. Parglare is an open-source project hosted at GitHub under the MIT license. The purpose of the project is to provide an easy to use Python implementation of a well-known LR algorithm and its generalized GLR extension. A well-written documentation and a comprehensive test suite are provided to ensure long-term development of the project.

Parglare is implemented in Python and operates in an interpreter style, i.e. the parser is not generated but configured by the grammar. This approach is well-aligned with the Python dynamic nature. Parglare is well suited for areas where Python is used and general parsing is needed, most notably in various fields of scientific computing where Python gained in popularity in recent years. Parglare is also well suited for use in educational settings due to its command line tools, reporting facilities, debugging, tracing, and visualization.

Further implementation directions will be oriented towards more efficient GLR variants (e.g. RNGLR, Elkhound style dynamic LR/GLR switching), the GLL algorithm, more powerful ambiguity detection and disambiguation approaches, and better error reporting and recovery strategies. A new interactive debugger with GSS/tree visualization is also planned.

At the time of this writing, parglare has been used for two years as a part of the Domain-Specific Languages course at master studies at Faculty of Technical Sciences, University of Novi Sad.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] D. Grune, Ceriel J.H. Jacobs, Parsing Techniques: A Practical Guide, 2nd edition, Monographs in Computer Science, Springer-Verlag New York Inc, 2007.
[2] E. Scott, A. Johnstone, Recognition is not parsing—SPPF-style parsing from cubic recognisers, Sci. Comput. Program. 75 (1–2) (2010) 55–70.
[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, The Java® language specification, Tech. Rep., Java SE 16 edition, 2021.
[4] A. Johnstone, E. Scott, Generalised recursive descent parsing and follow-determinism, in: International Conference on Compiler Construction, Springer, 1998, pp. 16–30.
[5] L.C. Kats, E. Visser, G. Wachsmuth, Pure and declarative syntax definition: paradise lost and regained, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010, pp. 918–932.
[6] M.G. Van den Brand, J. Scheerder, J.J. Vinju, E. Visser, Disambiguation filters for scannerless generalized LR parsers, in: International Conference on Compiler Construction, Springer, 2002, pp. 143–158.

[27] Previous two nodes `E` and `+` are not visible in the Figure due to space constraints.

