Parsing Multi-Ordered Grammars with the Gray Algorithm

Nick Papoulias[1]

[1]**University of La Rochelle, CNRS**

Corresponding author:
Nick Papoulias[1]

Email address: npapoylias@gmail.com

ABSTRACT

Background. Context-free grammars (CFGs) and Parsing-expression Grammars (PEGs) are the two main formalisms used by formal specifications and parsing frameworks to describe programming languages. They mainly differ in the definition of the choice operator, describing language alternatives. CFGs support the use of non-deterministic choice (*i.e.,* unordered choice), where all alternatives are equally explored. PEGs support a deterministic choice (*i.e.,* ordered choice), where alternatives are explored in strict succession. In practice the two formalisms, are used through concrete classes of parsing algorithms (such as Left-to-right, rightmost derivation (LR) for CFGs and Packrat parsing for PEGs), that follow the semantics of the formal operators.
Problem Statement. Neither the two formalisms, nor the accompanying algorithms are sufficient for a complete description of common cases arising in language design. In order to properly handle ambiguity, recursion, precedence or associativity, parsing frameworks either introduce implementation specific directives or ask users to refactor their grammars to fit the needs of the framework/algorithm/formalism combo. This introduces significant complexity even in simple cases and results in incompatible grammar specifications.
Our Proposal. We introduce Multi-Ordered Grammars (MOGs) as an alternative to the CFG and PEG formalisms. MOGs aim for a better exploration of ambiguity, ordering, recursion and associativity during language design. This is achieved by (a) allowing both deterministic and non-deterministic choices to co-exist, and (b) introducing a form of recursive and scoped ordering. The formalism is accompanied by a new parsing algorithm (Gray) that extends chart parsing (normally used for Natural Language Processing) with the proposed MOG operators.
Results. We conduct two case-studies to assess the expressiveness of MOGs, compared to CFGs and PEGs. The first consists of two idealized examples from literature (an expression grammar and a simple procedural language). The second examines a real-world case (the entire Smalltalk grammar and eleven new Smalltalk extensions) probing the complexities of practical needs. We show that in comparison, MOGs are able to reduce complexity and naturally express language constructs, without resorting to implementation specific directives.
Conclusion. We conclude that combining deterministic and non-deterministic choices in a single grammar specification is indeed not only possible but also beneficial. Moreover, augmented by operators for recursive and scoped ordering the resulting multi-ordered formalism presents a viable alternative to both CFGs and PEGs. Concrete implementations of MOGs can be constructed by extending chart parsing with MOG operators for recursive and scoped ordering.

1 BACKGROUND

Parsing is ubiquitous, from rudimentary data storage and retrieval, to protocol and communication structures and from there to file-formats, domain-specific, general purpose and natural language specifications. This wide range of applications 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"* [38]. With so many different areas of application, come inherent trade-offs in terms of expression power, speed, comprehensibility or memory consumption of different approaches.

Since their introduction in 1956 by N. Chomsky [12], context-free grammars have been the de-facto formalism for describing languages. In fact as J. Kegler [28] notes, the very definitions of "parser",

”recognizer” and ”language” as we use them today can be traced back to this first paper on the subject by N. Chomsky. CFGs will be later popularized in the context of Computer Science with the introduction of Backus’s notation [4], later enhanced by Naur for Algol 60 [5] (resulting in the acronym BNF, for Backus-Naur Form, and most commonly used in its extended form – EBNF [25]).

CFGs present a number of challenges, both for the algorithms that consume them (for parsing or recognition) and for the end-users defining them (handling of ambiguity, recursion, precedence and associativity). This is why in practice parsing frameworks based on CFGs introduce additional implementation specific directives, or ask users to restrict their grammars in ways that fit the framework’s design (*e.g.,* eliminating left-recursion). The most obvious, perceived drawback of CFGs for Informatics is that of ambiguity. Since CFGs allow for non-deterministic choices between syntactic alternatives, they can lead to ambiguous parses of the same input.

It is exactly this drawback that B. Ford focused on, in one of the most influential papers since the introduction of CFGs [24] making the following startling claim:

”For decades we have been using Chomsky’s 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 essence Ford argues that most of our problems with parsing have been due to our bias towards linguistic solutions [12, 13] that are suitable for Natural Languages [14]. We thus have been disregarding the needs of ”easier” domains such as data or programming language specifications, where expressing ambiguity is unnecessary, 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 [8, 1]. PEGs by construction cannot express non-deterministic (*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 but do not consume their input (*i.e.,* determine if rule A is/not followed by rule B).

Since their introduction in 2004, Parsing Expression Grammars 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 (in the same way that CFG-based solutions do). Our analysis in this paper takes us a step further, questioning the very core of the initial PEG proposal: (c) by design unambiguous grammars. We explain 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.

Starting with **Section 2**, we will use a simple expression grammar to discuss the differences between CFGs, PEGs and their proposed enhancements (as seen in Figures 1 to 5). 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. We will explain why this direction of the PEG project hints to a need for expressing ambiguity during language design, while providing support for *incremental disambiguation*. To this end in **Section 3** we will introduce MOGs (Multi-ordered grammars) and the Gray algorithm in detail, as alternatives to the PEG and CFG formalisms. Then in **Section 4** we will present the results of our case-studies assessing the expressiveness of MOGs, compared to CFGs and PEGs. Two

[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/`

idealized examples (an expression grammar and a simple procedural language) and two real-world cases (the entire Smalltalk grammar and eleven new Smalltalk extensions) probing the complexities of practical needs, will be discussed. Finally, **Section 5** will conclude the paper and discuss future perspectives.

2 PROBLEM & RELATED WORK

Figures 1 to 3 show us how two of the most prominent CFG-based algorithms (LALR [16, 15] and Earley [20, 21]) compare to a parsing algorithm using the vanilla PEG formalism described by Ford, 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 here that the rule *number* has a right-hand side terminal representing integers and that *expression*, *power*, *product*, *sum* are non-terminal rules for arithmetic operations (caret ˆ denotes exponentiation). Finally all other non-bracketed sequence of characters represent terminal character sequences and groups of ordinary regular expressions.

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, ambiguity, 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:

Figure 1: The Earley CFG for expressions

```
<expression> ::= <expression> "[+-]" <expression>
               | <expression> "[*/]" <expression>
               | <expression> "^" <expression>
               | <number>

<number> ::= [0-9]+
```

Figure 2: The LALR CFG for expressions

```
%right '^'
%left '[+-]'
%left '[*/]'

<expression> ::= <expression> "[+-]" <expression>
               | <expression> "[*/]" <expression>
               | <expression> "^" <expression>
               | <number>

<number> ::= [0-9]+
```

The **Earley algorithm** (Figure 1) provides the shortest and most natural way to express the grammar. 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”, computational 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: $2*3+4\wedge 5\wedge 6$ Earley will answer all 14 possible trees). Only manually re-writing the grammar (that will end-up resembling a lot like the PEG version, studied below) can produce an unambiguous Earley parse. Later enhancements to the algorithm, have focused on empty-rule handling [3] parallelism [11], complexity [30, 27] and performance [2, 31, 32]. The ambiguous output of Earley is widely considered a feature (especially for Natural Language Processing) rather than a problem, with the exception of precedence handling through external directives [27].

The **LALR algorithm** (Figure 2) is close to the Earley version but in order to avoid ambiguity we again need to provide implementation specific hints to handle shift/reduce and reduce/reduce conflicts. These are the three percentage (%) directives (on lines 1 to 3) 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 [17]). Nevertheless neither LALR or GLR algorithms in state-of-the-art implementations (as in GNU/Bison or SmaCC [10]) can handle all precedence or reduce conflicts (See Sections 5.7: ”Mysterious Conflicts” and 5.3.6: ”Using Precedence For Non Operators” in [17]) without grammar rewriting (as in the case of Adaptive LL(*) parsing[35]).

The **PEG version** (Figure 3, predominately parsed by Packrat algorithms [22, 23]) 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.

Figure 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 CFGs is at least partially false. CFGs do express more naturally and correctly grammars outside NLP, but need to resort to implementation specific directives to handle precedence and ambiguity. Ford’s formulation of PEGs on the other hand, is unambiguous by design but forces the user to adopt a very specific way to describe grammars, with no left-recursion, problematic associativity and hand-coded precedence. Subsequent refinements to PEGs from literature tried to remedy these problems, adding support for left-recursion (in OMeta [39, 41] and Ohm [18, 40]) as seen in Figure 4 and associativity [29], as seen in Figure 5. It is worth noting here that these solutions address the problems either in isolation (A. Warth et al [39] 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 ”Autumn” framework [29]). Both solutions report additional performance penalties for supporting these extensions [39, 29]. Relying on the specificities of the framework rather than the operators of the formalism to circuvment the drawbracks of PEGs is not restricted to the examples above. Other widely used PEG frameworks like PetitParser [36, 33] also adopt this strategy, albeit in a less general way. PetitParser introduces additional programmatic API for terms, groups and associativity on top of PEGs for the sole purpose of handling arithmetic grammars.

To this day it is still unclear if PEGs can successfully resolve these issues in a general way without introducing implementation directives external to the formalism. Moreover 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 extensions. The difference is that LALR needs the directives to avoid conflicts and ambiguity (as in the case of Earley), while 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 adopting external directives like their CFG counterparts. Afterall, the usage of compilcated informal ”meta-rules” by CFGs, was one of the initial argumentations for the introduction of PEGs [24].

Figure 4: Left-recursion extension for PEGs

```
<expression> ::= <sum>

<sum> ::=  <sum> "[+-]" <product>
        | <product>

<product> ::= <product> "[*/]" <pow>
        | <pow>

<pow> ::= <number> "^" <pow>
        | <number>

<number> ::= [0-9]+
```

Nevertheless, there are still valid reasons why one might choose to use PEGs over CFGs. 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. Such a grammar is likely to be well-behaved under CFG algorithms as well, but in the PEG case this is guaranteed.

Moreover 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 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, for which we may need to provide external directives to parse correctly.

Figure 5: Associativity solution for PEGs

```
<expression> ::= <expression> "[+-]" <expression>
        @+ @left_recur
        | <expression> "[*/]" <expression>
        @+ @left_recur
        | <expression> "^" <expression>
        | <number> @+

<number> ::= [0-9]+
```

In conclusion, neither PEGs nor CFGs (and their accompanying algorithms) are sufficient for a complete description of common cases arising in language design. In order to properly handle ambiguity, recursion, precedence or associativity, parsing frameworks either introduce implementation specific directives or ask users to refactor their grammars to fit the the needs of the framework/algorithm/formalism combo. This introduces significant complexity even in simple cases and results in incompatible grammar specifications.

3 OUR PROPOSAL

Since in the absence of external directives, both CFGs and PEGs cannot provide a complete solution, we might conclude that the last 15 years of research (since the introduction of PEGs in 2004) have come full-circle. Nevertheless as we saw in Section 2 PEGs did provide us with a means of ”disambiguation” (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 disambiguation 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. Given the insights that we gained from the PEG program, we introduce mechanisms for **incremental disambiguation** of languages through new ordered-choice operators that act within (not instead of) an ambiguous grammar.

An overview of the MOG operators and semantics can be seen in Tables 1 and 2. Table 1 lists the operators that are common in MOGs and other formalisms, whereas Table 2 describes MOG-specific operators, or operators that have significantly different semantics in Multi-ordered grammars. Figure 6 summarizes the class structure of the Gray parsing algorithm used to recognize MOGs, its relation with chart parsing and the incremental introduction of the new operators in a chart-parsing base. A detailed analysis of the Gray algorithm follows in sub-section 3.1.

Depending on your point of origin (CFGs or PEGs), MOGs can be loosely described as, either:

$$MOG = PEG + Unordered_c + RecScopedOrdered_c \tag{1}$$

That is a Multi-Ordered Grammar is a PEG augmented by unordered and recursively-scoped ordered choice operators, or as:

$$MOG = CFG + LAhead_o + Ordered_c + RecScopedOrdered_c \tag{2}$$

That is a Multi-Ordered Grammar is a CFG augmented by two lookahead operators $LAhead_o$ (& and !), plus the ordered and recursively-scoped ordered choices. In essence what is unique about MOGs compared to CFGs or PEGs is:

1. The experimental mixing of $Ordered_c$ and $Unordered_c$ choices, that are mutually exclusive in other formalisms.

2. The $RecScopedOrdered_c$ (*Recursively Scoped Ordered*) choice operators, that are unique to MOGs.

These relations between Multi-ordered Grammars, CFGs and PEGs can be more easily understood through the inheritance semantics of the Gray algorithm seen in Figure 6. The Gray hierarchy consists of a succession of recognizers beginning with a CFG-recognizer (*GrayBaseAlgo*, at the base of the hierarchy), followed by a PEG-compatible recognizer (*GrayOrdered*) and finally a recognizer that is able to handle MOGs (*GrayMixedOrdered*).

