# For and Against PEGs: Why we need Multi-Ordered Grammars

Nick Papoulias
*University of La Rochelle, France*
*npapoylias@gmail.com*

***Abstract*—Since their introduction in 2004, Parsing Expression Grammars (PEGs for short) have been gaining widespread adoption both in industry and academia. More than 400 subsequent works [1] cite B. Ford's original paper, while a total of 29 implementations in 14 different programming languages are reported in active use [2]. Nevertheless reviewing PEG-related bibliography reveals that the original argumentation in favor of PEGs has actually been weakened by subsequent work, regarding basic parsing features such as (a) recursion handling and (b) associativity support. To this day all proposed enhancements either address these issues in isolation or in implementation specific ways. It is still unclear if there is a single way to parse PEGs without facing these issues or introducing implementation directives external to the formalism. Our subsequent analysis takes us a step further, questioning the very core of the initial PEG proposal: (c) by design unambiguous grammars. We then discuss why a form of ordered choice and conditional operators that PEGs advocate are worth saving, but only within a wider synthesis that could address the aforementioned issues. To this end we present our on-going effort with the Gray algorithm and MOGs (Multi-Ordered Grammars), a possible alternative to the PEG and CFG formalisms.**

***Index Terms*—Parsing, PEGs, Multi-Ordered Grammars, LALR, Earley, Gray**

## I. Introduction: PEGs' Past

Parsing is everywhere. From rudimentary data storage and retrieval, to protocol and communication structures and from there to file-formats, domain-specific dialects and general purpose language specifications. It's ubiquitous application can partially explain why it is still such an active area of research, or as L. Tratt describes it: "The Solved Problem That Isn't". [3] With so many different areas of application, come inherent trade-offs in terms of expression power, speed, comprehensibility or memory consumption of different approaches. Yet on 2004, B. Ford in one of the most influential papers in the domain [1] made the following startling claim:

*"For decades we have been using Chomskys generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs) [...] The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages,* ***but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs****."*

In essense Ford argues that most of our problems with parsing have been due to our bias towards linguistic solutions [2], [3] that are suitable for NLP (Natural Language Processing). We thus have been disregarding the needs of "easier" domains such as data or programming language specifications, where expressing ambiguity is unessary, leading to cumbersome solutions and implementations.

He then goes on to propose the PEG formalism as an alternative, which he shows to be reducible to earlier well understood systems such as TS/TDPL and gTS/GTDPL [4], [5]. PEGs by construction cannot express nondeterministic (*i.e.,* unordered) choice, thus avoiding ambiguities. They instead use prioritized (ordered) choice when presented with parsing alternatives, making the choice deterministic and efficient. Only if a chosen alternative fails directly (not through subsequent backtracking), will PEG-parsers try its alternatives. This is much like how a human developer will manually hard-code alternatives in a top-down parser. Since determining a correct order often involves look-aheads, PEGs also introduce the ! (not) and & (and) operators, which recognize (*i.e.,* determine if rule A is/not followed by rule B) but do not consume their input.

Figures 1 to 3 show us how a parsing algorithm using the vanilla PEG formalism described by Ford, compares to two of the most prominent CFG-based algorithms (LALR [6], [7] and Earley [8], [9]) when describing a simple expression grammar. For being precise in our comparison we use the same BNF symbols ($::=$ and $|$) for rule definition and choice in all examples (instead of the PEG-only variants $<-$ and $/$) assuming the appropriate semantics (ordered choice for PEGs and unordered choice for CFGs) in each case. Note that $number$ is a rule with a right-hand side terminal representing integers and that $expression, power, product, sum$ are non-terminal rules for arithmetic operations (caret ^ denotes exponentiation). Finally all other non-bracketed sequence of characters represent terminal character sequences and groups of ordinary regular expressions.

Starting with Section 2, we will discuss the different flavors of the expression grammar shown in Figures 1 to 3 in detail. We will argue that although PEG-related bibliography tried to enhance Ford's initial proposal, it has actually provided insights that weaken the argumentation in favor of PEGs.