[7] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, J. Vinju, Safe specification of operator precedence rules, in: International Conference on Software Language Engineering, Springer, 2013, pp. 137–156.
[8] D.G. Cantor, On the ambiguity problem of Backus systems, J. ACM 9 (4) (1962) 477–479, ambiguity undecidable.
[9] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, ACM SIGPLAN Not. 39 (1) (2004) 111–122.
[10] L. Diekmann, L. Tratt, Eco: a language composition editor, in: International Conference on Software Language Engineering, Springer, 2014, pp. 82–101.
[11] A. Warth, J. Douglass, T. Millstein, Packrat parsers can support left recursion, in: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, 2008, pp. 103–110.
[12] S. Medeiros, F. Mascarenhas, R. Ierusalimschy, Left recursion in parsing expression grammars, Sci. Comput. Program. 96 (2014) 177–190.
[13] A. Aho, J. Ullman, The Theory of Parsing, Translation, and Compiling, Series in Automatic Computation, vol. 1, Prentice-Hall, 1972.
[14] D.E. Knuth, On the translation of languages from left to right, Inf. Control 8 (6) (1965) 607–639.
[15] F.L. DeRemer, Practical translators for LR(k) languages, Massachusetts Institute of Technology, 1969.
[16] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (2) (1970) 94–102.
[17] D.H. Younger, Recognition and parsing of context-free languages in time n3, Inf. Control 10 (2) (1967) 189–208.
[18] M. Tomita, An efficient context-free parsing algorithm for natural languages, in: IJCAI, vol. 2, 1985, pp. 756–764.
[19] R. Nozohoor-Farshi, GLR parsing for ε-grammers, in: Generalized LR Parsing, Springer, 1991, pp. 61–75.
[20] A. Johnstone, E. Scott, G. Economopoulos, Generalised parsing: some costs, in: International Conference on Compiler Construction, Springer, 2004, pp. 89–103.
[21] A. Johnstone, E. Scott, G. Economopoulos, The grammar tool box: a case study comparing GLR parsing algorithms, Electron. Notes Theor. Comput. Sci. 110 (2004) 97–113, https://doi.org/10.1016/j.entcs.2004.06.008.
[22] J.G. Rekers, Parser generation for interactive environments, Ph.D. thesis, Citeseer, 1992.
[23] E. Visser, Syntax definition for language prototyping, Ph.D. thesis, University of Amsterdam, 1997.
[24] M. Johnson, The computational complexity of GLR parsing, in: Generalized LR Parsing, Springer, 1991, pp. 35–42.
[25] J.R. Kipps, GLR parsing in time $O(n^3)$, in: Generalized LR Parsing, Springer, 1991, pp. 43–59.
[26] J. Aycock, N. Horspool, J. Janoušek, B. Melichar, Even faster generalized LR parsing, Acta Inform. 37 (9) (2001) 633–651.
[27] E. Scott, A. Johnstone, Right nulled GLR parsers, ACM Trans. Program. Lang. Syst. 28 (4) (2006) 577–618, https://doi.org/10.1145/1146809.1146810.
[28] E. Scott, A. Johnstone, R. Economopoulos, BRNGLR: a cubic Tomita-style GLR parsing algorithm, Acta Inform. 44 (6) (2007) 427–461.
[29] E. Scott, A. Johnstone, GLL parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189.
[30] E. Scott, A. Johnstone, GLL parse-tree generation, Sci. Comput. Program. 78 (10) (2013) 1828–1844.
[31] A. Afroozeh, A. Izmaylova, Faster, practical GLL parsing, in: CC 15, 2015, pp. 89–108.
[32] A. Izmaylova, A. Afroozeh, T.v.d. Storm, Practical, general parser combinators, in: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2016, pp. 1–12.
[33] L.T. van Binsbergen, E. Scott, A. Johnstone, GLL parsing with flexible combinators, in: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, 2018, pp. 16–28.
[34] E. Visser, Scannerless generalized-LR parsing, 1997.
[35] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, in: Special Issue on Second Issue of Experimental Software and Toolkits (EST), Sci. Comput. Program. 72 (1) (2008) 52–70, https://doi.org/10.1016/j.scico.2007.11.003, https://www.sciencedirect.com/science/article/pii/S0167642308000452.
[36] P. Klint, T. Van Der Storm, J. Vinju, RASCAL: a domain specific language for source code analysis and manipulation, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2009, pp. 168–177.
[37] S. McPeak, G.C. Necula, Elkhound: a fast, practical GLR parser generator, in: International Conference on Compiler Construction, Springer, 2004, pp. 73–88.
[38] P. McGuire, Getting Started with Pyparsing, O'Reilly Media, Inc., 2007.
[39] I. Dejanović, G. Milosavljević, R. Vaderna, Arpeggio: a flexible PEG parser for Python, Knowl.-Based Syst. 95 (2016) 71–74, https://doi.org/10.1016/j.knosys.2015.12.004, http://www.sciencedirect.com/science/article/pii/S0950705115004761.
[40] I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković, TextX: a Python tool for domain-specific languages implementation, Knowl.-Based Syst. 115 (2017) 1–4, https://doi.org/10.1016/j.knosys.2016.10.023, http://www.sciencedirect.com/science/article/pii/S0950705116304178.
[41] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd edition, Addison Wesley, 2006, http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&amp, http://www.bibsonomy.org/bibtex/2c3bc558a481f3cf0445068988a80f1b2/earthfare.
[42] C. Isradisaikul, A.C. Myers, Finding counterexamples from parsing conflicts, ACM SIGPLAN Not. 50 (6) (2015) 555–564.
[43] M. Crepinšek, L. Mernik, An efficient representation for solving Catalan number related problems, Int. J. Pure Appl. Math. 56 (4) (2009) 598–604.