Direct Left-Recursive Parsing Expressing Grammars

Laurence Tratt
laurie@tratt.net

October 25, 2010

Technical report EIS-10-01
Middlesex University, The Burroughs, London, NW4 4BT, United Kingdom.

Direct Left-Recursive Parsing Expressing Grammars

Laurence Tratt laurie@tratt.net

October 25, 2010

Abstract. Parsing Expression Grammars (PEGs) are specifications of unambiguous recursive-descent style parsers. PEGs incorporate both lexing and parsing phases and have valuable properties, such as being closed under composition. In common with most recursive-descent systems, raw PEGs cannot handle left-recursion; traditional approaches to left-recursion elimination lead to incorrect parses. In this paper, I show how the approach proposed for direct left-recursive Packrat parsing by Warth *et al.* can be adapted for 'pure' PEGs. I then demonstrate that this approach results in incorrect parses for some PEGs, before outlining a restrictive subset of left-recursive PEGs which can safely work with this algorithm. Finally I suggest an alteration to Warth *et al.*'s algorithm that can correctly parse a less restrictive subset of directly recursive PEGs.

1 Introduction

Parsing is the act of discovering the structure of text with respect to a particular grammar. Parsing algorithms such as LL and LR parsing were developed alongside the first generation of high level programming languages. In general, such parsing algorithms aim to parse subsets of the full class of Context Free Grammars (CFGs). By limiting the class of parseable languages, such algorithms are both time and space efficient, considerations that were of huge practical importance given the performance limitations of hardware available at the time. In this classical sense, parsing is generally split into two phases. In the first phase, *tokenization* (or 'lexing') splits the input into distinct tokens (or 'lexemes'), which can be broadly thought of as being similar to splitting English text into words. In the second phase, the token stream is parsed, checking that it conforms to the user's grammar and, optionally, ordering it into a tree structure (the *parse tree*). While both tokenizers and grammars can be written by hand, it is common to use specialised tools to generate them from customised domain specific languages (e.g. the ubiquitous `yacc` tool). The speed of modern computers means that relatively inefficient approaches to parsing are now often practical. For example, Earley's algorithm [1] can parse the entire class of CFGs; while it is $O(n^3)$, even a simple implementation can parse in the low thousands of lines per second [2]. For many people, parsing is a solved problem: there are a wide variety of well understood algorithms, with a reasonable body of knowledge about which are the most suitable in differing circumstances.

Traditional parsing algorithms have one major weakness: it is at best difficult, and generally impossible, to combine two parsers. For example, one may wish to embed SQL

inside Java so that SQL expressions can be used wherever a Java expression is expected; such examples require a parser which understands both constituent languages. Ideally one would like to take existing, separate, Java and SQL parsers and combine them. Many parsing algorithms restrict the class of grammars they can accept (in general to a subset of CFGs); for most such approaches, combining two valid grammars can lead to a grammar which falls outside the class of valid grammars. Equally critically, the separation of the tokenization and parsing phases hinders switching between the different language parsers. Put simply, since tokenizers are largely stateless they cannot know when to stop tokenizing (meaning that the Java tokenizer may raise an error when it comes across SQL input)—attempts to hack around this problem are rarely pleasant and generally fragile.

Visser introduced arguably the first practical scannerless parsing approach [3], which unites the tokenization and parsing phases, making the combination of grammars practical. Visser's approach has no theoretical problem combining grammars, since it can parse the full class of CFGs, and CFGs are closed under composition—however a combined grammar may contain new ambiguities relative to its constituent parts. Since the static detection of ambiguity is, in general, impossible, this can make grammar composition a difficult exercise in practice.

Ford introduced Parsing Expression Grammars (PEGs) [4] which are an alternative approach to scannerless parsing. PEGs have three major differences when compared to other parsing approaches. First, the class of languages PEGs can express has no relation to CFGs. Second, PEGs describe an unambiguous parser; a string is either not parsed by a PEG or has a single unambiguous parse. Third, PEG parsers are closed under union, intersection, and complement. Because of this last point, PEGs are of substantial interest to those looking to combine parsers (see e.g. [5, 6]).