More specifically in the bottom part of Figure 6, we see the *GrayBaseAlgo* class. *GrayBaseAlgo* is our chart-parsing base similar to that of a 3-op Earley parser (scan, predict, complete) [20], traditionally used in NLP [26]. Similarly to [3], we extend this chart-base with the empty derivation (ε). In Gray this is achieved by treating ε as a special terminal of zero length that *unconditionally succeeds*. By introducing ε-rules in the chart base, we can then trivially define EBNF operators such +, *, ?, through recursion terminating at ε. For example `<s> ::= <a> +`, can be readily consumed as `<s> ::= <a> (<s> | <e>)`, with the group operation between () defined as an intermediate rule `<t> ::= <s> | <e>`. *GrayBaseAlgo* maintains a reference to the initial input, the grammar and the charts describing the current state of the recognition (its methods *scan, predict, complete* etc. will be described in more detail in Subsection 3.1). Notice here that almost all base methods have synonyms (*scanBase, predictBase, completeBase*), allowing the default parsing behavior (CFG-compatible recognition) to be accessed by the sub-classes, even if the main methods have been overridden somewhere along the hierarchy chain.

In the middle part of Figure 6 we see the *GrayOrdered* class, which models the PEG-compatible recognizer of the Gray hierarchy. *GrayOrdered*, extends our chart-parsing base with two new operations (*backtrack* and *fork*), while overriding the standard CFG behaviour for *predict* and *complete*. These extensions (described in more detail in Subsection 3.1) handle the non-scoped backtracking ordered choice (||) and the two lookahead operators (& , !), by maintaining a *backTrackStack*. Notice here that backtracking for this recognizer is ultimately supported by the *Charts* class at the bottom, through the *backTrackAt(chartIndex,stateIndex,state)* method.

Finally, in the upper part of Figure 6, we find the MOG-compatible GrayMixedOrdered recognizer. *GrayMixedOrdered* maintains a *lastSeenStack*, that remembers the current ordered alternative for each ordered rule we are seeking to recognize. The variable references a stack of values rather than a single value, taking into account that alternatives can be recursively invoked and backtracked. This book-keeping allows *GrayMixedOrdered* to implement recursive ordering (/ , \) and introduce a new recursive scope upon invocation of a scoped ordered choice (||). The mixing of order with unordered choices, is achieved by overriding the *predict* and *complete* operations. As we will see in more detail in Subsection 3.1, *GrayMixedOrdered* invokes either the PEG or CFG-compatible operations of each of its parent classes (*e.g., GrayOrdered's predict(state)* or *GrayBaseAlgo's predictBase(state)*) depending on whether the rule under consideration is ordered or not. To optimize scanning and memoization, we pre-compute all first, follow and predict sets of the grammar rules to pre-filter unwanted alternatives (as seen in *GrayMixedOrderedFiltered* class) by overriding the *predict* operation.

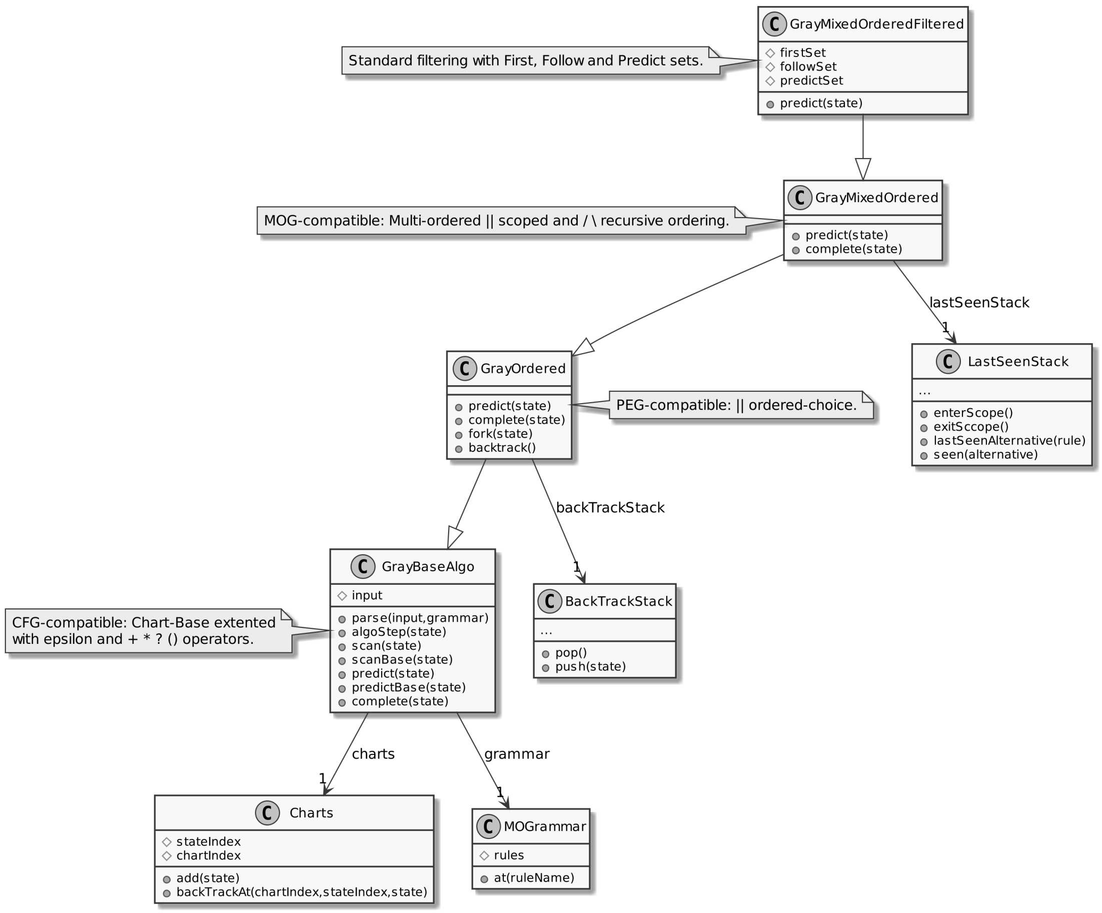


Figure 6. Gray Hierarchy Overview: Extending a chart-base to parse Multi-Ordered Grammars

3.1 The Gray Algorithm

The Gray algorithm is a chart-parsing algorithm employing *dynamic programming*, to memoize intermediate results, thus avoiding excessive re-parsing of the input [34]. What distinguishes Gray from other chart-based algorithms and especially the Earley algorithm [20, 21], is its ability to:

(a) **Handle ε-rules gracefully**, implementing all EBNF operators (+, *, ?, ())

(b) **Backtrack and fork**, to express the PEG-compatible ordered-choice and the two look-ahead operators (& and !).

Common Operators	Appears In	Description
`<S> ::= <A> <B>`	**MOG**, EBNF, PEG, RE	**Composition operator**. Non-terminal S is a sequence of exactly one occurrence of A, followed by exactly one occurrence of B.
`<S> ::= <A> <B> ?`	**MOG**, EBNF, PEG, RE	**Optional operator**. Non-terminal S is a sequence of exactly one occurrence of A, followed by an optional occurrence of B.
`<S> ::= <A> <B> *`	**MOG**, EBNF, PEG, RE	**Zero-or-more operator**. Non-terminal S is a sequence of exactly one occurrence of A, followed by zero or more occurrences of B.
`<S> ::= <A> <B> +`	**MOG**, EBNF, PEG, RE	**One-or-more operator**. Non-terminal S is a sequence of exactly one occurrence of A, followed by one or more occurrences of B.
`<S> ::= (<A> <B>) +`	**MOG**, EBNF, PEG, RE	**Grouping operator**. Non-terminal S consists of one or more sequences of exactly one occurrence of A followed by exactly one occurence of B.
`<S> ::= <A> | <B>`	**MOG**, EBNF, BNF	**Non-deterministic (unordered) choice operator**. Non-terminal S consists of either exactly one occurrence of A or exactly one occurrence of B.
`<S> ::= <A> & <B>`	**MOG**, PEG	**Conditional-and operator**. Non-terminal S consists of exactly one occurrence of A, only if it followed by B. B is not consumed.
`<S> ::= <A> ! <B>`	**MOG**, PEG	**Conditional-not operator**. Non-terminal S consists of exactly one occurrence of A, only if it is not followed by B. B if present is not consumed.

Table 1. Operators common in MOGs and other formalisms

(c) **Recognize multi-ordered grammars** (MOGs) consisting of both deterministic and non-deterministic alternatives, augmented by **scoped** (||) **and recursively ordered-choices** (/ **or** \).

The Gray chart consists of a two-dimensional memory Π, filled with $\pi_{i,j}$ entries of unique (non-duplicate) *dotted-rules*. The i-dimension, denotes the i-th terminal in the recognition process, while the j dimension denotes the j-th state (*dotted-rule*) processed during the recognition of the aforementioned terminal. Dotted rules $\pi_{i,j}$ are grammar rules, augmented by (i) their starting position α in the input I, (ii) a dot . on the left of the currently processed sub-rule and (iii) the dot position ω in the input. As an example consider the dotted rule `<e> → <e> . "+" <e> [0,3]`, describing the recognition attempt of e starting at $\alpha = 0$, having successfully recognized its recursive invocation up until position $\omega = 3$ and waiting to process the terminal rule ”+”. A state with a dot after its last constituent (for *e.g.*, `<e> → <e> "+" <e> . [0,8]`), is said to be completed/recognized. $C(\pi_{i,j})$ returns $true$ if $\pi_{i,j}$ is complete or $false$ otherwise. Γ denotes the set of rules describing the grammar to be recognized, with γ_k referring to each particular rule. γ_0 references the rule: $<\gamma_0>$ → `<S>` where S is the starting

MOG-Specific Operators	Appears In	Description
`<S> ::= <A> || <B>`	**MOG**, PEG (no-scope)	**Deterministic (ordered) choice operator**. Non-terminal S consists of exactly one occurrence of A. If recognition of non-terminal A fails then S consists of exactly one occurrence of B. Deterministic choice can be exhaustive in MOGs (successful recognition of A should lead to successful recognition of the input). || **introduces a new scope for recursive ordering that is MOG-specific (see below)**.
`<S> ::= <A> / <B>`	**MOG-ONLY**	**Self-recursive (ordered) choice operator**. Non-terminal S consists of exactly one occurrence of A. If recognition of non-terminal A fails then S consists of exactly one occurrence of B. **If S is to be recursively recognized from within A, start at A. If S is to be recursively recognized from within B start at B. If non-recursive ordered choice || is invoked from within S, start a new scope for S at A.**
`<S> ::= <A> \ <B> | <C>`	**MOG-ONLY**	**Recursive (ordered) choice operator**. Non-terminal S consists of exactly one occurrence of A. If recognition of non-terminal A fails then S consists of exactly one occurrence of B (similarly for C). **If S is to be recursively recognized from within A, start at A. But if S is to be recursively recognized from within B, start at the next alternative (*i.e.*, C).**
`<S> ::= <A> \ <B> | <C>`	**MOGs-ONLY**	**Taint operation**. If either || or / or \are present in a rule, **the rule is tainted (*i.e.*, all alternatives are ordered). If the recursive order of a tainted sub-rule is not explicitly set, it defaults to self-recursion.** Here S consists of of a self-recursive A. If A fails it consists of a simply recursive B (explicitely set). If B fails it consists of a self-recursive C.

Table 2. MOG-specific operators, or operators that have significantly different semantics in Multi-ordered grammars

non-terminal of the language. $\Delta(\gamma_k)$ returns the index of γ_k as a sub-rule, and $\gamma \equiv \gamma'$ is *true* if γ and γ' are sub-rules of the same parent. $\gamma_{k_{id}}$ represents the name of rule γ_k, rather than the rule itself (from which it follows that $\gamma \equiv \gamma' \leftrightarrow \gamma_{id} = \gamma'_{id}$). Our backtrack stack B holds $\pi_{i,j}$ alternatives that will be added to Π in the order they where pushed, *only if* the current considered alternative in Π fails. Failures are memoized and identical states that have previously failed are ignored. Λ_s denotes a stack of scopes, with each scope s describing a mapping from a rule γ_i to the index of its alternative that is currently being processed by the algorithm.

