Parsing Expression Grammars Made Practical

Nicolas Laurent *
ICTEAM, Université catholique de Louvain, Belgium
nicolas.laurent@uclouvain.be

Kim Mens
ICTEAM, Université catholique de Louvain, Belgium
kim.mens@uclouvain.be

Abstract

Parsing Expression Grammars (PEGs) define languages by specifying a recursive-descent parser that recognises them. The PEG formalism exhibits desirable properties, such as closure under composition, built-in disambiguation, unification of syntactic and lexical concerns, and closely matching programmer intuition. Unfortunately, state of the art PEG parsers struggle with left-recursive grammar rules, which are not supported by the original definition of the formalism and can lead to infinite recursion under naive implementations. Likewise, support for associativity and explicit precedence is spotty. To remedy these issues, we introduce Autumn, a general purpose PEG library that supports left-recursion, left and right associativity and precedence rules, and does so efficiently. Furthermore, we identify infix and postfix operators as a major source of inefficiency in left-recursive PEG parsers and show how to tackle this problem. We also explore the extensibility of the PEG paradigm by showing how one can easily introduce new parsing operators and how our parser accommodates custom memoization and error handling strategies. We compare our parser to both state of the art and battle-tested PEG and CFG parsers, such as Rats!, Parboiled and ANTLR.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Parsing

Keywords parsing expression grammar, parsing, left-recursion, associativity, precedence

1. Introduction

Context Parsing is well studied in computer science. There is a long history of tools to assist programmers in this task. These include parser generators (like the venerable Yacc) and more recently parser combinator libraries [5].

Most of the work on parsing has been built on top of Chomsky's context-free grammars (CFGs). Ford's parsing expression grammars (PEGs) [3] are an alternative formalism exhibiting interesting properties. Whereas CFGs use non-deterministic choice between alternative constructs, PEGs use prioritized choice. This makes PEGs unambiguous by construction. This is only one of the manifestations of a broader philosophical difference between CFGs and PEGs.

CFGs are generative: they describe a language, and the grammar itself can be used to enumerate the set of sentences belonging to that language. PEGs on the other hand are recognition-based: they describe a predicate indicating whether a sentence belongs to a language.

The recognition-based approach is a boon for programmers who have to find mistakes in a grammar. It also enables us to add new parsing operators, as we will see in section 4. These benefits are due to two PEG characteristics. (1) The parser implementing a PEG is generally close to the grammar, making reasoning about the parser's operations easier. This characteristic is shared with recursive-descent CFG parsers. (2) *The single parse rule*: attempting to match a parsing expression (i.e. a sub-PEG) at a given input position will always yield the same result (success or failure) and consume the same amount of input. This is not the case for CFGs. For example, with a PEG, the expression $(a*)$ will always greedily consume all the a's available, whereas a CFG could consume any number of them, depending on the grammar symbols that follow.

Challenges Yet, problems remain. First is the problem of left-recursion, an issue which PEGs share with recursive-descent CFG parsers. This is sometimes singled out as a reason why PEGs are frustrating to use [11]. Solutions that do support left-recursion do not always let the user choose the associativity of the parse tree for rules that are both left- and right-recursive; either because of technical limitations [1] or by conscious design [10]. We contend that users should be able to freely choose the associativity they desire.

Whitespace handling is another problem. Traditionally, PEG parsers do away with the distinction between lexing and parsing. This alleviates some issues with traditional lex-

* Nicolas Laurent is a research fellow of the Belgian fund for scientific research (F.R.S.-FNRS).

SLE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3686-4/15/10...
http://dx.doi.org/10.1145/2814251.2814265

ing: different parts of the input can now use different lexing schemes, and structure is possible at the lexical level (e.g. nested comments) [3]. However, it means that whitespace handling might now pollute the grammar as well as the generated syntax tree. Finally, while PEGs make linear-time parsing possible with full memoization[1], there is a fine balance to be struck between backtracking and memoization [2]. Memoization can bring about runtime speedups at the cost of memory use. Sometimes however, the run time overhead of memoization nullifies any gains it might bring.