In Section 3, we will explain why this apparent stagnation

[1] 439 citations according to Google Scholar as of 30/09/2018: https://scholar.google.com/scholar?q=Parsing+expression+grammars%3A+a+recognition-based+syntactic+foundation

[2] implementations reported in http://bford.info/packrat/

[3] Parsing: The Solved Problem That Isn't. (https://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt.html)

Fig. 1: The Earley CFG for expressions

⟨*expression*⟩ ::= ⟨*expression*⟩ `[+-]` ⟨*expression*⟩
| ⟨*expression*⟩ `[*/]` ⟨*expression*⟩
| ⟨*expression*⟩ `^` ⟨*expression*⟩
| ⟨*number*⟩

⟨*number*⟩ ::= [0-9]+

Fig. 2: The LALR CFG for expressions

```
%right '^'
%left '[+-]'
%left '[*/]'
```
⟨*expression*⟩ ::= ⟨*expression*⟩ `[+-]` ⟨*expression*⟩
| ⟨*expression*⟩ `[*/]` ⟨*expression*⟩
| ⟨*expression*⟩ `^` ⟨*expression*⟩
| ⟨*number*⟩

⟨*number*⟩ ::= [0-9]+

of the PEG project hints to a need for expressing ambiguity during design (even in cases considered "simpler" than NLP) while providing support for *incremental dis-ambiguation* within a new formalism. To this end we will present our on-going effort with the Gray algorithm and MOGs (Multi-Ordered Grammars), a possible alternative to the PEG and CFG formalisms. Finally, Section 4 concludes the paper and discusses future perspectives.

## II. Problem Statement: PEGs' Present

The expression grammar frequently appears in literature, not only because it is one of the simplest "realistic" examples. It is also an example where the need for handling left/right recursion, precedence and associativity, co-occur. The initial PEG paper does not provide such examples, but as we will promptly see these have been the focus of subsequent PEG-related papers.

Regarding the different grammar flavors (understood by Earley, LALR and PEG-parser respectively) in Figures 1 to 3, we observe the following:

(a) The shortest (and in our view the most natural way) to express the grammar is by using the Earley algorithm. This is despite the fact that the algorithm was explicitly designed for NLP. This conclusion is in contrast to what Ford argues, since the expression grammar falls under the "simpler than NLP" problems described in his initial paper. Nevertheless the result provided by Earley is indeed highly problematic, since it consists of all possible parsing trees (*e.g.,* for an expression as simple as: $2*3+4\wedge5\wedge6$ Earley will answer all 14 possible trees). Only manually re-writing the grammar (that will end-up resembling a lot like the PEG version) can produce an unambiguous Earley parse.

(b) The LALR algorithm is closer to Earley but in order to avoid ambiguity we need to provide implementation specific hints to handle shift/reduce and reduce/reduce conflicts. These are the three percentage (%) directives handling operator precedence (lower directives have higher precedence) and associativity (operators are explicitly stated as left or right associative). The Generalized LR algorithm can return the ambiguous forest as Earley does, with a few caveats (see 1.5: "Writing GLR Parsers" in [10]). Nevertheless neither LALR or GLR algorithms in state-of-the-art implementations (as in GNU/Bison) can handle all precedence or reduce conflicts (See Sections 5.7: "Mysterious Conflicts" and 5.3.6: "Using Precedence For Non Operators" in [10]) without grammar rewriting (as in the case of Adaptive LL(*) parsing[11]).

(c) The PEG version is the most verbose of the three, since it cannot directly handle left recursion or associativity and thus needs to distinguish between products, powers and sums. No support for left-recursion also means that the parsing output will be wrongly right-associative by default. Precedence is only partially defined using ordered choice, since we need to hard-code explicit right-recursive relations between sums, products and numbers. Nevertheless the parsing is indeed unambiguous without resorting to implementation specific directives.