Algorithm 1 The Gray Base Algorithm

```
1:  procedure GRAYBASEPARSE($\Gamma$, $S$, $\Pi$, $I$)
2:      $\Pi_{0,0} \leftarrow (\gamma_0 \rightarrow \bullet S\ [0,0])$
3:      for all $\pi_{i,j} \in \Pi$ do
4:          if $\neg C(\pi_{i,j})$ then
5:              if $K'(\pi_{i,j}) \notin (T(\Gamma) \cup \{\varepsilon\})$ then
6:                  GRAYBASEPREDICT($\pi_{i,j}, \Gamma, \Pi, I$)
7:              else
8:                  GRAYBASESCAN($\pi_{i,j}, I, \Pi$)
9:              end if
10:         else
11:             GRAYBASECOMPLETE($\pi_{i,j}, \Pi$)
12:         end if
13:     end for
14:     return $\Pi$
15: end procedure

16: procedure GRAYBASEPREDICT($\pi_{i,j}, \Gamma, \Pi, I$)
17:     for all $\gamma \in \Gamma$ where $\gamma \equiv K'(\pi_{i,j})$ do
18:         $\pi \leftarrow (\gamma_{id} \rightarrow \bullet \ldots\ [\Omega(\pi_{i,j}), \Omega(\pi_{i,j})])$
19:         if $\pi \notin \Pi$ then $\Pi_{i,max(j)+1} \leftarrow \pi$
20:     end for
21: end procedure

22: procedure GRAYBASESCAN($\pi_{i,j}, I, \Pi$)
23:     $\alpha \leftarrow \Pi_{index}(I)$ , $\omega \leftarrow \Pi_{index}(I, K'(\pi_{i,j}))$
24:     if $\omega > 0$ then
25:         $\pi \leftarrow (K'_{id}(\pi_{i,j}) \rightarrow \ldots \bullet\ [\alpha, \omega])$
26:         if $\pi \notin \Pi$ then $\Pi_{i+1,max(j)+1} \leftarrow \pi$
27:     end if
28: end procedure

29: procedure GRAYBASECOMPLETE($\pi_{i,j}, \Pi$)
30:     for all $\pi \in \Pi$ where $K'(\pi) \equiv K(\pi_{i,j}) \wedge \Omega(\pi) = A(\pi_{i,j})$ do
31:         $\pi' \leftarrow (K_{id}(\pi) \rightarrow \ldots K_{id}(\pi_{i,j}) \bullet \ldots\ [A(\pi), \Omega(\pi_{i,j})])$
32:         if $\pi' \notin \Pi$ then $\Pi_{i,max(j)+1} \leftarrow \pi'$
33:     end for
34: end procedure
```

We also define the following helper functions to aid our description: $K(\pi_{i,j}), K'(\pi_{i,j})$ return the rule γ_k at the head or at the dot of a state $\pi_{i,j}$. $T(\Gamma), N(\Gamma)$ denote respectively the set of terminals and non-terminal rules in Γ. $A(\pi_{i,j}), \Omega(\pi_{i,j})$ return the α and ω of the dotted-rule $\pi_{i,j}$. $P(I), P_{index}(I)$ return the current character and index of the input. While $P_{index}(I, \gamma_k)$ returns the new input index after the terminal rule γ_k has been recognized (or 0 otherwise). $Op(\gamma_k)$ returns the operator associated with the rule γ_k in the current context (&, !, | , ||, / or \). While $Ch_{index}(\gamma_k, i)$ maps the type of ordered choice $\{||, /, \backslash\}$ associated with a sub-rule at index i, to the integer indexes $\{-1, 0, 1\}$.

Algorithm 2 The Gray Ordered Overrides and Extensions

```
procedure GRAYORDPREDICT($\pi_{i,j}, \Gamma, \Pi, I$)
    if $Op(K'(\pi_{i,j})) \in \{\&, !\}$ then
        if $\neg$ GRAYORDFORK($\Pi, \pi_{i,j}, \Gamma, I$) then return
    end if
    for all $\gamma \in \Gamma$ where $\gamma \equiv K'(\pi_{i,j})$ do
        $\pi \leftarrow (\gamma_{id} \rightarrow \bullet \ldots [\Omega(\pi_{i,j}), \Omega(\pi_{i,j})])$
        if $\pi \notin \Pi \wedge \neg n$ then
            $\Pi_{i,max(j)+1} \leftarrow \pi$ , $n \leftarrow true$
        else
            $B_{i,max(j)+1} \leftarrow \{\pi, P_{index}(I)\}$                ▷ Push-op (optim. hook)
        end if
    end for
end procedure

procedure GRAYORDCOMPLETE($\pi_{i,j}, \Pi$)
    for all $\pi \in \Pi$ where $K'(\pi) \equiv K(\pi_{i,j}) \wedge \Omega(\pi) = A(\pi_{i,j})$ do
        $\pi' \leftarrow (K_{id}(\pi) \rightarrow \ldots K_{id}(\pi_{i,j}) \bullet \ldots [A(\pi), \Omega(\pi_{i,j})])$
        if $\pi' \notin \Pi \wedge \neg n$ then
            $\Pi_{i,max(j)+1} \leftarrow \pi'$ , $n \leftarrow true$
        else
            $B_{i,max(j)+1} \leftarrow \{\pi, P_{index}(I)\}$
        end if
    end for
end procedure

procedure GRAYORDBACKTRACK($\Pi, B, I$)
    $b \leftarrow B_{pop}$
    if $b \in \Pi$ then
        GRAYORDBACKTRACK($\Pi, B, I$)
    else
        $\Pi_{b_i,b_j} \leftarrow b[\pi]$                ▷ Del. or memoize the rest
        $P_{index}(I) \leftarrow b[P_{index}]$
    end if
end procedure

procedure GRAYORDFORK($\Pi, \pi_{i,j}, \Gamma, I$)
    $i \leftarrow P_{index}(I)$
    $\Pi' \leftarrow$ GRAYBASEPARSE($\Gamma, S \rightarrow K'(\pi_{i,j}), \Pi', I$)
    $s \leftarrow ((\gamma_0 \rightarrow S\bullet) \in \Pi')$
    $P_{index}(I) \leftarrow i$                ▷ Restore input
    if $(Op(K'(\pi_{i,j})) = \& \wedge s) \vee (Op(K'(\pi_{i,j})) = ! \wedge \neg s)$ then
        $\Pi_{i,j} \leftarrow (K_{id}(\pi_{i,j}) \rightarrow \ldots K'_{id}(\pi_{i,j}) \bullet \ldots [A(\pi_{i,j}), \Omega(\pi_{i,j})])$                ▷ Advance dot and continue
        return $true$
    else
        return $false$                ▷ Process next state
    end if
end procedure
```

3.1.1 *The GrayBase Algorithm*

Our chart-parsing base (Alg. 1) includes the main parsing loop (GrayBaseParse), which receives (in line 1) the grammar (Γ), the starting non-terminal (S), the charts (Π) and the string to recognize (I), as input. At the end of this procedure the filled charts will be returned, from which the parsing tree(s) can be derived, with the existence of a completed γ_0 state ($\gamma_0 \rightarrow S\bullet \in \Pi$) signaling success. As we saw earlier,

Algorithm 3 The Gray Mixed-Ordered Overrides

```
1:  procedure GrayMixedPredict($\pi_{i,j}, \Gamma, \Pi, I$)
2:      if $Op(K'(\pi_{i,j})) = |$ then
3:          GrayBasePredict($\pi_{i,j}, \Gamma, \Pi, I$)
4:      else
5:          if $Op(K'(\pi_{i,j})) \in \{\&, !\}$ then
6:              if $\neg$ GrayOrdFork($\Pi, \pi_{i,j}, \Gamma, I$) then return
7:          end if
8:          $\gamma'_{index} \leftarrow \Lambda_{max(s)}[K'(\pi_{i,j})]$
9:          $op_{index} \leftarrow Ch_{index}(K'(\pi_{i,j}), \gamma'_{index})$
10:         if $op_{index} = -1$ then
11:             $alt_{index} = 1$
12:         else
13:             $alt_{index} = \gamma'_{index} + op_{index}$
14:         end if
15:         for all $\gamma \in \Gamma$ where $\gamma \equiv K'(\pi_{i,j}) \wedge \Delta(\gamma) \geq alt_{index}$ do
16:             $\pi \leftarrow (\gamma_{id} \rightarrow \bullet \ldots [\Omega(\pi_{i,j}), \Omega(\pi_{i,j})])$
17:             if $\pi \notin \Pi \wedge \neg n$ then
18:                 $\Pi_{i,max(j)+1} \leftarrow \pi$ , $n \leftarrow true$
19:                 if $Ch_{index}(\gamma, \Delta(\gamma)) = -1$ then $\Lambda_{max(s)+1} \leftarrow \{\}$        ▷ New Scope
20:             else
21:                 $B_{i,max(j)+1} \leftarrow \{\pi, P_{index}(I)\}$
22:             end if
23:         end for
24:     end if
25: end procedure

26: procedure GrayMixedComplete($v$, $u$, $p$)
27:     if $Op(K'(\pi_{i,j})) = |$ then
28:         GrayBaseComplete($\pi_{i,j}, \Gamma, \Pi, I$)
29:     else
30:         GrayOrderedComplete($\pi_{i,j}, \Gamma, \Pi, I$)
31:         $\pi \leftarrow \Pi_{max(i),max(j)}$
32:         if $C(\pi) \wedge Ch_{index}(K(\pi), \Delta(K(\pi))) = -1$ then $\Lambda_{pop}$        ▷ Exit Scope
33:     end if
34: end procedure
```

this base algorithm deviates from other chart-based approaches by handling ε-rules as terminals (line 5), allowing us to implement all EBNF operators (+, *, ?, ()). Another notable difference is the complete absence of a separate lexing-phase, with the terminal recognition completely driven by the syntax (in lines 8 and 23). This is also the reason why both the α and ω indexes describing the recognition progress of dotted-rules, are defined in terms of character positions rather than terminals.

More specifically, starting at line 2 we add our first dotted-rule (γ_0) to our charts at position 0, describing the attempt to recognize the starting non-terminal of the grammar (*i.e.,* the $\bullet$ is at the left of S). Then from lines 3 to 13 we will iterate over all dotted-rules in the charts, calling one of the three base methods (grayBasePredict, grayBaseScan, grayBaseComplete) for each state. The reason we need to loop (despite having started with just the initial γ_0 state), is that all three methods add additional states to Π during recognition. The decision on which method to call depends on the state of each dotted rule, so that if $\pi_{i,j}$ is not complete (line 4) we will either call predict (line 6) or scan (line 8), depending on whether the rule at the left of the $\bullet$ is a terminal or not (line 5). If $\pi_{i,j}$ has instead been completed (*i.e.,* has been recognized), we will call the complete method instead (line 11).

GrayBasePredict (lines 16 to 21), will expand all alternatives of the rule at the left of the dot in $\pi_{i,j}$ (lines 17, 18). Then it will store them at the end of the current chart $\Pi_{i,max(j)+1}$ (line 19), if they have not already been added by a previous prediction (*i.e.,* all identical states in a specific position will be analyzed exactly once). GrayBaseScan (lines 22 to 28), will store the current input position (in α) and attempt to recognize the terminal left of the $\bullet$ in $\pi_{i,j}$, storing the resulting position in ω (line 23). If the recognition of the terminal was successful (*i.e.,* $\omega > 0$), a completed state for the terminal rule (with an end-$\bullet$ in line 25) will be added at the end of the next chart $\Pi_{i+1,max(j)+1}$. Finally, GrayBaseComplete (lines 29 to 34) will search for states in Π that are waiting for the completed state $\pi_{i,j}$ at position $A(\pi_{i,j})$ (line 30). It will then create a copy of the waiting states, advancing the $\bullet$ and ω of the copy at the right of the recognized rule (line 31). Finally it will attempt to add the updated copy in the current chart $\Pi_{i,max(j)+1}$, checking for duplicates (line 32).

3.1.2 The GrayOrdered Algorithm

The GrayOrdered algorithm (Alg. 2), overrides the base predict and complete methods with the grayOrdPredict and grayOrdComplete procedures (lines 1 to 23). Moreover, it adds two additional operations to the chart-parsing base (backtrack and fork, lines 24 to 39). Essentially, these extensions treat all choice operator as non-scoped versions of the ordered choice ($||$) while implementing the two look-ahead operators ($\&, !$). This is achieved by maintaining the backtrack stack B of $\pi_{i,j}$ alternatives, and introducing a *single alternative of ordered rules at a time*. If said alternative fails, Π will backtrack at the top alternative of B. Failures can be memoized so that identical states that have previously failed can be ignored.

More specifically, at line 2 the overridden predict checks the operator associated with the rule at the left of the dot in the current state $Op(K'(\pi_{i,j}))$. If this is found to be one of the look-ahead operators (that does not consume input), the algorithm will fork (lines 28 to 39) to deal with this rule. The fork operator first stores the current input index $P_{index}(I)$ (line 34), and then enters a nested parsing loop (line 35), with a new empty chart Π', the same grammar Γ and input I, but different starting rule. This starting rule $S \rightarrow K'(\pi_{i,j})$ is the rule of the initial invoking loop, which we wanted to look-ahead. Then on line 31 we will check if the nested parsing succeeded or not, by testing for the existence of a completed starting rule ($\gamma_0 \rightarrow S\bullet$) storing the result in s. Finally on lines 33 and 34 we will advance the dot *in place* ($P_{i,j}$) at the right of the rule $K'_{id}(\pi_{i,j})$, only if the look-ahead operator matches the result of the nested parsing (at which case we will return $true$ on line 35) or $false$ otherwise (line 37).

Back to the overridden predict on line 3, if the look-ahead failed, the method will return so that the algorithm can continue parsing at the next state in the chart. If the look-ahead succeeded (with the dot advancing in-place during fork) we will proceed with the prediction normally. Ordered prediction (lines 5 to 12) consists of attempting to add a single alternative for $K'(\pi_{i,j})$ to Π making sure that (a) it does not already exist ($\pi \in \Pi$) and (b) one has not already been added ($\neg n$), as seen in line 7. The single alternative to be considered ends up at the end of the current chart $\Pi_{i,max(j)+1}$ (line 8). The remaining alternatives will be pushed at the top of B, marking the current input index $P_{index}(I)$ to return to (line 10), as well as the current chart indexes $(i, max(j)+1)$. The push operation on line 10, also serves as a hook for failure memoization (*i.e.,* identical states that have already failed do not need to be added to the stack).