Although PEGs can perform limited context sensitive parsing (through semantic predicates, which are effectively side-conditions on parsing rules), in practice they resemble a combination of simple regular expressions and recursive-descent parsing. Because of this latter point, PEGs cannot handle left-recursion, which is when a rule refers to itself before any input has been consumed. For example – assuming a suitable tokenizer – a typical CFG grammar for a subset of mathematical expressions might be expressed as follows:

```
Expr ::= Expr "+" Prim
       | Expr "-" Prim
       | Prim
Prim ::= "(" Expr ")"
       | INT
```

The directly translated PEG parser looks as follows:

```
Expr <- Expr "+" Prim
      / Expr "-" Prim
      / Prim
Prim <- "(" Expr ")"
      / Int
Int  <- [0-9]+
```

Unfortunately, as with recursive descent parsers in general, the left-recursion in the `Expr` rule means that a PEG parser will enter an infinite loop and never terminate. Since many standard grammar constructs are most naturally represented with left-recursion, this limitation is often frustrating in practice. As I show in Section 3, traditional left-recursion techniques cannot provide the expected result with PEGs. Thus, while grammars can be manually rewritten to avoid left-recursion, it is difficult to verify that the language accepted by the rewritten grammar is identical to the original.

`"s"`	Match string literal `s`
`[c]`	Match against character class `c`
`.`	Match any character
`(e)`	Grouping
`e?`	Match `e` zero or one times
`e*`	Match `e` zero or more times
`e+`	Match `e` one or more times
`&e`	Non-consuming match of `e`
`!e`	Non-consuming negative match of `e`
e_1 e_2	Sequence
e_1 `/` e_2	Ordered choice

Figure 1: PEG operators.

Warth *et al.* recently proposed a modification to handle left-recursion in Packrat parsers [7]. Packrat parsers are essentially an optimisation of PEGs, utilising memoisation to make Packrat parsers O(n) [8]; Warth *et al.*'s approach relies on Packrat memoisation. Warth *et al.* first present a relatively simple adaption of Packrat parsing to cope with direct left-recursion, before detailing a much more complicated adaption to cope with indirect left-recursion. As will become apparent as this paper continues, direct recursion raises sufficiently many issues that I concentrate solely on it. It should be noted that while Packrat parsing obviously adds an extra layer of complexity over 'pure' PEGs, contrary to expectations it often slows parsing down when used blindly for every rule [9].

The first research challenge tackled in this paper is: can Warth *et al.*'s approach be adapted to a 'pure' PEG parser without Packrat memoisation (Section 4.2)? Having then identified that certain PEG grammars then lead to incorrect parses (Section 5) the second research question tackled is: can a well-defined subset of PEGs be parsed correctly with Warth *et al.*'s algorithm (Section 6.1)? The final research question tackled is: can Warth *et al.*'s algorithm be modified so that a wider set of directly recursive PEGs be parsed correctly (Section 6.3)?

2 PEG overview

A full overview of PEGs can be found in Ford's original paper [4]. This section is a brief refresher.

Figure 1 shows the PEG operators. The crucial difference between PEGs and CFGs is the *ordered choice* operator e_1 `/` e_2. This means 'try e_1 first; if it succeeds, the ordered choice immediately succeeds and completes. Only if e_1 fails should e_2 be tried.' In other words, once an earlier part of an ordered choice matches against a piece of text, the ordered choice has committed to that piece of text, and latter parts of the ordered choice are never tried. Therefore an ordered choice `a / ab` will not match against the complete string 'ab' as the ordered choice immediately matches `a`, succeeds, and does not try the second part of the ordered choice. This in contrast to CFGs where both e_1 and e_2 are on an equal footing.

The `&e` and `!e` operators are semantic predicates which succeed if e matches or does not match, respectively; note that they do not advance the input position. Semantic predicates are the other significant difference between PEGs and CFGs.

Most other aspects of PEGs are similar to CFGs. A PEG grammar consists of one or more rules; rules are referenced by their name. In keeping with Ford's original

definition, a PEG rule R with a body expression e is written R `<-` e.

3 Traditional left-recursion removal

Put simply, left-recursive PEGs are those where a rule calls itself without advancing the input position being parsed; in other words, left-recursive PEGs cause infinite loops. Consider this example from Warth *et al.*:

```
expr <- expr "-" num / num
num  <- [0-9]+
```

When this parses against an input, the first action of the `expr` rule is to call itself; once called, the `expr` rule then calls itself again recursively. Since no input is consumed before `expr` calls itself, no progress is ever made (indeed, the general result of running such a PEG is a 'stack exhausted' error). As `expr` calls itself directly, this example shows a *direct* left-recursive rule. *Indirect* left-recursion is when a rule R calls a rule R' which then calls R (where R and R' are distinct rules). Indirect left-recursion adds a number of challenges over direct left-recursion which I do not consider further in this paper.