Fig. 3: The PEG version for parsing expressions

⟨*expression*⟩ ::= ⟨*sum*⟩

⟨*sum*⟩ ::= ⟨*product*⟩ `[+-]` ⟨*sum*⟩
| ⟨*product*⟩

⟨*product*⟩ ::= ⟨*pow*⟩ `[*/]` ⟨*product*⟩
| ⟨*pow*⟩

⟨*pow*⟩ ::= ⟨*number*⟩ `^` ⟨*pow*⟩
| ⟨*number*⟩

⟨*number*⟩ ::= [0-9]+

The expression grammar shows us that Ford's initial argument against CFG is at least partially false (ie CFGs do express more naturally and correctly grammars outside NLP).

Subsequent refinements to PEGs from literature tried to remedy these problems, adding support for left-recursion [12] as seen in Figure 4 and associativity [13], as seen in Figure 5. It is worth noting here that these solutions address the problems either in isolation (A. Warth et al [12] where concerned only with left-recursion) or in implementation specific ways (N. Laurent and K. Mens introduce implementation specific directives to guide PEG parsing for their specific system called "Autumn" [13]). Both solutions report additional performance penalties for supporting these extentions [12], [13].

To this day it is still unclear if PEGs can successfully resolve these issues without introducing implementation directives

external to the formalism. To make things worse, a direct comparison of state-of-the-art PEG extensions (Figure 5) and the classic LALR solution with directives (Figure 2) differ only in their taste of implementation specific directives. While LALR needs the directives to avoid conflicts and ambiguity, PEG parsers need them to actually produce the correct parsing tree in a readable manner. This is why we argue that subsequent contributions to the PEG-bibliography have further weakened the initial PEG vs CFG argumentation, by ending up mimicking CFGs.

Fig. 4: Left-recursion extention for PEGs

⟨*expression*⟩ ::= ⟨*sum*⟩

⟨*sum*⟩ ::= ⟨*sum*⟩ `[+-]` ⟨*product*⟩
| ⟨*product*⟩

⟨*product*⟩ ::= ⟨*product*⟩ `[*/]` ⟨*pow*⟩
| ⟨*pow*⟩

⟨*pow*⟩ ::= ⟨*number*⟩ ^ ⟨*pow*⟩
| ⟨*number*⟩

⟨*number*⟩ ::= [0-9]+

Is there any reason then to use PEGs instead of CFGs ?

- **Possible Efficiency:** In examples where it's possible to hard-code precedence and associativity without using left-recursion or extra directives in the grammar, we can benefit from a linear-parsing time. Then again, such a grammar is likely to be well-behaved under CFG algorithms as well.
- **Guaranteed Output:** Given that neither Earley nor LALR can provide unambiguous grammars without additional effort, we might choose to use PEG parsers that **are guaranteed to at least provide some kind of output** (even if this output is initially wrong). The trade-off here is with arcane shift/reduce, reduce/reduce errors, or with getting back the whole parsing forest (as in the case of Earley). This of course means that the argument in favor of PEGs being "unambiguous" (although not technically wrong) is misleading. PEGs are guaranteed to provide a single (possibly wrong) output, with which we need to experiment and possibly provide further directives (external to PEGs) to circumvent restrictions imposed by the formalism.

Our goal of course here is not to refute Ford (more than a decade later hindsight is 20/20). But to open a concrete discussion for ways to move forward, in what we perceive as stagnation. Moreover we believe that the ordered choice and conditional operators that PEGs advocate are worth saving, but only within a wider synthesis that could address the aforementioned issues.

Fig. 5: Associativity solution for PEGs

⟨*expression*⟩ ::= ⟨*expression*⟩ `[+-]` ⟨*expression*⟩
`@+ @left_recur`
| ⟨*expression*⟩ `[*/]` ⟨*expression*⟩
`@+ @left_recur`
| ⟨*expression*⟩ ^ ⟨*expression*⟩
| ⟨*number*⟩ `@+`

⟨*number*⟩ ::= [0-9]+

## III. Beyond PEGs: MOGs and The Gray Algorithm

Since the PEG program seems to be now mimicking CFGs we might conclude that the last 15 years of research have come full-circle. Nevertheless as we saw in Section 3 PEGs did provide us with a means of "dis-ambiguation" (the ordered choice) that despite its multiple problems, can be used to explore the "domain of possible parse trees" without resorting to cryptic errors and conflicts.