The overridden complete (lines 14 to 23) will first search (line 15) for those states in Π that are waiting for the completed state $\pi_{i,j}$ at position $A(\pi_{i,j})$. As before, for each waiting state it will create a copy (line 16) advancing the $\bullet$ and ω. Then, similarly to the overridden predict it will attempt to add *a single alternative* at the end of the current chart $\Pi_{i,max(j)+1}$ (line 18), while pushing the rest at the top of B (on line 20, marking input and chart indexes to return to). Finally during backtrack (lines 24 to 32, for failed predicted/completed states), the algorithm will check (line 26) if the popped backtrack state (line 25) is already in Π and recursively continue backtracking (line 27). Else, it will rewind Π to the stored b_i, bj indexes, over-writing the previous alternative (lines 29, 30) and restoring the input index to $b[P_{index}]$ (previously stored on line 20).

3.1.3 The GrayMixedOrder Algorithm

The GrayMixedOrdered algorithm (Alg. 3), extends both the base and ordered versions of Gray, to (a) allow the co-existence of both deterministic and non-deterministic ordering in the grammar and (b) implement the MOG-only recursive and scoped ordering operators ($||$, $/$ and $\backslash$). The mixing of deterministic with non-deterministic choices, is achieved by overriding the *predict* and *complete* operations to delegate execution to either their base (non-deterministic) or ordered (deterministic) counterparts, depending on whether a rule is ordered or not. The additional scoped and recursive operators are implemented by maintaining the Λ stack (*lastSeenStack*), that remembers the current ordered alternative for each rule. Λ consists of a stack of scopes that registers which alternatives have been seen, taking into account that rules can be recursively invoked and backtracked.

More specifically, starting at line 2 we check the operator $Op(K'(\pi_{i,j}))$ that is associated with the rule at the left of the $\bullet$, in the current state $\pi_{i,j}$. If the rule is unordered we delegate prediction to the base-predict procedure (line 3), otherwise we proceed (lines 5 to 24) with an extended version of the ordered case. As before (in GrayOrdPredict), we begin (lines 5 to 7) by handling the look-ahead operators. If there is a look-ahead operator at the current $\bullet$ position and the look-ahead succeeded (with the dot advancing in-place during fork) we will proceed with the prediction (lines 8 to 24), otherwise we will return to handle the next state in the chart.

Ordered prediction in the mixed case first determines the value of γ'_{index} (at the current $\Lambda_{max(s)}$ scope) which represents the index of the last seen alternative of $K'(\pi_{i,j})$. Then for that particular index it will retrieve the integer value $Ch_{index}(K'(\pi_{i,j}), \gamma'_{index})$ associated with the scoped or recursive choice of the $K'(\pi_{i,j})$ sub-rule. Remember here that Ch_{index} maps the type of ordered choice $\{||, /, \backslash\}$ to the integer indexes $\{-1, 0, 1\}$ respectively. We can thus now calculate the index (alt_{index}) of the alternative that should be recursively considered. If the alternative that was last seen was that of a scoped ordered choice (*i.e.*, $op_{index} = -1$), the recursive invocation re-starts at the top of the rule $(alt_{index} = 1)$. If the op_{index} is either 0 or 1 (for self-recursive or simply recursive ordered choices) the next alternative will be at $alt_{index} = \gamma'_{index} + op_{index}$. That is for the self-recursive case we will re-start invocation by repeating the last seen alternative $(alt_{index} = \gamma'_{index})$. While for the simply recursive ordered choice we will continue parsing at the next alternative $(alt_{index} = \gamma'_{index} + 1)$. Mixed-ordered prediction will proceed (lines 15 to 23) as before, by attempting to add a single alternative for $K'(\pi_{i,j})$ to Π (line 18) while the rest of the alternatives will end-up in the backTrack stack B (line 21). What is different from the ordered case is that (a) only the alternatives whose index satisfies $\Delta(\gamma) \geq alt_{index}$ (line 15) will be considered (implementing as we saw above the semantics of scoped and recursive choices) and (b) in the case where the single alternative that was added in Π is associated with a scoped ordered choice $(Ch_{index}(\gamma, \Delta(\gamma)) = -1)$, a new scope will be created (line 19) at the top of Λ (our last-seen stack of alternatives).

Finally the mixed-ordered complete (lines 26 to 34), will first check for the choice operator associated with $K'(\pi_{i,j})$ and invoke either the base version of complete (line 28) or the ordered one (line 30). In the latter case, it will also check if the ordered alternative that has just been added is complete (through $C(\pi)$), and if said alternative was a scoped ordered choice: $Ch_{index}(K(\pi), \Delta(K(\pi)))$, previously introduced at line 19. If both conditions are true then the current scope has been successfully recognized and can thus be removed from Λ (line 32). The backtrack operation in the mixed-order case is essentially the same as before with the additional step of restoring Λ, at the backtrack index.

4 RESULTS & DISCUSSION

We conducted two case-studies to assess the expressiveness of MOGs, compared to CFGs and PEGs. The first consists of two idealized examples from literature (an expression grammar and a simple procedural language). The second examines a real-world case (the entire Smalltalk grammar and eleven new Smalltalk extensions) probing the complexities of practical needs. All MOG-based examples in this section are readily reproducible, by simply downloading and running the alpha-version of the Lan.d.s platform at: `https://npapoylias.gitlab.io/lands-project/`. This on-line portal is dedicated to Multi-ordered grammars and hosts several additional examples that you can explore. The Lan.d.s project is currently implemented on top of the Pharo [9] and Moose platforms [19], but only its visualization and code-generation sub-systems (that depend on the Roassal [7] and Opal frameworks [6]), are specific to the Pharo ecosystem. Both the MOG formalism and the Gray algorithm are language agnostic. The source-code is distributed under the MIT License, and is available at: `https://gitlab.com/npapoylias/lands`.

4.1 Case-study I: Expressions and Control-Flow

Figures 7 and 9 show us the expression language defined through a Multi-Ordered Grammar. These expression rules are both valid MOG-rules, recognized using the same algorithm (Gray). Their only difference is their choice of operators (beginnings of lines 2 to 6), that can be lively edited from within our environment. Figure 7 shows us a full non-deterministic MOG-rule (all alternatives are unordered), that can be used to explore ambiguities arising from the structure of the grammar. Figure 8 depicts what this exploratory parsing looks like inside the Lan.d.s platform (this is a still shot of an interactive session), while parsing the expression: $(2 * 3 \wedge 4 \wedge 5) + (6 * 7/8)$. The parenthesized expression in the left has 5 alternatives in total that can be explored. Two of them are readily accessible in the current configuration of the sub-tree. The other three can be viewed by navigating through the up/down arrows on the composite nodes of the left side. Similarly the right-parentheses has two alternatives that can be explored, with a total of $5*2=10$ alternatives for the whole expression. Notice here that these ambiguous alternatives essentially present us with all possible precedence and associativity configurations of the numerical expression.

Figure 7: The Explorative MOG for Expressions (1/2)

```
<expression> ::=
                    | <expression> "[+-]" <expression>
                    | <expression> "[*/]" <expression>
                    | <expression> "^" <expression>
                    | "(" <expression> ")"
                    | <number>
<number> ::= [0-9]+
```

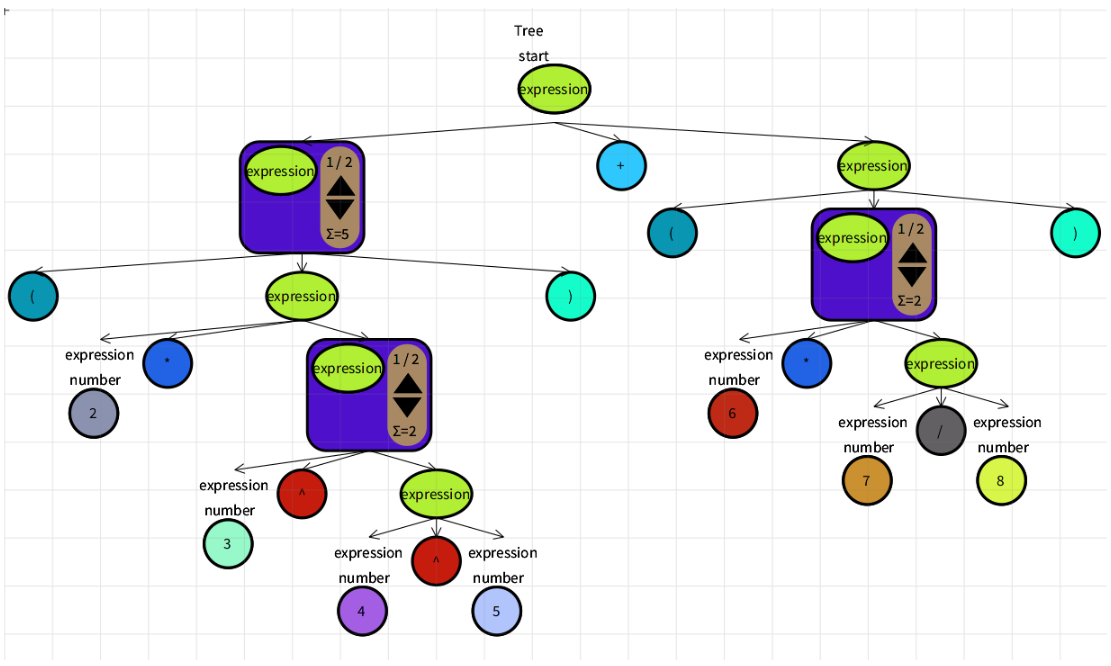


Figure 8. Parsed View: The MOG Expr. Grammar (1/2)

Subsequently, through experimentation and live-editing of the choice operators we end up with the MOG-rule of Figure 9, that produces the correct unambiguous parsing tree seen in Figure 10. This incremental live-editing is what we call *incremental disambiguation*. As seen in Figure 9, incremental disambiguation is based on the recursively scoped MOG-operators that we introduced (|| , / and \), acting within (not instead of) an ambiguous grammar. In lines 2 and 3 of Figure 9 the $[+-]$ and $[*/]$ alternatives are declared *simply-recursive*. As we discussed in Section 3.1, this means that the alternatives are ordered and that any expression recursively invoked from within line 2, will continue from the alternative at line 3 (similarly, recursive invocation from within line 3 continues at 4). Since expressions already include ordered alternatives, the rest of the rule is *tainted*, with non-ordered sub-rules defaulting to self-recursion. This is the case for the $\wedge$ (exponentiation) alternative, which by being self-recursive will recurse on itself for expressions invoked from within line 4. The last alternative (number rule on line 6) is typically also self-recursive, but with no explicit or implicit recursion from line 6, only its order in the rule matters. Finally the grouping operator () in line 5 is declared as *recursively scoped*, which as we saw means that a new recursive scope will be pushed before and popped after its successful recognition. Recursive invocation of expressions from within line 5 will begin again (in a new scope) at line 2.

What the rule ordering and the semantics of the MOG operators achieve here is to: (a) Define the *precedence* of numerical expressions (the lower we are in a rule, the higher the precedence).(b) Properly handle associativity ($[+ - * /]$ are left-associative, while $\wedge$ is right-associative), while (c) allowing rules with both left and right recursion. And finally (d) facilite the exploration of ambiguities (seen in Figure 7) and their *incremental disambiguation*.

Indeed any other combination we may have wished for precedence or associativity can be lively explored by re-ordering the alternatives or editing the choice operators (*e.g.,* a left-associative $\wedge$, can be achieved by simply switching the tainted self-recursion of line 4 to a simply-recursive choice $\backslash$).

Figure 9: The Unambiguous MOG for Expressions (2/2)

```
1  <expression> ::=
2                      \ <expression> "[+-]" <expression>
3                      \ <expression> "[*/]" <expression>
4                      | <expression> "^" <expression>
5                      || "(" <expression> ")"
6                      | <number>
7  <number> ::= [0-9]+
```

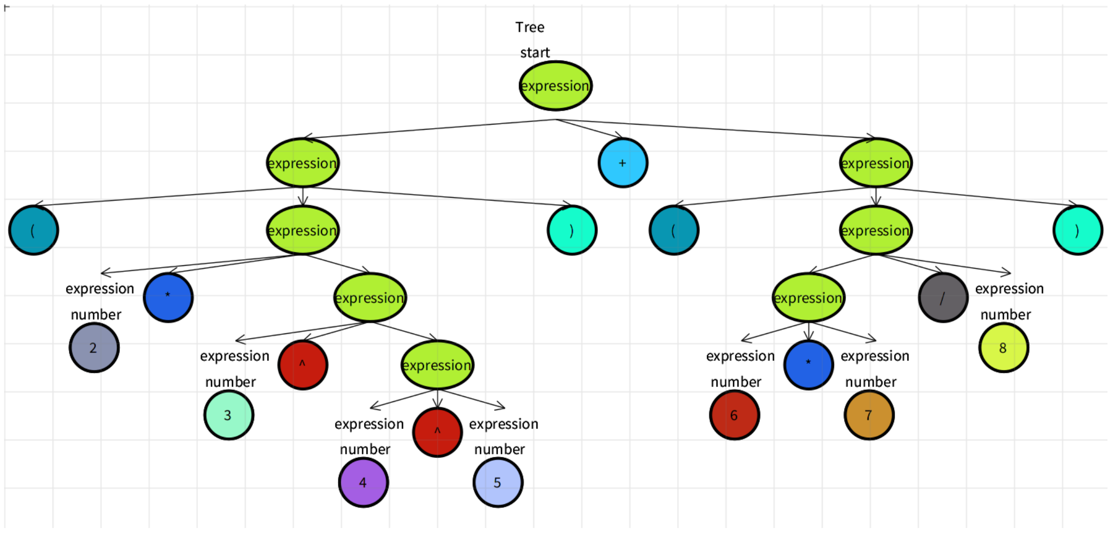


Figure 10. Parsed View: The MOG Expr. Grammar (2/2)

Comparing with state-of-the-art solutions for CFGs and PEGs (discussed in Section 2), the MOG version here is able to handle ambiguity, recursion, precedence and associativity without resorting to external directives or asking users to refactor their grammars. In terms of expressive power and clarity, this MOG version is only comparable to the Earley CFG (in Figure 1). The difference being that MOGs have a well-defined incremental way to resolve ambiguities through the choice operators, without re-writing the grammar. Specifically for Earley the grammar needs to be re-written in a way resembling the PEG version of Figure 3.