Other problems plague parsing tools of all denominations. While solutions exist, they rarely coexist in a single tool. Error reporting tends to be poor, and is not able to exploit knowledge held by users about the structure of their grammars. Syntax trees often consist of either a full parse tree that closely follows the structure of the grammar, or data structures built on the fly by user-supplied code (*semantic actions*). Both approaches are flawed: a full parse tree is too noisy as it captures syntactic elements with no semantic meaning, while tangling grammatical constructs and semantic actions (i.e. code) produces bloated and hard-to-read grammars. Generating trees from declarative grammar annotations is possible, but infrequent.

Solution To tackle these issues, we introduce a new parsing library called Autumn. Autumn implements a generic PEG parser with selective memoization. It supports left-recursion (including indirect and hidden left-recursion) and both types of associativity. It also features a new construct called *expression cluster*. Expression clusters enable the aforementioned features to work faster in parts of grammars dedicated to postfix and infix expressions. Autumn also tackles, to some extent, the problems of whitespace handling, error reporting, and syntax tree creation. By alleviating real and significant pain points with PEG parsing, Autumn makes PEG parsing more practical.

Structure This paper starts by describing the issues that occur when using left-recursion to define the syntax of infix and postfix binary operators (section 2). Next we will describe our solutions to these issues (section 3). Then, we show the freedom afforded by the PEG paradigm regarding extensibility and lay down our understanding of how this paradigm may be extended further (section 4). Finally, we compare Autumn to other PEG and CFG parsers (section 5) and review related work (section 6) before concluding. Because of space restrictions, we do not review the basics of the PEG formalism, but refer to Ford's original paper [3].

2. Problems Caused by Binary Operators

This section explains why infix and postfix binary[2] operators are a significant pain point in terms of expressivity, performance, and syntax tree construction. Even more so with PEGs, due to their backtracking nature and poor handling of left-recursion. These issues motivate many of the features supported by our parser library. Our running example is a minimalistic arithmetic language with addition, subtraction, multiplication and division operating on single-digit numbers. Table 1 shows four PEGs that recognise this language, albeit with different associativity. They all respect the usual arithmetic operator precedence. Grammars (a), (b) and (c) are classical PEGs, whose specificities we now examine. Grammar (d) exhibits our own *expression clusters* and represents our solution to the problems presented in this section.

No support for left-recursion. The recursive-descent nature of PEGs means that most PEG parsers cannot support all forms of left-recursion, including indirect and hidden left recursion.[3] Left-recursion is direct when a rule designates a sequence of which the first element is a recursive reference, or when a rule designates a choice which has such a sequence as alternate. The reason this type of recursion is singled out is that it is easy to transform into a non-left-recursive form. Left-recursion is hidden when it might or might not occur depending on another parsing expression. For instance, $X = Y?\ X$ can result in hidden left-recursion because $Y?$ might succeed consuming no input.

PEG parsers that do not support left-recursion can only handle grammars (a) and (b). These parsers are unable to produce a left-associative parse of the input. Some tools can handle direct left-recursive rules by rewriting them to the idiomatic (b) form and re-ordering their parse tree to simulate a left-associative parse [4, 9].

We argue it is necessary to support indirect and hidden left-recursion, so that the grammar author is able to organise his grammar as he sees fit. Autumn supports left-recursion natively, as will be described in section 3. Using expression clusters, the associativity for operators that are both left- and right-recursive can be selected explicitly by using the @left_recur annotation (as shown in grammar (d)). Right-associativity is the default, so no annotations are required in that case.

Performance issues in non-memoizing parsers. Grammar (a) is parsed inefficiently by non-memoizing parsers. Consider a grammar for arithmetic with L levels of precedence and P operators. In our example, $L = P = 2$. This grammar will parse a single number in $O((P+1)^L)$ expression invocations (i.e. attempts to match an expression). For the Java language this adds up to thousands of invocations to parse a single number. The complexity is somewhat amortized for longer arithmetic expressions, but the cost remains prohibitively high. Memoizing all rules in the grammar makes the complexity $O(PL)$, but this coarse-grained solution might slow things down because of the inherent memoization overhead: cache lookups can be expensive [2].