The problem of left-recursion is a long-standing issue in LL and recursive-descent parsing. In this section I show a simplified version of the standard technique for left-recursion elimination from parsing theory (see e.g. [10] for more details), and demonstrate why it is not suitable for PEGs. In essence, a left-recursive rule:

$$\begin{aligned} R &\rightarrow R\ a \\ &\mid b \end{aligned}$$

can be translated to the right-recursive:

$$\begin{aligned} R &\rightarrow b\ R' \\ R' &\rightarrow a\ R' \\ &\mid \varepsilon \end{aligned}$$

The language accepted by the latter non-left-recursive CFG is provably the same as the former left-recursive CFG. It therefore seems sensible to apply this approach to left-recursive PEG, rewriting the example from above as follows:

```
expr  <- num expr2
expr2 <- "-" num expr2
       /
num   <- [0-9]+
```

If we use this new PEG to parse the input `1-2-3` and write a standard tree walker over it to evaluate the expression we get the answer 2, instead of the expected -4. Where we expect `1-2-3` to parse as equivalent to `(((1)-2)-3)`, it has in fact been parsed as the equivalent of `(1-(2-(3)))`; the left-recursion elimination has turned the left-associative left-recursive PEG into a right-associative right-recursive PEG.

Attempting to restore left-associativity is difficult and, in the presence of semantic actions or predicates, impossible. Grimm's *Rats!* [5] uses traditional left-recursion elimination but 'promises' to transparently transform the resulting right-associative parse tree into the equivalent left-associative structure. This technique is only applicable to those directly left-recursive rules which do not call external code (e.g. no semantic predicates), as the order that the text is parsed is still incorrect; thus, actions cannot appear in such rules, as they could unwittingly be victims of the incorrect parse order.

In summary, naïve traditional left-recursion destroys left-associativity. While in certain limited situations left-associativity can be restored *ex post facto*, this is not a general solution.

4 Warth et al.'s proposal

Warth *et al.* propose a solution to both direct and indirect left-recursion in PEGs. Their approach is based on the concept of 'growing the seed'. In essence, when direct left-recursion on a rule R at input position P is detected a default *seed* value (initially set to a 'fail' value) is immediately returned for subsequent references to R, meaning that infinite recursion is avoided. An unbounded loop is then started; whenever the body of R matches input successfully, the seed is updated (or 'grown'), and R is reevaluated at input position P. When no further input is matched, or if less input than the previous iteration is matched, the loop is finished. The seed value (which may not have been grown beyond 'fail') is then returned. In essence, the parser turns from (recursive) top-down in normal operation to (iterative) bottom-up when left-recursion is detected.

In this section I first show that Warth *et al.*'s approach is not limited to Packrat parsers and can be naturally applied to pure PEGs (Section 4.2). I also provide a much more detailed explanation of the parser than Warth et. al.

4.1 Conventions

Sets are mutable datatypes (used conventionally otherwise), written as $\{E_0, ..., E_n\}$. An element E can be added to a set S with ADD*(S, E)* and removed with DEL*(S, E)*.

Maps are data types relating keys to elements. A map is written $<K_0{:}E_0, ..., K_n{:}E_n>$ where K represents a key and E an element. In this paper, keys are either integers or rule names; elements can be of an arbitrary type. The existence of a key K within a map M can be checked with $K \in M$. The corresponding element E for a key K in a map M can be obtained with $M[K]$; looking up a non-existent key is a fatal error.

4.2 A pure PEG version of Warth et al.'s proposal

While Warth *et al.*'s approach is given in terms of Packrat parsing, there is nothing about the approach which makes it fundamentally incompatible with a pure PEG approach—Packrat memoisation is used to detect left-recursion and (in an adaption over the original Packrat definition) to store the growing seed. In this section I show that by using a simple map to record left-recursive seed growth, one can provide a pure PEG analogue of the Packrat approach, without the (conceptual and storage) overhead of total memoisation.