This dimension of **exploration** through dis-ambiguation is the starting point of our own efforts. Unlike Ford, we begin by embracing the ambiguity of CFGs and the non-deterministic nature of algorithms inspired by NLP. But, given the insights that we gained from the PEG program, we are experimenting with **incremental dis-ambiguation** through ordered-choice operators acting within (not instead of) an ambiguous grammar. This led us to the following research questions:

*Are multi-ordered grammars possible ? Can a mix of ordered (deterministic) and unordered (non-deterministic) rules produce consistent grammar semantics ?*

To answer these questions we are developing in parallel a possible MOG (Multi-Ordered Grammar) formalism and an accompanying parsing algorithm (the Gray algorithm, short for "Grammar Ray") that is able to analyze MOGs. Our short exposition here of both MOGs and Gray here, aims only to show that this is an alternative worth exploring. Depending on your point of origin (CFGs or PEGs), MOGs can be loosely described as, either:

$$MOG = PEG + Unordered_c + RecOrdered_c \quad (1)$$

That is a MOG is a PEG augmented by unordered and recursively ordered choice operators, or as:

$$MOG = CFG + LAhead_o + Ordered_c + RecOrdered_c \quad (2)$$

That is a MOG is a CFG augmented by the two lookahead operators $LAhead_o$ (& and !), plus the ordered and recursively ordered choices. Besides this experimental mixing of ordered and unordered choices, on other thing unique to MOGs are the $RecOrdered_c$ (*Recursively Ordered Choice* operators). Their meaning (as well as that of all other MOG operators) is described in Table I.

Figures 6 and 7 shows what we can currently achieve with MOGs, in terms of exploring ambiguity and incrementally dis-

| Operators | Appears In | Description |
|---|---|---|
| *, +, () | **MOG**,EBNF,PEG,RE | *Semantics from reg-expression rules, for zero/one-or-more and grouping.* |
| \| | **MOG**,EBNF,BNF | ***Unordered choice**. In MOGs unordered choice can be tainted (see below)* |
| \|\| | **MOG(scoped)**,PEG | ***Ord. choice**. Appears as / in PEGs. In MOGs ordered choice can be exhaustive, and introduces a new scope for recursive ordering that is MOG-specific. All ordered rules, taint* (i.e., *order their alternatives)* |
| / , \ | **MOG-only** | ***Recursive Ord. choice (MOGs only),** comes in two flavors: self-recursive (/) and simply-recursive (\). Has identical semantics to (\|\|) unless the rule is recursively invoked, in which case parsing continues from prevously seen (self-recursive) or next (simply-recursive) alternatives.* |
| & , ! | **MOG**, PEG | ***Recognize but do not consume**. Determine if A is/not followed by B* |