Moreover, if we take practical considerations into account (seen in Figure 11) the external directives for CFGs of Figure 2, although work as advertised for the expression grammar (seen in actual use in lines 16 to 18 of Figure 11 for the SmaCC framework [10]), do not scale well in larger cases (see sub-section 4.2). For PEGs, apart from the directives we saw in Figure 5 of Section 2 and the inherent inability to explore ambiguities, we may also be presented with API artifacts as those seen in lines 1 to 11 of Figure 11. Here the PEG-based PetitParser framework [36, 33], is forced to introduce an additional programmatic

API for terms, groups and associativity on top of its PEG back-end, for the sole purpose of properly handling arithmetic grammars. In essence, the PEG formalism needs to be completely encapsulated so that (as J. Kegler notes in [28]) the proper arithmetic semantics are reproduced in a second hard-coded pass.

Figure 11: Additional Considerations for PEGs and CFGs

```
"PParser, REPL/Class based:"
expression := PPExpressionParser new.
parens := $( asParser token trim , expression , $)
                asParser token trim.
number := #digit asParser plus flatten trim.
expression term: parens / number.
expression group: [ :g | g right: $^ asParser token trim ];
        group: [ :g | g left: $* asParser token trim.
                      g left: $/ asParser token trim];
        group: [ :g | g left: $+ asParser token trim.
                      g left: $- asParser token trim].

"SmaCC, File based:"
'<number> : [0-9]+ ;
<whitespace> : \s+;
%left "+" "-";
%left "*" "/";
%right "^";
Expression
        : Expression "+" Expression
        | Expression "-" Expression
        | Expression "*" Expression
        | Expression "/" Expression
        | Expression "^" Expression
        | "(" Expression ")"
        | Number
        ;
Number
        : <number>
        ;'
```

4.1.1 The Calc Grammar

Incremental disambiguation can be aplied in larger examples as well (as we show in Figure 12). Here we define the three main MOG-rules of a small procedural language that is comprised of assignments, conditionals, loops, print statements and code-blocks (lines 1 to 6). As before we started with a fully unordered grammar and worked our way towards an unambiguous definition. As seen in Figure 13, we had two different kinds of ambiguities co-occuring (while parsing the input: $if(x)\ if(y)\ z = x/2\ else\ z = 3*x+1$). The first one on the top of the figure, is related to the dangling-else ambiguity, while the one in the bottom is an expression ambiguity similar to those we saw previously. It is worth noting that not all rules in the grammar needed to be ordered for complete disambiguation (as seen in Figure 12). In contrast with PEGs, MOGs can seamlessly mix the unordered statement rule (lines 1 to 6) with the ordered conditional and expression rules (lines 7 to 20) without forcing the user to make ordering choices where there is no need to.

The expression rule on lines 10 to 20 is an extension of our previous example, with 8 more operators in order of precedence (similar to those found in languages like C). All additions (lines 10 to 12) are simply-recursive, resulting in them being left-associative as expected. The conditional rule on lines 7 and 8 (Figure 12) shows us how to handle the dangling-else ambiguity with MOGs. Our goal here is to match "else" statements only with inner if constructs. We thus first try to match all outer bare-if statements (as seen on line 7). The logic here is completely equivalent to that of the expression case where we were trying to first match the outer $[+-]$ operators that have lower precedence. The sub-rule in line 7 is tainted (since line 8 has an ordered scope) defaulting to self-recursion. This means that both if and if/else statements (in that order) can correctly (recursively) occur from within the if statement of line 7. On line 8 the if/else sub-rule is recursively scoped meaning that although the sub-rule is at the end, recognition can resume from the top if needed (which is the case when we have a bare-if statement after the else). Of course, as we did before, we can deduce the correct definition for conditions that will complete the disambiguation of Figure 13 (resulting in the parsing tree of Figure 14) by lively re-ordering and editing our MOG operators.

With the aid of Figure 15 we can contrast this result with the way PEGs (in [24]) and CFGs (in [17]) solve the dangling else problem. MOGs handle the ambiguity at least as naturally as PEGs (lines 3 to 4) with the main difference being that MOGs are fully explorable. This is achieved either by using an unordered choice to navigate the parsing trees or by re-ordering and editing the choice operators. This is not possible with PEGs since (by construction) they cannot handle ambiguity. Specifically for this case an alternative ordering for the PEG version (*i.e.,* switching lines 3 and 4) will completely mask the bare-if alternative. This is a known issue with PEGs documented by Ford himself [24]. In a PEG-rule of the form $A \rightarrow a|ab$ the second alterantive will never be considered, since upon every recognition of a, the A rule will always eagerly succeed. The CFG version on the other hand, has to use one of the directives seen in lines 8 to 11 to handle ambiguities, but not without some well documented caveats (described by Donnelly and Stallman in [17]). First, if we use the external %expect directive in line 8, we are essentially instructing the algorithm to expect n shift/reduce conflicts (and to shift by default). Yet, we are not guaranteed that these n conflicts we expect are those n that will actually occur during parsing, and we run the risk of mishandling some other ambiguity [17]. In the case of the %precedence (lines 9 and 10) and %right (line 11) directives both precedence and associativity is set globally (*i.e.,* in relation with all other operators in the grammar) and can thus cause problems even in simple cases (such as "$if\ test\ then$ 1 $else$ $2+3$") if scope is not taken into account (as noted in [17]).

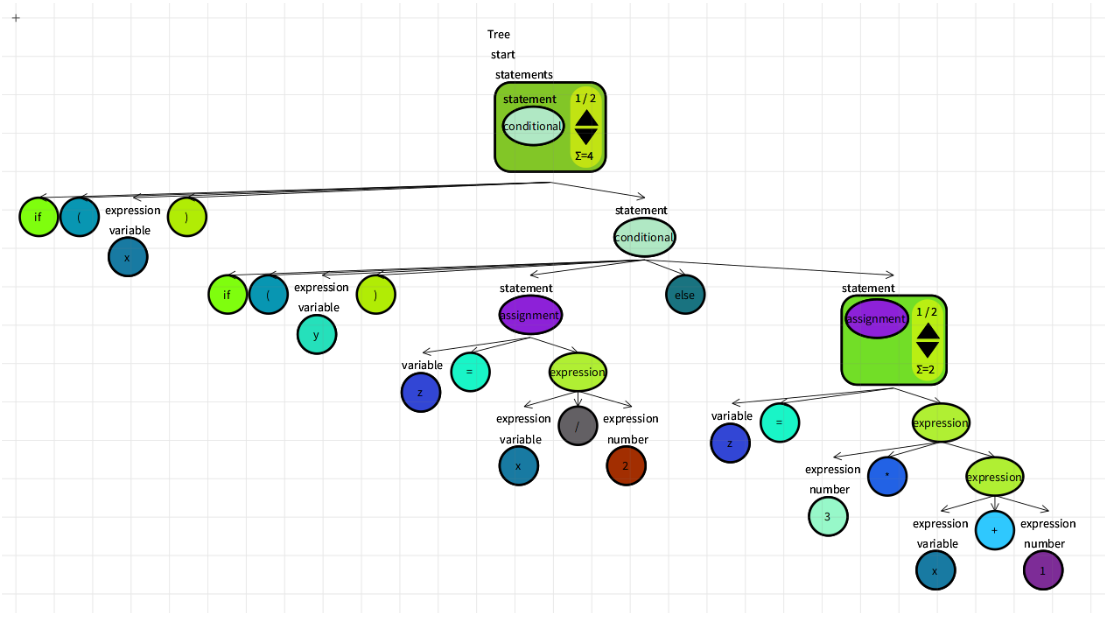


Figure 13. Parsed View: The MOG Calc Grammar (1/2)

Figure 12: The MOG Calc Grammar

```
<statement> ::= <assignment>
          | <conditional>
          | <loop>
          | <print>
          | <block>
...
<conditional> ::= "if" "(" <expression> ")" <statement>
|| "if" "(" <expression> ")" <statement> "else" <statement>
...
<expression> ::= \ <expression> ("|"|"&") <expression>
          \ <expression> ("=="|"~=") <expression>
          \ <expression> ("<="|">="|">"|"<") <expression>
          \ <expression> ("+"|"-") <expression>
          \ <expression> ("*"|"/"|"%") <expression>
          | <expression> "^" <expression>
          || "(" <expression> ")"
          | "true"
          | "false"
          | <variable>
          | <number>
```

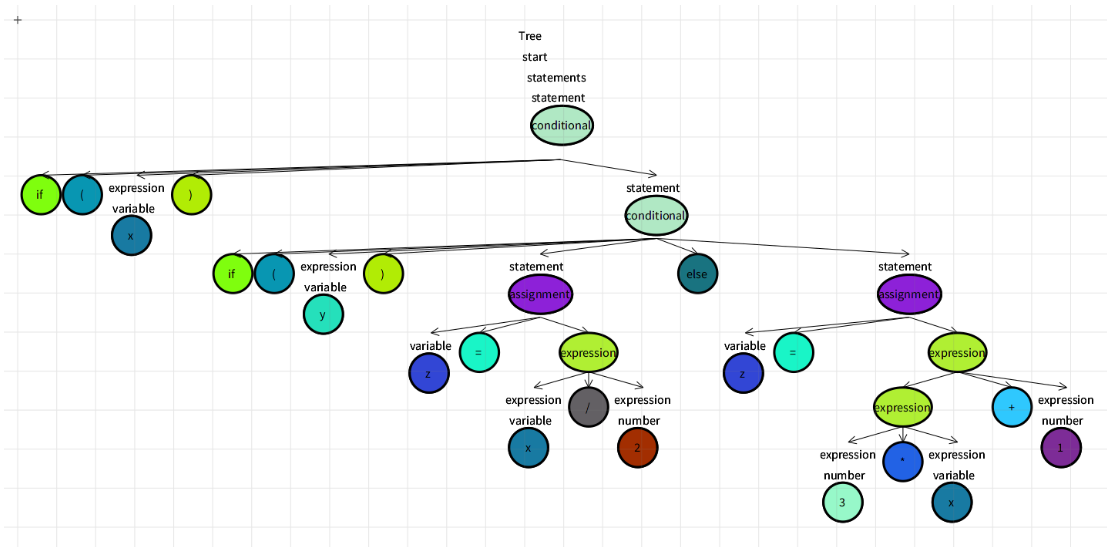


Figure 14. Parsed View: The MOG Calc Grammar (2/2)

Figure 15: PEG and CFG handling of dangling-else

```
... B. Ford 2004

IF Cond THEN Statement ELSE Statement
/ IF Cond THEN Statement

... Gnu/Bison, one directive + caveats

%expect n
%precedence "then"
%precedence "else"
%right "then" "else"
stmt:
  expr
| if_stmt
;

if_stmt:
  "if" expr "then" stmt
| "if" expr "then" stmt "else" stmt
;

expr:
  "identifier"
;
```

4.2 Case-study II: Parsing and Extending Smalltalk

Thus far we have seen classic examples from literature for which we argued that MOGs are more flexible than CFGs/PEGs without needing to resort to external directives. We now turn our attention to a real-world case where external directives are even harder to apply and disambiguation of PEGs or CFGs can only be achieved through inflexible rule definitions that severely obfuscate the grammar.

This is the case of Smalltalk messages discussed in this section, although the problem we illustrate is more general. At its root is the need for describing recursive precedence of arbitary rules without any anchoring terminals (like arithmetic operators or the ”then”, ”else” keywords) with respect to which we can define ordering (see also discussion of non-operators in [17]). In order to easily follow the examples we will briefly explain here the role of messages in Smalltalk.