[1] In this context, memoization means caching the result of the invocation of a parsing expression at a given position.

[2] *Binary* should be understood broadly here: n-ary infix operators (such as the ternary conditional operator) can be modelled in terms of binary operators.

[3] To the best of our knowledge, Autumn is the only available parser to support all forms of left-recursion with associativity selection.

$E = S\ \text{`+'}\ E \mid S\ \text{`}-\text{'}\ E \mid S$
$S = N\ \text{`}*\text{'}\ S \mid N\ \text{`/'}\ S \mid N$
$N = [0-9]$

(a) Layered, right-associative

$E = S\ (\ \text{`+'}\ E)* \mid S\ (\ \text{`}-\text{'}\ E)* \mid S$
$S = N\ (\ \text{`}*\text{'}\ S)* \mid N\ (\ \text{`/'}\ S)* \mid S$
$N = [0-9]$

(b) Idiomatic

$E = E\ \text{`+'}\ S \mid E\ \text{`}-\text{'}\ S \mid S$
$S = S\ \text{`}*\text{'}\ N \mid S\ \text{`/'}\ N \mid N$
$N = [0-9]$

(c) Layered, left-associative

$E = expr$
$\quad \rightarrow E\ \text{`+'}\ E \quad$ @+ @left_recur
$\quad \rightarrow E\ \text{`}-\text{'}\ E$
$\quad \rightarrow E\ \text{`}*\text{'}\ E \quad$ @+ @left_recur
$\quad \rightarrow E\ \text{`/'}\ E$
$\quad \rightarrow [0-9] \quad$ @+

(d) Autumn expression cluster

Table 1: 4 PEGs describing a minimal arithmetic language. E stands for Expression, S for Summand and N for Number.

In contrast, parsing a number in grammar (b) is always $O(PL)$. Nevertheless, grammar (a) still produces a meaningful parse if the operators are right-associative. Not so for grammar (b), which flattens the parse into a list of operands.

PEG parsers supporting left-recursion can use grammar (c), the layered, left-associative variant of grammar (a). Our own implementation of left-recursion requires breaking left-recursive cycles by marking at least one expression in the cycle as left-recursive. This can optionally be automated. If the rules are marked as left-recursive, using grammar (c) we will parse a single-digit number in $O(PL)$. If, however, the cycle breaker elects to mark the sequence expressions corresponding to each operator (e.g. $(E\ \text{`+'}\ S)$) as left-recursive, then the complexity is $O((P+1)^L)$.

Expression clusters (grammar (d)) do enable parsing in $O(PL)$ without user intervention or full memoization. This is most welcome, since the algorithm we use to handle left-recursion does preclude memoization while parsing a left-recursive expression.

Implicit precedence. Grammars (a), (b) and (c) encode precedence by grouping the rules by precedence level: operators in S have more precedence than those in E. We say such grammars are *layered*. We believe that these grammars are less legible than grammar (d), where precedence is explicit. In an expression cluster, precedence starts at 0, the @+ annotation increments the precedence for the alternate it follows, otherwise precedence remains the same. It is also easy to insert new operators in expression clusters: simply insert a new alternate. There is no need to modify any other parsing expression.[4]

3. Implementation

This section gives an overview of the implementation of Autumn, and briefly explains how precedence and left-recursion handling are implemented.

[4] We are talking about grammar evolution here, i.e. editing a grammar. Grammar composition is not yet supported by the library.

3.1 Overview

Autumn is an open source parsing library written in Java, available online at `http://github.com/norswap/autumn`.

The library's entry points take a PEG and some text to parse as input. A PEG can be thought of as a graph of parsing expressions. For instance a sequence is a node that has edges towards all parsing expressions in the sequence. The PEG can be automatically generated from a grammar file, or built programmatically, in the fashion of parser combinators.