TABLE I: Operators common in MOGs and other formalisms

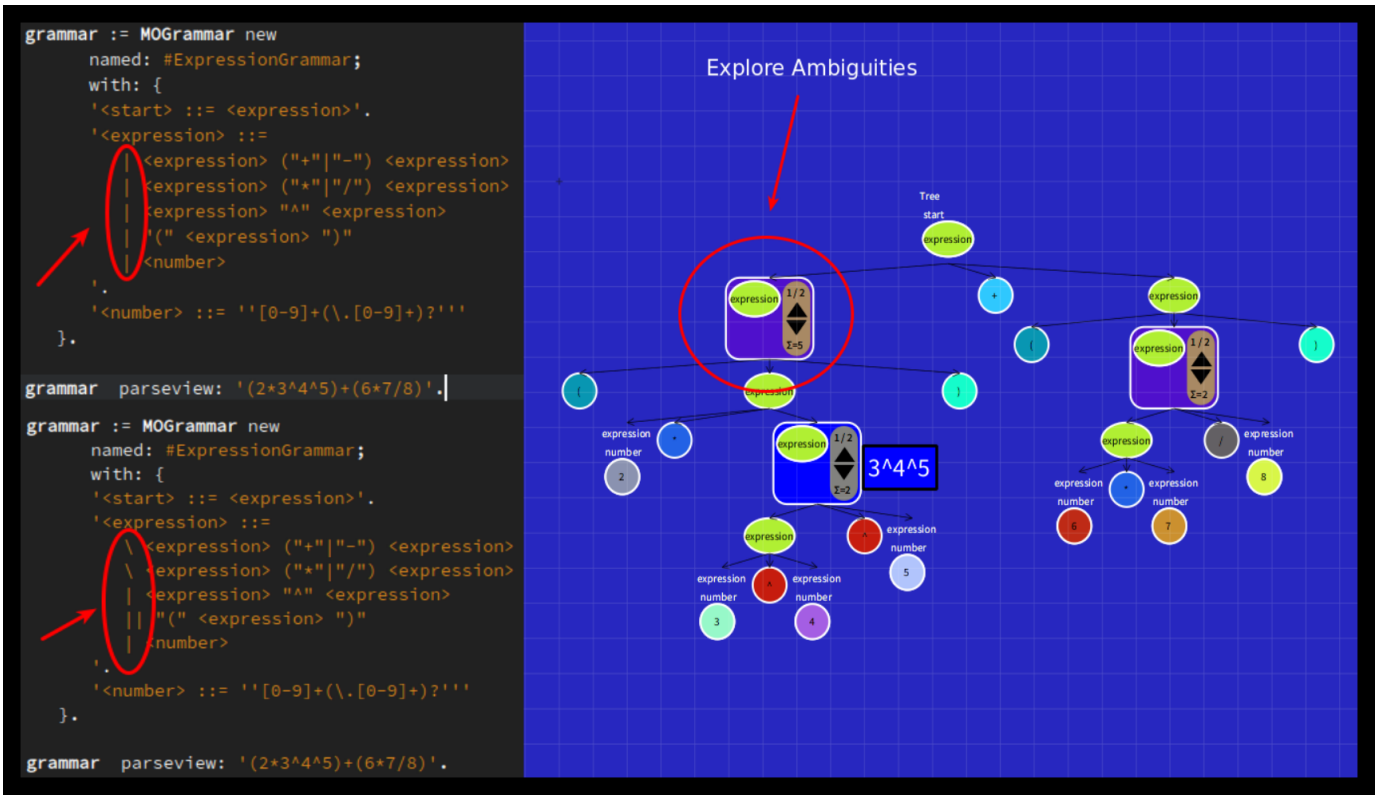


Fig. 6: Exploration: The MOG Expr. Grammar and Tooling

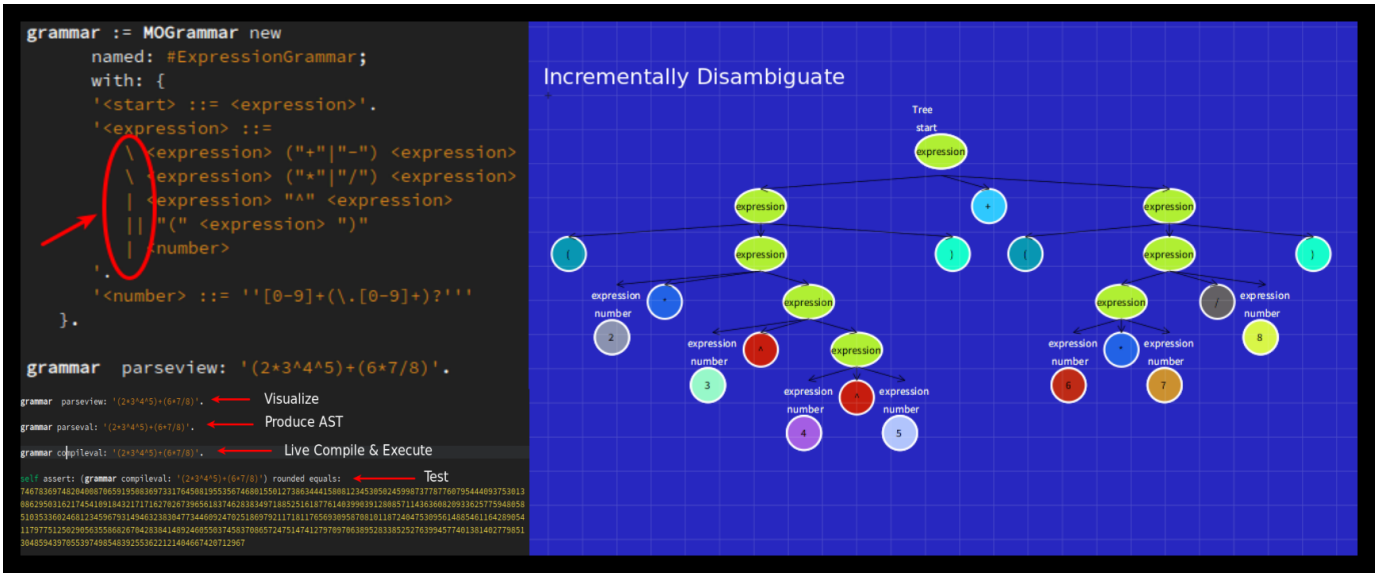


Fig. 7: Disambiguation: The MOG Expr. Grammar and Tooling

ambiguating naturally expressed grammars (screen-shots from our prototype with Gray).

The successful mixing of the choice alternatives in Gray (source code available at: https://npapoylias.gitlab.io/lands-project/) is achieved via an extended set of chart parsing operators [14] (traditionally used in NLP). We start with a parsing base similar to that of a 3-op Earley parser (scan, predict, complete) [8] and first extend with standard EBNF operators (+,*,()) and the empty rule. On top of this charter base we implement two additional operations (backtrack and fork) to handle the full-backtracking ordered choice (||) and the two lookahead operators (& , !). Then finally we override the standard predict operators to accommodate for recursive ordering (/ , \) and the mixing of order with unordered choices. To optimize scanning and memoization, we pre-compute all first,follow and predict sets to pre-filter unwanted alternatives.

## IV. Conclusion & Future Perspectives

By reviewing PEG-related bibliography we have argued that the original motivation for PEGs (*i.e.,* provide a CFG alternative) has actually been weakened by subsequent work. In fact we showed that today's state-of-the-art PEG solutions are mimicking LALR parsing by introducing implementation specific directives to circumvent PEG restrictions. We then showed why ordered choice and conditional operators (that are PEG-specific), do worth our attention but only within a wider synthesis that could successfully address precedence, recursion and associativity issues. A possible route forward with the Gray algorithm and MOGs (Multi-Ordered Grammars) was discussed, as alternative to PEG and CFG formalisms. From this perspective we are preparing for a formalized presentation of MOGs and Gray, in tandem with a complexity analysis of the algorithm.