In Smalltalk everything is a message (including control structures, like if, while loops etc). There are three types of messages: unary, binary, and keyword messages. Unary messages have the highest priority and are parsed first, then binary messages are parsed followed by keyword ones. Messages are sent to receivers with attached arguments, in which case we talk of a *message-send*. As syntactic structures, messages allow for a more fluid way to express method invocations. For example, if in a language like Java you were to write:

```
emailService.sendTo(mail + attachement , contact.address())
```

This would be expressed in Smalltalk as follows:

```
emailService send: mail + attachment to: contact address
```

Notice here that we did not need to parenthesize an empty argument list for `contact address` to invoke the address method, since #address is simply a unary message, with the highest parsing precedence, taking no arguments. Similarly the + operator (in `mail + attachment`), is just a binary message (ie a #+ is sent to 'mail' with 'attachement' as an argument). Finally, the results of the two aforementioned message-sends will serve as arguments for the keyword message `send: arg1 to: arg2` invoked with `emailService` as a receiver. In case we needed to send multiple emails we would use what is called a cascade (*i.e.,* separated messages with a semicolon as follows):

```
emailService send: mailA + attachmentA to: contactA address;
             send: mailB + attachmentB to: contactB address
```

Messages (besides primitive expressions) accept other messages as arguments, always following the *unary* > *binary* > *keyword* precedence. This recursive precedence rule (that does not involve any clear anchoring terminals) is naturally expressed in MOGs as follows:

Figure 16: The MOG Smalltalk Msg-Sends

```
1  ...
2  <msgSend> ::= <expression> <message> (";" <message>) *
3  <message> ::=
4            \ ( <keyword> <expression> ) +
5            \ <binaryOp> <expression>
6            | <identifier>
```

In line 2 of Figure 16 we define the syntax of msgSends which consist of an expression (that plays the role of the receiver) followed by a message. Then optionally this receiver can be sent multiple messages separated by commas (the cascade). Expressions here can either be primary expressions or other msgSends invoked recursively. On lines 3 to 6 we define the structure of our messages. These can consist of a list of one or more keywords followed by expressions (line 4, keyword-messages). Keyword messages appear first since they are the outer-most (*i.e.,* smallest precedence) messages. Followed (on line 5) by binary messages that consist of a binary operator (like + , - , / etc.) and an expression as a message argument. Finally on line 6 we describe the highest priority unary messages, which simply consist of an identifier.

Both keyword and binary message-rules are simply recursive, meaning that recognition of message-rules from within these alternatives, will only consider the next alternatives in the rule. The design logic here apart from being compact is completely equivalent to what we did before for the expression and dangling-else ambiguities. Moreover, by using MOGs the language designer does not need to statically reason about recursion, precedence and ambiguity, but can incrementally find the correct syntactical definition through exploration. Figure 17 shows us the parsed view of a completely unordered message definition, parsing the following Smalltalk snippet:

```
1  at: index put: aValue
2           dict at: index asNumber put: aValue
```

Where line 1 consists of a method definition for the keyword message #at:put:, and line 2 delegates this #at:put: message to a variable named `dict`, with the unary message `index asNumber` as first argument and `aValue` as second. Figure 17 shows us that the unordered case has four different ways to parse the message-send in line 2 (for all different precedence combinations). By recursively ordering the message rule as we showed in Figure 16, we can arrive at the unambiguous parse of Figure 18.

Let us now contrast the simplicity of the MOG definition in Figure 16 with two CFGs and one PEG Smalltalk grammars in active use. The first one is the CFG appearing in the ANSI standard of the language itself [37], seen here on Figure 19. The ANSI definition starts by defining message-sends (on line 2, through the "basic expression" rule) as constructs starting from a primary value (the receiver) followed by messages and zero or more cascades. Notice here that both the receiver and the messages (plural) are defined in a way that aims to avoid recursion. Receivers for *e.g.,* are not only primary values in the general

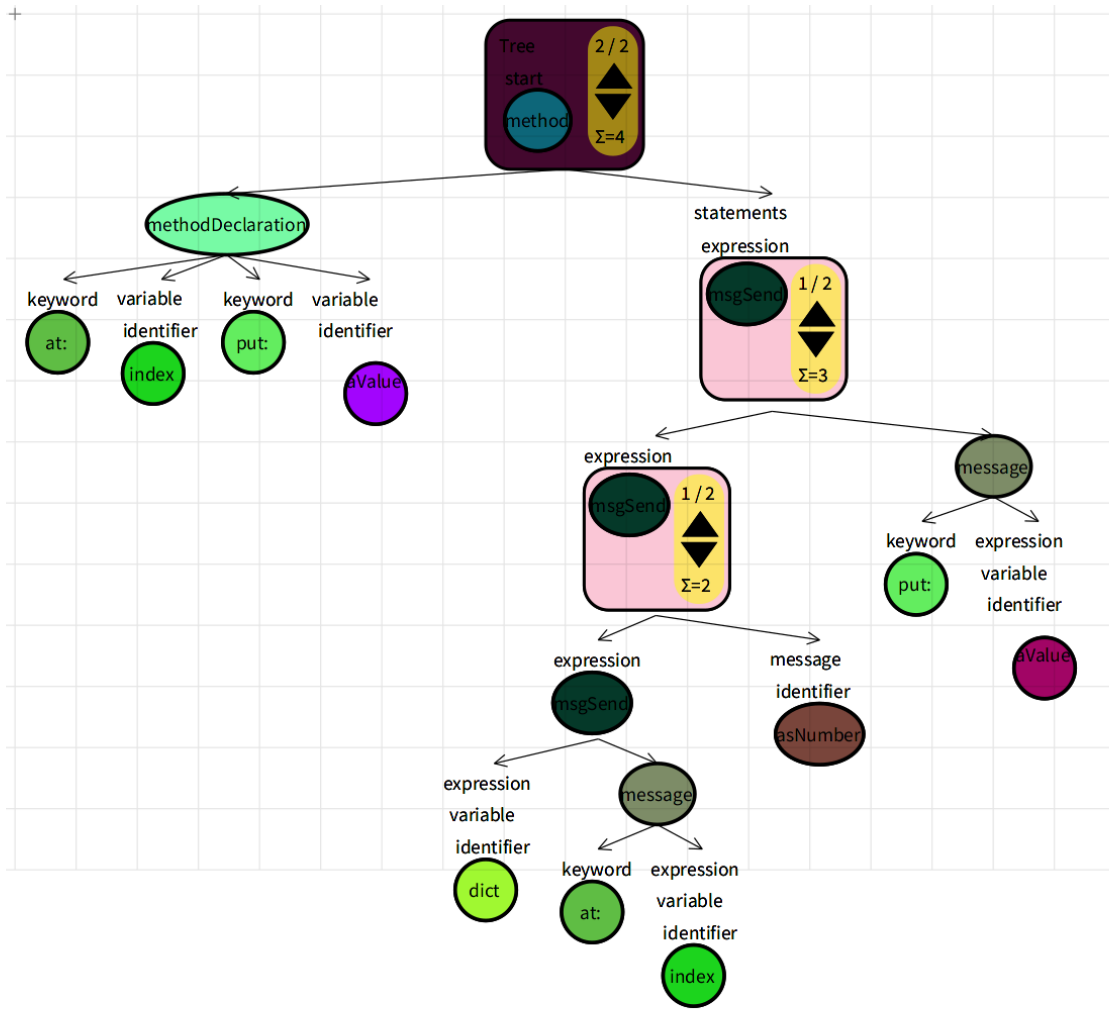


Figure 17. Parsed View: The MOG Smalltalk Grammar (1/2)

case (they can also be message-send themselves). But here the grammar is trying to define the left-most part of a series of expanded messages (lines 4 to 13). Lines 4 and 5 hard-code unary messages that follow each other and act as subsequent receivers for binary messages, themselves optionally followed by a single keyword message (in that order). Then on lines 8 to 13 unary, binary and keyword messages are defined as distinct entities (from each other) and from unary, binary and keyword arguments, again for the sole purpose of hard-coding precedence. Binary arguments can only consist of primaries with optionally zero or more unary msgs as arguments (line 10) and keyword arguments can only consist of primaries and optional unary or binary messages in exactly that order (line 11). This obfuscated expansion is indeed here mandatory, since CFGs have no easy way to define precedence in general, let alone precedence of complicated mutually recursive rules.

But even when external directives are available, like in the case of the SmaCC framework [10] for CFGs (seen in Figure 20), this hard-coded precedence is unavoidable. The reason is that there are no anchoring terminals in the message definition (like arithmetic operators or the ”then”, ”else” keywords that we saw earlier) with respect to which the directives can define ordering. The result is again a one-by-one expansion of rules in order to define precedence, that severely obfuscates the grammar, seen in Figure 20. Again, each type of message-send, message and argument (unary, binary and keyword), has each own distinct rule (9 in total + 3 for cascades) in order to avoid recursion. Keyword message sends (lines 12 to 14) hard-code their receivers (only primary, unary and binary receivers are allowed) and so do their arguments (line 17 to 19). Similary for binary message sends (lines 20 to 22) and arguments (24 to 25), which accept only their primary and unary counter-parts. Finally because of this decomposition, cascades

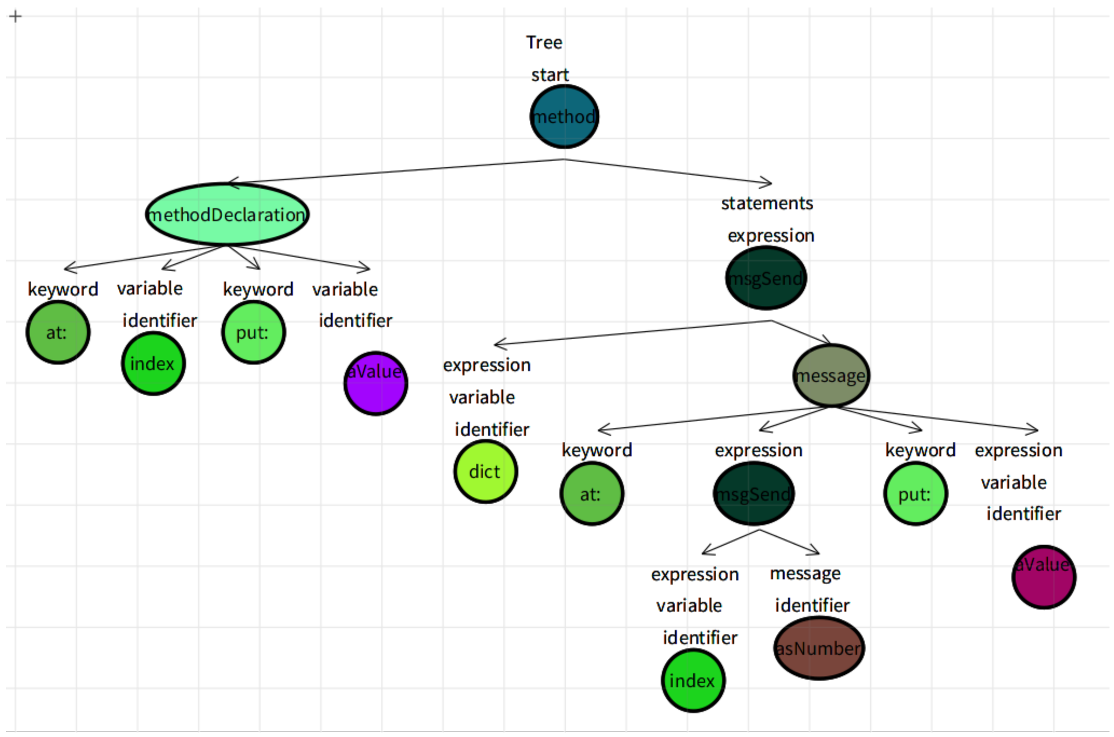


Figure 18. Parsed View: The MOG Smalltalk Grammar (2/2)

Figure 19: CFG Ansi/Smalltalk Msg-sends

```
...
<basic expression> ::= <primary>
         [<messages> <cascaded messages>]
<messages> ::= (<unary message>+ <binary message>*
                  [<keyword message>] )
         | (<binary message>+ [<keyword message>] ) |
         | <keyword message>
<unary message> ::= unarySelector
<binary message> ::= binarySelector <binary argument>
<binary argument> ::= <primary> <unary message>*
<keyword message> ::= (keyword <keyword argument> )+
<keyword argument> ::= <primary> <unary message>*
                        <binary message>*
<cascaded messages> ::= (';' <messages>)*
```

(lines 2 to 5) need to be defined through a separate SimpleMessage rule (lines 9 to 11) that re-states the fact that keyword messages accept only keyword arguments (lines 15 to 16) or that binary messages receive only binary arguments (line 23).

Unsuprisingly, PEG versions (given their strict ordering semantics) handle this case in a very similar way as their CFG counter-parts. Here is how Smalltalk messages are defined in the PEG-based PetitParser framework [36], seen in Figure 21. Starting at lines 2 to 7 cascades are described by defining them as keyword expressions followed by optional cascade messages. This is at first glance peculiar, since a cascade receiver can be any kind of message-send, yet in closer inspection this is yet another manual expansion that avoids recursion. Keyword expressions themselves are defined as possible binary expressions

Figure 20: CFG (LR) Smalltalk Msg-sends

```
1  ...
2  Cascade: MessageSend CascadeList
3           | Primary;
4  CascadeList
5           | CascadeList ";" SimpleMessage;
6  MessageSend: KeywordMessageSend
7           | BinaryMessageSend
8           | UnaryMessageSend;
9  SimpleMessage: UnaryMessage
10          | BinaryMessage
11          | KeywordMessage;
12 KeywordMessageSend: BinaryMessageSend KeywordMessage
13          | UnaryMessageSend KeywordMessage
14          | Primary KeywordMessage;
15 KeywordMessage: keyword KeywordArgument
16          | KeywordMessage keyword KeywordArgument;
17 KeywordArgument: BinaryMessageSend
18          | UnaryMessageSend
19          | Primary;
20 BinaryMessageSend: BinaryMessageSend BinaryMessage
21          | UnaryMessageSend BinaryMessage
22          | Primary BinaryMessage;
23 BinaryMessage: binarySymbol BinaryArgument;
24 BinaryArgument: UnaryMessageSend
25          | Primary;
26 UnaryMessageSend: UnaryMessageSend UnaryMessage
27          | Primary UnaryMessage;
28 UnaryMessage: name
```

(in a top down fashion) followed by an optional keyword message (defined separately), hard-coding the precedence for the receiver in this case (lines 8 to 9). Keyword messages are then defined as keywords followed by binary expressions, in order to hard-code the precedence for their arguments (lines 10 to 11). Similarly binary expressions are potential unary expressions followed by optional binary messages (lines 12 to 13) and binary messages are explicitely defined to receive only unary expressions (lines 14 to 15). Finally unary expressions are defined as possible primaries, followed by one or more unary messages (lines 16 to 17) as the final step of the hard-coded expansion.