Similarly, parsing can be seen as traversing the parsing expression graph. The order and number of times the children of each node are visited is defined by the node's parsing expression type. For instance, a sequence will traverse all its children in order, until one fails; a choice will traverse all its children in order, until one succeeds. This behaviour is defined by how the class associated to the parsing expression type implements the `parse` method of the `ParsingExpression` interface. As such, each type of parsing expression has its own mini parsing algorithm.

3.2 Precedence

Implementing precedence is relatively straightforward. First, we store the current precedence in a global parsing state, initialized to 0 so that all nodes can be traversed. Next, we introduce a new type of parsing expression that records the precedence of another expression. A parsing expression of this type has the expression to which the precedence must apply as its only child. Its role is to check if the precedence of the parsing expression is not lower than the current precedence, failing if it is the case, and, otherwise, to increase the current precedence to that of the expression.

Using explicit precedence in PEGs has a notable pitfall. It precludes memoization over *(expression, position)* pairs, because the results become contingent on the precedence level at the time of invocation. As a workaround, we can disable memoization for parts of the grammar (the default), or we can memoize over *(expression, position, precedence)* triplets using a custom memoization strategy.

3.3 Left-Recursion and Associativity

To implement left-recursion, we build upon Seaton's work on the Katahdin language [8]. He proposes a scheme to handle left-recursion that can accommodate both left- and right-associativity. In Katahdin, left-recursion is strongly tied to precedence, much like in our own expression clusters. This is not a necessity however, and we offer stand-alone forms of left-recursion and precedence in Autumn too.

Here also, the solution is to introduce a new type of parsing expression. This new parsing expression has a single child expression, indicating that this child expression should be treated as left-recursive. All recursive references must be made to the new left-recursive expression.

Algorithm 1 presents a simplified version of the parse method for left-recursive parsing expressions. The algorithm maintains two global data structures. First, a map from *(po-*

sition, expression) pairs to parse results. Second, a set of blocked parsing expressions, used to avoid right-recursion in left-associative parses. A parse result represents any data generated by invoking a parsing expression at an input position, including the syntax tree constructed and the amount of input consumed. We call the parse results held in our data structure *seeds* [1] because they represent temporary results that can "grow" in a bottom-up fashion. Note that our global data structures are "global" (in practice, scoped to the ongoing parse) so that they persist between (recursive) invocations of the algorithm. Other implementations of the `parse` method need not be concerned with them.

Let us first ignore left-associative expressions. When invoking a parsing expression at a given position, the algorithm starts by looking for a seed matching the pair, returning it if present. If not, it immediately adds a special seed that signals failure. We then parse the operand, update the seed, and repeat until the seed stops growing. The idea is simple: on the first go, all left-recursion is blocked by the failure seed, and the result is our base case. Each subsequent parse allows one additional left-recursion, until we have matched all the input that could be. For rules that are both left- and right-recursive, the first left-recursion will cause the right-recursion to kick in. Because of PEG's greedy nature, the right-recursion consumes the rest of the input that can be matched, leaving nothing for further left-recursions. The result is a right-associative parse.

Things are only slightly different in the left-associative case. Now the expression is blocked, so it cannot recurse, except in left position. Our loop still grows the seed, ensuring a left-associative parse.

The algorithm has a few pitfalls. First, it requires memoization to be disabled while the left-recursive expression is being parsed. Otherwise, we might memoize a temporary result. Second, for left-associative expressions, it blocks all non-left recursion while we only need to block right-recursion. To enable non-right recursion, our implementation includes an escape hatch operator that inhibits the blocked set while its operand is being parsed. This operator has to be inserted manually.

3.4 Expression Clusters

Expression clusters integrate left-recursion handling with precedence. As outlined in section 2, this results in a readable, easy-to-maintain and performant construct.