## References

[1] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *ACM SIGPLAN Notices*, vol. 39, no. 1. ACM, 2004, pp. 111–122.

[2] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[3] ——, "On certain formal properties of grammars," *Information and control*, vol. 2, no. 2, pp. 137–167, 1959.

[4] A. Birman, "The tmg recognition in schema." Ph.D. dissertation, Princeton., 1970.

[5] A. V. Aho and J. D. Ullman, "The theory of parsing, translating, and compiling, vol. ii," 1972.

[6] F. L. DeRemer, "Practical translators for lr (k) languages." Ph.D. dissertation, Massachusetts Institute of Technology, 1969.

[7] F. DeRemer and T. Pennello, "Efficient computation of lalr (1) look-ahead sets," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 4, pp. 615–649, 1982.

[8] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[9] ——, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 26, no. 1, pp. 57–61, 1983.

[10] C. Donnely and R. Stallman, "Gnu bison–the yacc-compatible parser generator," 2015.

[11] T. Parr, S. Harwell, and K. Fisher, "Adaptive ll (*) parsing: the power of dynamic analysis," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 579–598.

[12] A. Warth, J. R. Douglass, and T. D. Millstein, "Packrat parsers can support left recursion." *PEPM*, vol. 8, pp. 103–110, 2008.

[13] N. Laurent and K. Mens, "Parsing expression grammars made practical," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2015, pp. 167–172.

[14] D. Jurafsky, *Speech & language processing*. Pearson Education India, 2000.