4.3 Extending Smalltalk

To conclude this case study, we present in Figures 22 through 24 (see also Appendix A) a series of extensions for the Smalltalk grammar. The goal here is to show that the flexibility and expressiveness of MOGs is not limited only to language design, but can also prove useful for language evolution. During language evolution the main goal is to extend an existing grammar with entirely new language constructs, but it is preferable to do so in a comprehensible and non-intrusive way.

Given the unfamiliar syntax that new Smalltalk users face in their first contact with the language, we consider here the following extensions that can give any Smalltalk method a more mainstream representation:

Figure 21: PEG Smalltalk Msg-sends

```
...
cascadeExpression
        ^ keywordExpression , cascadeMessage star.
cascadeMessage
        ^ $; asParser smalltalkToken , message.
message
        ^ keywordMessage / binaryMessage / unaryMessage.
keywordExpression
        ^ binaryExpression , keywordMessage optional.
keywordMessage
        ^ (keywordToken , binaryExpression) plus.
binaryExpression
        ^ unaryExpression , binaryMessage star.
binaryMessage
        ^ (binaryToken , unaryExpression).
unaryExpression
        ^ primary , unaryMessage star.
unaryMessage
        ^ unaryToken.
```

Imperative declaration `postcard(x):` Allow keyword and unary declarations to be written in a more familiar imperative style. The above example would be expressed initially as a keyword method: `postcard: x`

Method invocation `a.intersection(b)` Allow message sends to resemble familiar method invocations using the dot operator. Here our example is the dot equivalent of the keyword message-send: `a intersection: b`

Variable init `var y := #[100] + self.bSize() + super.bSize().` Declare and initialize variables in a single statement, using the familiar *var* keyword. Pure Smalltalk forces separation of declaration (at the top of methods and blocks) and their initialization as follows: `|y| ... y :=  #[100] + self bSize + super bSize.`

Bracket indexing `t := t[1::t.size()-1]. d[t] := item.` Allow familiar bracket indexing both for interval `t[from::to]` and single value read/write access `d[index]:= value.` In the initial syntax these are plain messages resulting in the quite verbose statements: `t := t copyFrom: 1 to: t size -1. d at: t put: item.`

Functor invocation `step(x)` Introduce a () operator for functor invocation, treating every object as callable. In the example above *step* can be a lambda but also any other object that responds to the message #value:. In plain Smalltalk we would write: `step value: x`

Brace blocks `{ ... }` Brace blocks are lambdas masqueraded as code-blocks. Depending on context they can receive one or more arguments from their surroundings (see function block and for-statements below). They are normally defined in the initial syntax as: `[:a :b | |temp| ... ].` Braces can be optional, in which case the resulting lambda will consist of a single statement (see for *e.g.,* the while construct below).

Function blocks `f(i) { y := y + i }` Function blocks are a more familiar way to define lambdas, using the $f(x)$ notation, binding the function arguments to the brace block that follows. The example above would be normally expressed as: `[:i | y := y + i ]`

For statements `for item in inter do: { ... }` For statements are parametric collection messages, that receive (a) a list of iteration variables (this would be the variable *item* in our example above) (b) a collection to iterate over (variable *inter* above) (c) a collection operation (*like the #do*: *message*) and (d) a brace-block that receives the initial variable list as input arguments. In the example above, we are iterating over the collection *inter*, passing *item* as a lambda argument to the brace-block for each iteration. In the initial form the loop would be defined as `inter do: [ :item | ... ]`

While statements `while y.first() + x < 255. step(x).` A while statement that receives an expression as a condition and executes a brace-block (can be a single statement like in the example above) while the condition is true. In plain Smalltalk the above would be written as: `[y first + x < 255] whileTrue: [step value: x].`

If statements `if true & false.not() & nil.isNil() { ... }` Similarly if statements consist of a condition followed by a brace-block. In the initial grammar our example would be written as: `true & false not & nil isNil ifTrue: [ ... ].`

Return statements `return x < y.first()` A simple return statement, which in plain Smalltalk would be expressed as: `^ x < y first`

Figure 22 shows us the key MOG rules of the base Smalltalk grammar that have been extended to accommodate our new constructs. Due to the recursive nature of the MOG semantics, we were able to keep the extension points brief and intuitive. Method and variable declarations have been extended in lines 4 and 7 with a simple unordered choice to accommodate the new imperative definitions. The brace block is defined in lines 9 to 12, using the scoped recursive choice (since it introduces a new lexical scope). Lines 11 and 12 define the brace-less block (used in one-liner if/while/for statements). Statements (lines 14 to 18), which previously consisted only of top level expressions (line 18), now include top-level if, while, return and for statements. The rule is ordered to avoid ambiguity between return statements and unary messages of the form: `return var`. Since the brace-block associated with statements is recursively scoped, they can all be mutually nested. The dot method invocation, functor invocation and indexing (lines 24 to 27) are all defined as messages (with scoped recursion used for the invocation parentheses and indexing brackets). Finally function blocks (line 32) are defined as primary values (*i.e.,* equivalent to block closures) with scoped recursion.

The results can be seen in Figures 23 and 24 (of Appendix A). In the upper part of Figure 23 we can see an extended version of the famous Smalltalk postcard [3] (*i.e.,* a method showcasing all syntactic structures of the language [9]). In the lower part of the same figure, we see an alternative mixed-postcard using both the original and the extended syntactic structures that we introduced. Finally in Figure 24, we take an even more radical approach, translating the entire Smalltalk postcard to the new constructs (seen in the upper part of Figure 24), with an annotated version of the same method at the bottom. Especially in this latter case we can see how the extension points that MOGs allowed us to introduce, were able to cover an entire alternative syntax for the language.

5 CONCLUSION

Starting with the realization that neither CFGs nor PEGs are sufficient for a complete description of common cases arising in language design, we propose Multi-Ordered Grammars (MOGs) as an alternative. In order to properly handle ambiguity, recursion, precedence or associativity, current solutions either introduce implementation specific directives or ask users to refactor their grammars to fit the needs of the framework/algorithm/formalism combo. To remedy this situation MOGs (a) allow both deterministic and non-deterministic choices to co-exist, and (b) define a form of recursive and scoped ordering. The formalism is accompanied by a new parsing algorithm (Gray), whose execution semantics we have presented in detail.

Gray first extends chart parsing (normally used for Natural Language Processing) with the empty derivation (ε) to support common EBNF operators (+,*,?,()). Two additional chart operations (backtrack and fork) are then defined to handle ordered backtracking (||) and parsing look-aheads (&, !). Finally, Gray overrides the standard predict and complete procedures of chart-parsing, to accommodate for scoped

[3] http://wiki.c2.com/?SmalltalkSyntaxInaPostcard

Figure 22: Extending Smalltalk using MOGs

```
<methodDeclaration> ::= <identifier>
                      | <binaryOp> <variable>
                      | (<keyword> <variable>) +
                      | <impMethodDeclaration>
...
<temporariesDeclaration> ::= "|" <variable> + "|"
                      | <impVarDeclaration> +
...
<braceBlock> ::= || "{" <temporariesDeclaration> ?
                        <statements> "}"
                 || <dots> ? <statement>
                 || <return>
...
<statement> ::= <classicIf>
                      / <classicWhile>
                      | <classicReturn>
                      | <classicFor>
                      | <expression>
...
<message> ::=
                      \ ( <keyword> <expression> ) +
                      \ <binaryOp> <expression>
                      | <identifier>
                      | <impMethodInvocation>
                      | <functorInvocation>
                      | <impIndexingAssignment>
                      | <impIndexing>
...
<primaryValue> ::= <literal>
                      || <dynArray>
                      || <block>
                      || <functionBlock>
...
```

recursive ordering (/ , \) and the mixing of order with unordered choices. To optimize scanning and memoization, Gray precomputes all first, follow and predict sets to pre-filter unwanted alternatives.

We assessed the expressiveness of Gray and MOGs through two case-studies, where we compared our results to equivalent CFG and PEG solutions. The first case-study analyzed two idealized examples from literature (an expression grammar and a simple procedural language). The second examined a real-world case (the entire Smalltalk grammar and eleven new Smalltalk extensions) probing the complexities of practical needs during language evolution. We showed that in comparison, MOGs were able to reduce complexity and more naturally express language constructs, without resorting to implementation specific directives.

We conclude that combining deterministic and non-deterministic choices in a single grammar specification is not only possible but also beneficial. Moreover, augmented by operators for recursive and scoped ordering the resulting Multi-ordered grammars present a viable alternative to both CFGs and PEGs.

Further research is indeed warranted to bring MOGs into maturity, in tandem with a detailed complexity analysis of the Gray algorithm.

ACKNOWLEDGMENTS

The author would like to thank S. Ducasse, L. Fabresse, M. Denker and S. Stinckwich for their support, as well as N. Laurent and B. Combemale for their feedback.

REFERENCES

[1] Aho, A. V. and Ullman, J. D. (1972). The theory of parsing, translating, and compiling, vol. ii.
[2] Aycock, J. and Horspool, N. (2001). Directly-executable earley parsing. In *International Conference on Compiler Construction*, pages 229–243. Springer.
[3] Aycock, J. and Horspool, R. N. (2002). Practical earley parsing. *The Computer Journal*, 45(6):620–630.
[4] Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*.
[5] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1963). Revised report on the algorithmic language algol 60. *The Computer Journal*, 5(4):349–367.
[6] Béra, C. and Denker, M. (2013). Towards a flexible pharo compiler. In *IWST*.
[7] Bergel, A. (2016). *Agile Visualization*. Lulu. com.
[8] Birman, A. (1970). *The TMG Recognition in Schema.* PhD thesis, Princeton.
[9] Black, A. P., Nierstrasz, O., Ducasse, S., and Pollet, D. (2010). *Pharo by example*. Lulu. com.
[10] Brant, J. and Roberts, D. (2009). The smacc transformation engine: how to convert your entire code base into a different programming language. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 809–810. ACM.
[11] Chiang, Y. and Fu, K.-S. (1984). Parallel parsing algorithms and vlsi implementations for syntactic pattern recognition. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (3):302–314.
[12] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.
[13] Chomsky, N. (1959). On certain formal properties of grammars. *Information and control*, 2(2):137–167.
[14] Chomsky, N. and Lightfoot, D. W. (2002). *Syntactic structures, 1957 and 2002*. Walter de Gruyter.
[15] DeRemer, F. and Pennello, T. (1982). Efficient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):615–649.
[16] DeRemer, F. L. (1969). *Practical translators for LR (k) languages.* PhD thesis, Massachusetts Institute of Technology.
[17] Donnely, C. and Stallman, R. (2015). Gnu bison–the yacc-compatible parser generator.
[18] Dubroy, P. and Warth, A. (2017). Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 14–25. ACM.
[19] Ducasse, S., Lanza, M., and Tichelaar, S. (2000). Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, volume 4.
[20] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.
[21] Earley, J. (1983). An efficient context-free parsing algorithm. *Communications of the ACM*, 26(1):57–61.
[22] Ford, B. (2002a). *Packet parsing: a practical linear-time algorithm with backtracking*. PhD thesis, Massachusetts Institute of Technology.
[23] Ford, B. (2002b). Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM.
[24] Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM.

[25] ISO/IEC 14977:1996 (1996). Information technology – syntactic metalanguage – extended bnf. Standard, International Organization for Standardization, Geneva, CH.

[26] Jurafsky, D. and Martin, J. H. (2000). *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice-Hall.

[27] Kegler, J. (2012). Marpa, a practical general parser: The recognizer.

[28] Kegler, J. (2018). Parsing: A timeline. `https://jeffreykegler.github.io/personal/timeline_v3`. Accessed: 2018-12-06.

[29] Laurent, N. and Mens, K. (2015). Parsing expression grammars made practical. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 167–172. ACM.

[30] Leo, J. M. (1991). A general context-free parsing algorithm running in linear time on every lr (k) grammar without using lookahead. *Theoretical computer science*, 82(1):165–176.

[31] McLean, P. and Horspool, R. N. (1996). A faster earley parser. In *International Conference on Compiler Construction*, pages 281–293. Springer.

[32] Nederhof, M.-J. and Satta, G. (1997). A variant of earley parsing. In *Congress of the Italian Association for Artificial Intelligence*, pages 84–95. Springer.