An expression cluster is a choice where each alternate must be annotated with a precedence (recall the @+ annotation from earlier), and can optionally be annotated with an associativity. Alternates can additionally be marked as left-associative, right-associativity being the default. All alternates at the same precedence level must share the same associativity, hence it needs to be mentioned only for the first alternate.

Like left-recursive and precedence expressions, expression clusters are a new kind of parsing expression. Algorithm 2 describes the `parse` method of expression clusters. The code presents a few key differences with respect to the regular left-recursion parsing algorithm. We now maintain a map from cluster expressions to their *current precedence*. We iterate over all the precedence groups in our cluster, in decreasing order of precedence. For each group, we verify that the group's precedence is not lower than the current precedence. If not, the current precedence is updated to that of the group. We then iterate over the operators in the group, trying to grow our seed. After growing the seed, we retry all operators in the group *from the beginning*. Note that we can do away with the blocked set: left-associativity is handled via the precedence check. For left-associative groups, we increment the precedence by one, forbidding recursive entry in the group. Upon finishing the invocation, we remove the current precedence mapping only if the invocation was not recursive: if it was, another invocation is still making use of the precedence.

```
1  seeds = {}
2  blocked = []
3  parse expr: left-recursive expression at position:
4      if seeds[position] [expr] exists then
5          return seeds[position][expr]
6      else if blocked contains expr then
7          return failure
8      current = failure
9      seeds[position][expr] = failure
10     if expr is left-associative then
11         blocked.add(expr)
12     repeat
13         result = parse(expr.operand)
14         if result consumed more input than current then
15             current = result
16             seeds[position][expr] = result
17         else
18             remove seeds[position][expr]
19             if expr is left-associative then
20                 blocked.remove(expr)
21             return current
```

Algorithm 1: Left-recursion and associativity handling.

4. Customizing Parser Behaviour

4.1 Adding New Parsing Expression Types

The core idea of Autumn is to represent a PEG as a graph of parsing expressions implementing a uniform interface. By implementing the `ParsingExpression` interface, users can create new types of parsing expressions. Many of the features we will introduce in this section make use of this capability.

Restrictions The only restriction on custom parsing expressions is *the single parse rule*: invoking an expression at a given position should always yield the same changes to the parse state. Custom expressions should follow this rule, and ensure that they do not cause other expressions to violate it. This limits the use of global state to influence the behaviour

```
1  seeds = {}
2  precedences = {}
3  parse expr: cluster expression at position:
4      if seeds[position] [expr] exists then
5          return seeds[position][expr]
6      current = failure
7      seeds[position][expr] = failure
8      min_precedence = precedences[expr] if defined, else 0
9      loop: for group in expr.groups do
10         if group.precedence < min_precedence then
11             break
12         precedences[expr] = group.precedence +
13             group.left_associative ? 1 : 0
14         for op in group.ops do
15             result = parse(op)
16             if result consumed more input than current then
17                 current = result
18                 seeds[position][expr] = result
19                 goto loop
20     remove seeds[position][expr]
21     if there is no other ongoing invocation of expr then
22         remove precedences[expr]
23     return current
```

Algorithm 2: Parsing with expression clusters.

of sub-expressions. Respecting the rule makes memoization possible and eases reasoning about the grammar.

The rule is not very restrictive, but it does preclude the user from changing the way other expressions parse. This is exactly what our left-recursion and cluster operators do, by blocking recursion. We get away with this by blocking memoization when using left-recursion or precedence. There is a workaround: use a transformation pass to make modified copies of sub-expressions. Experimenting with it was not one of our priorities, as experience shows that super-linear parse times are rare. In practice, the fact that binary operators are exponential in the number of operators (while still linear in the input size) is a much bigger concern, which is adequately addressed by expression clusters.

Extending The Parse State To be practical, custom parsing expressions may need to define new parsing states, or to annotate other parsing expressions. We enable this by endowing parsing expressions, parsers and parse states with an *extension object*: essentially a fast map that can hold arbitrary data. There are also a few hooks to the library's internals. Our design objective was to allow most native operators to be re-implemented as custom expressions. Since many of our features are implemented as parsing expressions, the result is quite flexible.

4.2 Grammar Instrumentation

Our library includes facilities to transform the expression graph before starting the parse. Transformations are specified by implementing a simple visitor pattern interface. This can be used in conjunction with new parsing expression types to instrument grammars. In particular, we successfully used custom parsing expression types to trace the execution of the parser and print out debugging information.

We are currently developing a grammar debugger for Autumn and the same principle is used to support breakpoints: parsing expressions of interest are wrapped in a special parsing expression that checks whether the parse should proceed or pause while the user inspects the parse state.

Transforming expression graphs is integral to how Autumn works: we use such transformations to resolve recursive reference and break left-recursive cycles in grammars built from grammar files.

4.3 Customizable Error Handling & Memoization

Whenever an expression fails, Autumn reports this fact to the configured error handler for the parse. The default error reporting strategy is to track and report the farthest error position, along with some contextual information.

Memoization is implemented as a custom parsing expression taking an expression to memoize as operand. Whenever the memoization expression is encountered, the current parse state is passed to the memoization strategy. The default strategy is to memoize over *(expression, position)* pairs. Custom strategies allow using memoization as a bounded cache, discriminating between expressions, or including additional parse state in the key.

4.4 Syntax Tree Construction

In Autumn, syntax trees do not mirror the structure of the grammar. Instead, an expression can be *captured*, meaning that a node with a user-supplied name will be added in the syntax tree whenever the expression succeeds. Nodes created while parsing the expression (via captures on sub-expressions) will become children of the new node. This effectively elides the syntax tree and even allows for some nifty tricks, such as flattening sub-trees or unifying multiple constructs with different syntax. The text matched by an expression can optionally be recorded. Captures are also implemented as a custom parsing expression type.

4.5 Whitespace Handling

The parser can be configured with a parsing expression to be used as whitespace. This whitespace specification is tied to *token parsing expressions*, whose foremost effect is to skip the whitespace that follows the text matched by their operand. A token also gives semantic meaning: it represents an indivisible syntactic unit. The error reporting strategy can use this information to good effect, for instance.

We mentioned earlier that we can record the text matched by an expression. If this expression references tokens, the text may contain undesirable trailing whitespace. To avoid this, we make Autumn keep track of the furthest non-whitespace position before the current position.

5. Evaluation

In Table 2, we measure the performance of parsing the source code of the Spring framework ($\sim$ 34 MB of Java code) and producing matching parse trees. The measure-

Parser	Time (Single)	Time (Iterated)	Memory
Autumn	13.17 s	12.66 s	6 154 KB
Mouse	101.43 s	99.93 s	45 952 KB
Parboiled	12.02 s	11.45 s	13 921 KB
Rats!	5.95 s	2.41 s	10 632 KB
ANTLR v4 (Java 7)	4.63 s	2.31 s	44 432 KB

Table 2: Performance comparison of Autumn to other PEG parsing tools as well as ANTLR. Measurements done over 34MB of Java code.

ments were taken on a 2013 MacBook Pro with a 2.3GHz Intel Core i7 processor, 4GB of RAM allocated to the Java heap (Java 8, client VM), and an SSD drive. The *Time (Single)* column reports the median of 10 task runs in separate VMs. The *Time (Iterated)* column reports the median of 10 task runs inside a single VM, after discarding 10 warm-up runs. The reported times do not include the VM boot time, nor the time required to assemble the parser combinators (when applicable). For all reported times, the average is always within 0.5s of the median. All files are read directly from disk. The *Memory* column reports the peak memory footprint, defined as the maximum heap size measured after a GC activation. The validity of the parse trees was verified by hand over a sampling of all Java syntactical features.

The evaluated tools are *Autumn*; *Rats!* [4], a state of the art packrat PEG parser generator with many optimizations; *Parboiled*, a popular Java/Scala PEG parser combinator library; *Mouse* [6], a minimalistic PEG parser generator that does not allow memoization; and, for comparison, *ANTLR v4* [9] a popular and efficient state of the art CFG parser.