[33] Nierstrasz, O. and Kurš, J. (2015). Parsing for agile modeling. *Science of computer programming*, 97:150–156.

[34] Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98.

[35] Parr, T., Harwell, S., and Fisher, K. (2014). Adaptive ll (*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*, volume 49, pages 579–598. ACM.

[36] Renggli, L., Ducasse, S., Gîrba, T., and Nierstrasz, O. (2010). Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*.

[37] Shan, Y., Krasner, G., Schuchardt, B., and DeNatale, R. (2007). Ncits j20 draft of ansi smalltalk standard, revision 1.9, december 1997. *Available at wiki. squeak. org/squeak/uploads/172/standard_v1_9-indexed. pdf. Accessed in June*.

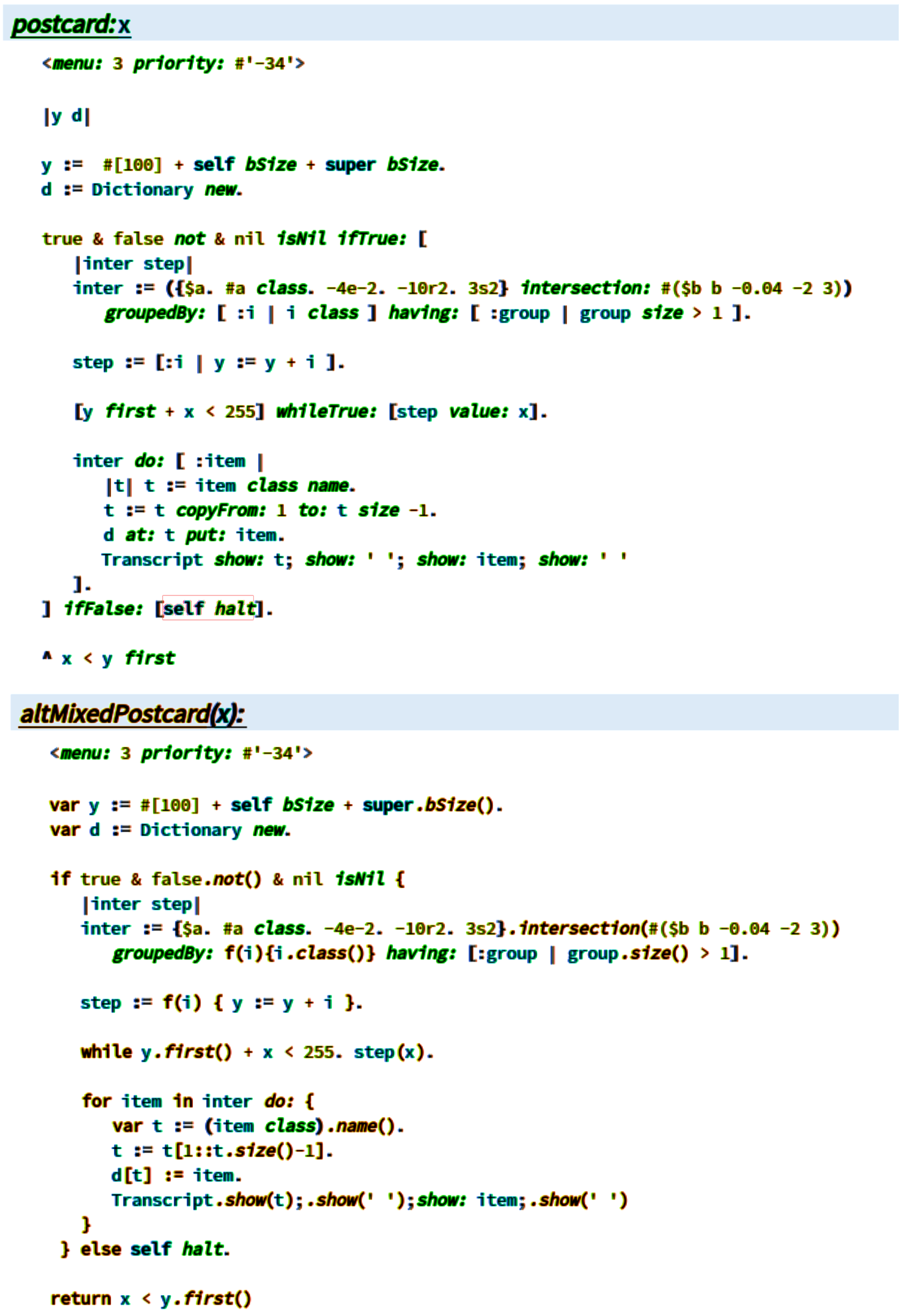
[38] Tratt, L. (2011). Parsing: The solved problem that isn't. `https://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt.html`. Accessed: 2018-12-06.

[39] Warth, A., Douglass, J. R., and Millstein, T. D. (2008). Packrat parsers can support left recursion. *PEPM*, 8:103–110.

[40] Warth, A., Dubroy, P., and Garnock-Jones, T. (2016). Modular semantic actions. In *ACM SIGPLAN Notices*, volume 52, pages 108–119. ACM.

[41] Warth, A. and Piumarta, I. (2007). Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19. ACM.

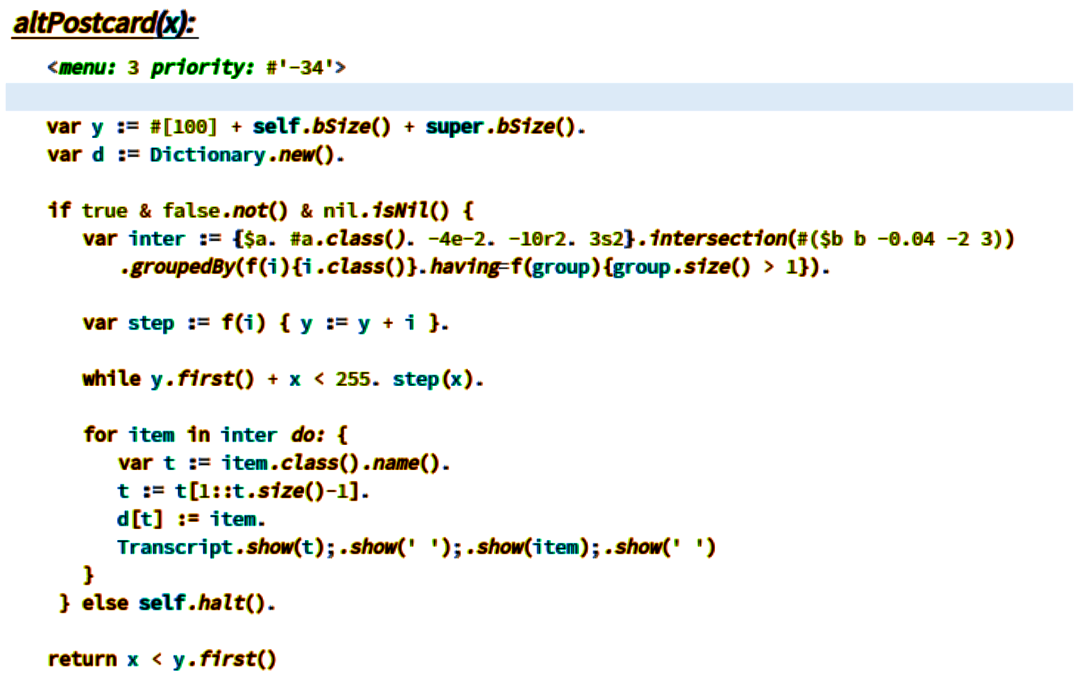
A SMALLTALK POSTCARD EXAMPLES AND EXTENSIONS



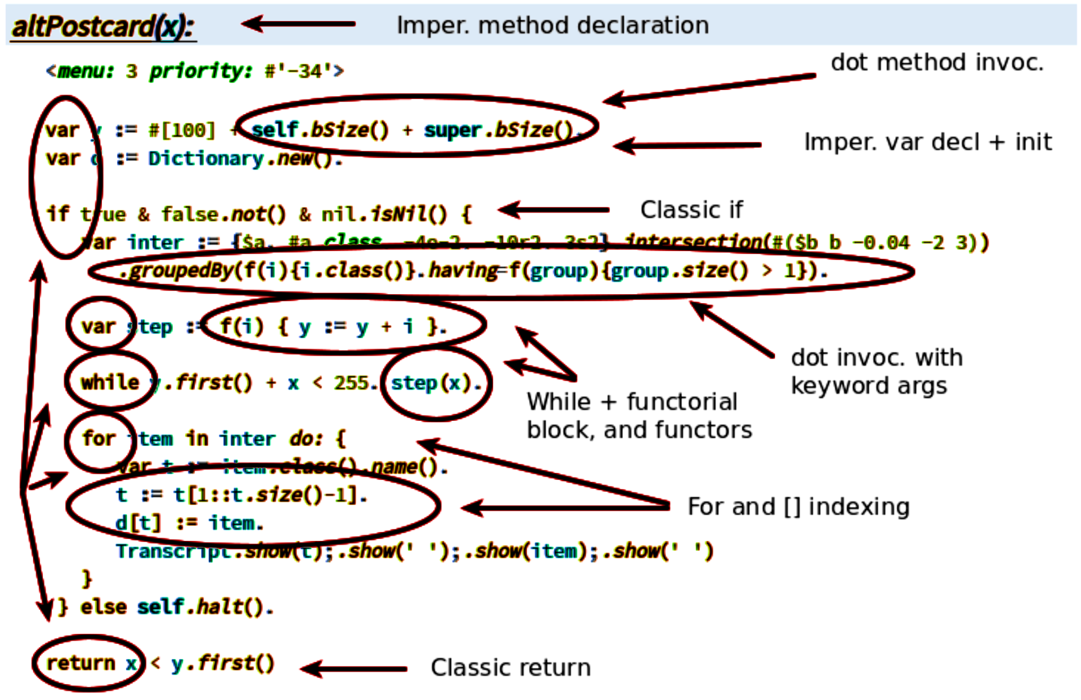
```
postcard:x
    <menu: 3 priority: #'-34'>

    |y d|

    y :=  #[100] + self bSize + super bSize.
    d := Dictionary new.

    true & false not & nil isNil ifTrue: [
       |inter step|
       inter := ({$a. #a class. -4e-2. -10r2. 3s2} intersection: #($b b -0.04 -2 3))
          groupedBy: [ :i | i class ] having: [ :group | group size > 1 ].

       step := [:i | y := y + i ].

       [y first + x < 255] whileTrue: [step value: x].

       inter do: [ :item |
          |t| t := item class name.
          t := t copyFrom: 1 to: t size -1.
          d at: t put: item.
          Transcript show: t; show: ' '; show: item; show: ' '
       ].
    ] ifFalse: [self halt].

    ^ x < y first
```

```
altMixedPostcard(x):
    <menu: 3 priority: #'-34'>

    var y := #[100] + self bSize + super.bSize().
    var d := Dictionary new.

    if true & false.not() & nil isNil {
       |inter step|
       inter := {$a. #a class. -4e-2. -10r2. 3s2}.intersection(#($b b -0.04 -2 3))
          groupedBy: f(i){i.class()} having: [:group | group.size() > 1].

       step := f(i) { y := y + i }.

       while y.first() + x < 255. step(x).

       for item in inter do: {
          var t := (item class).name().
          t := t[1::t.size()-1].
          d[t] := item.
          Transcript.show(t);.show(' ');show: item;.show(' ')
       }
     } else self halt.

    return x < y.first()
```

Figure 23. Parsed View: Original (top) and Mixed Smalltalk Example (bottom) (1/2)



```
altPostcard(x):
   <menu: 3 priority: #'-34'>

   var y := #[100] + self.bSize() + super.bSize().
   var d := Dictionary.new().

   if true & false.not() & nil.isNil() {
      var inter := {$a. #a.class(). -4e-2. -10r2. 3s2}.intersection(#($b b -0.04 -2 3))
         .groupedBy(f(i){i.class()}.having=f(group){group.size() > 1}).

      var step := f(i) { y := y + i }.

      while y.first() + x < 255. step(x).

      for item in inter do: {
         var t := item.class().name().
         t := t[1::t.size()-1].
         d[t] := item.
         Transcript.show(t);.show(' ');.show(item);.show(' ')
      }
    } else self.halt().

   return x < y.first()
```



```
altPostcard(x):
   <menu: 3 priority: #'-34'>

   var y := #[100] + self.bSize() + super.bSize().
   var d := Dictionary.new().

   if true & false.not() & nil.isNil() {
      var inter := {$a. #a.class(). -4e-2. -10r2. 3s2}.intersection(#($b b -0.04 -2 3))
         .groupedBy(f(i){i.class()}.having=f(group){group.size() > 1}).

      var step := f(i) { y := y + i }.

      while y.first() + x < 255. step(x).

      for item in inter do: {
         var t := item.class().name().
         t := t[1::t.size()-1].
         d[t] := item.
         Transcript.show(t);.show(' ');.show(item);.show(' ')
      }
    } else self.halt().

   return x < y.first()
```

Imper. method declaration
dot method invoc.
Imper. var decl + init
Classic if
dot invoc. with keyword args
While + functorial block, and functors
For and [] indexing
Classic return

Figure 24. Parsed View: Extended Smalltalk Example (top) and Annotations (bottom) (2/2)