Results show that Autumn's performance is well within the order of magnitude of the fastest parsing tools. This is encouraging, given that we did not dedicate much effort to optimization yet. Many optimizations could be applied, including some of those used in Rats! [4]. Each parser was evaluated with a Java grammar supplied as part of its source distribution. For Autumn, we generated the Java grammar by automatically converting the one that was written for Mouse. We then extracted the expression syntax into a big expression cluster and added capture annotations. The new expression cluster made the grammar more readable and is responsible for a factor 3 speedup of the parse with Autumn (as compared to Autumn without expression clusters).

6. Related Work

Feature-wise, some works have paved the way for full left-recursion and precedence handling. *OMeta* [12] is a tool for pattern matching over arbitrary data types. It was the first tool to implement left-recursion for PEGs, albeit allowing only right-associative parses. *Katahdin* [8] is a language whose syntax and semantics are mutable at run-time. It pioneers some of the techniques we successfully deployed, but is not a parsing tool per se. *IronMeta* is a port of OMeta to C# that supports left-recursion using an algorithm developed by Medeiros et al. [7]. This algorithm enables left-recursion, associativity and precedence by compiling parsing expressions to byte code for a custom virtual machine. However, Iron Meta doesn't support associativity handling.

7. Conclusion

Left-recursion, precedence and associativity are poorly supported by PEG parsers. Infix and postfix expressions also cause performance issues in left-recursion-capable PEG parsers. To solve these issues, we introduce Autumn, a parsing library that handles left-recursion, associativity and precedence in PEGs, and makes it efficient through a construct called *expression cluster*. Autumn's performance is on par with that of both state of the art and widely used PEG parsers. Autumn is built with extensibility in mind, and makes it easy to add custom parsing expressions, memoization strategies and error handlers. It offers lightweight solutions to ease syntax tree construction, whitespace handling and grammar instrumentation. In conclusion, Autumn is a practical parsing tool that alleviates significant pain points felt in current PEG parsers and constitutes a concrete step towards making PEG parsing practical.

Acknowledgments

We thank Olivier Bonaventure, Chris Seaton, the SLE reviewers and our shepherd Markus Völter for their advice.

References

[1] A. Warth et al. Packrat Parsers Can Support Left Recursion. In *PEPM*, pages 103–110. ACM, 2008.

[2] R. Becket and Z. Somogyi. DCGs + Memoing = Packrat Parsing but Is It Worth It? In *PADL*, LNCS 4902, pages 182–196. Springer, 2008.

[3] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *POPL*, pages 111–122. ACM, 2004.

[4] R. Grimm. Better Extensibility Through Modular Syntax. In *PLDI*, pages 38–51. ACM, 2006.

[5] G. Hutton. Higher-order functions for parsing. *J. Funct. Program. 2*, pages 323–343, 1992.

[6] R. R. Redziejowski. Mouse: From Parsing Expressions to a practical parser. In *CS&P 2*, pages 514–525. Warsaw University, 2009.

[7] S. Medeiros et al. Left Recursion in Parsing Expression Grammars. *SCP 96*, pages 177–190, 2014.

[8] C. Seaton. A Programming Language Where the Syntax and Semantics Are Mutable at Runtime. Master's thesis, University of Bristol, 2007.

[9] T. Parr et al. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *OOPSLA*, pages 579–598. ACM, 2014.

[10] L. Tratt. Direct left-recursive parsing expression grammars. Technical Report EIS-10-01, Middlesex University, 2010.

[11] L. Tratt. Parsing: The solved problem that isn't, 2011. URL `http://tratt.net/laurie/blog/entries/parsing_-the_solved_problem_that_isnt`.

[12] A. Warth et al. OMeta: An Object-oriented Language for Pattern Matching. In *DLS*, pages 11–19. ACM, 2007.