Algorithm 1 is written in the style of Warth *et al.*'s direct left-recursion Packrat algorithm and shows my adaption of their algorithm for 'pure' PEGs. While Warth *et al.*'s algorithm is presented as an APPLY-RULE function which evaluates a rule R at position P, the pure PEG APPLY-RULE takes two extra arguments R_{orig} and P_{orig}; these are used so that the 'calling' rule can transmit to APPLY-RULE its identity (R_{orig}) and the input position in effect when R_{orig} was called (P_{orig} where $P_{orig} \leq P$). APPLY-RULE returns a result object on a successful parse; unsuccessful parses return *null*. The precise details of a result are left largely abstract; I assume only that it has a *pos* attribute which details its finishing input position. Using this algorithm, we can now use the left-recursive PEG from Section 3 to parse the input `1-2-3`, receiving a parse tree equivalent to the input `(((1)-2)-3)`. Happily, Algorithm 1 not only terminates (left-recursion does not cause infinite loops), but, for this example, correctly retains the expected left-associativity of the original PEG (a standard tree walker over this parse tree will return the answer -4).

While Algorithm 1 can initially appear rather confusing, in reality it is relatively simple. First of all, we can assume that all aspects of PEG parsing other than rule calling are exactly as they would be in 'normal' PEG evaluation. Where Warth *et*

Algorithm 1 Pure PEG adaption of Warth *et al.*'s algorithm

1: $growing \leftarrow <\overline{R} : <>>$
2: **function** APPLY-RULE(R, P, R_{orig}, P_{orig})
3: **if** $R = R_{orig} \wedge P \in growing[R]$ **then**
4: **return** $growing[R][P]$
5: **else if** $R = R_{orig} \wedge P = P_{orig}$ **then**
6: $growing[R][P] \leftarrow null$
7: **while** true **do**
8: $result \leftarrow$ APPLY-RULE(R, P, R_{orig}, P_{orig})
9: $seed \leftarrow growing[R][P]$
10: **if** $result = null \vee (seed \neq \text{null} \wedge result.pos < seed.pos)$ **then**
11: remove P from $growing[R]$
12: **return** $seed$
13: **end if**
14: $growing[R][P] \leftarrow result$
15: **end while**
16: **else**
17: traditional PEG rule application
18: **end if**
19: **end function**

al.'s proposal uses memoisation to detect that left-recursion is occurring, Algorithm 1 instead uses a map:

> *growing* is a map $<R \rightarrow <P \rightarrow seed>>$ from rules to maps of input positions to seeds at that input position. This is used to record the ongoing growth of a seed for a rule R at input position P.

growing is the data structure at the heart of the algorithm. A programming language-like type for it would be `Map<Rule,Map<Int,Result>>`. Since we statically know all the rules for a PEG, *growing* is statically initialised with an empty map for each rule at the beginning of the algorithm (line 1).

Algorithm 1's simplest case is when R_{orig} is not left-recursively calling itself, in which case rule calling happens as normal (line 17). However, if the call of R is left-recursive then there are two cases. It is most natural to consider the second case first, which is when left-recursion has not yet occurred but is about to begin. This is triggered when no input has been consumed, detected when P has not advanced over P_{orig} (line 5). We first set the seed in the *growing* map at input position P for rule R to *null* (line 6), meaning that if / when left-recursion happens for this rule at this input position, the initial left-recursion will fail. For a rule like `expr <- expr "-" num / num`, this means that, on the first recursive call, the first part of the ordered choice will fail (due to line 4), forcing the ordered choice to then try its second part.

The while loop starting at line 7 is where the algorithm changes from a standard recursive top-down PEG parser to a Warth *et al.* iterative bottom-up parser. In essence, we continually re-evaluate the rule R at input position P (note that P does not advance); each time this re-evaluation is successful, we update the seed in $growing[R][P]$ (line 14). As expected, this means that each update of the seed includes within it the previous seed.

Re-evaluation of R at input position P can be unsuccessful for two reasons: if the re-evaluation fails completely; or if the result returned by re-evaluation consumes less of the input than the current seed (if one exists). The former case is trivial (though

note that, by definition, it can only trigger on the first attempt at left-recursion). The latter is less obvious, and is not explained in depth by Warth *et al.* Intuitively, if a left-recursive call returns a 'shorter' result than the previous known one, then by definition it has not used the current seed; in other words, the left-recursion must have exhausted itself. When re-evaluation is unsuccessful, the seed is returned (line 12). Note that the seed will be *null* if the left-recursive call failed to match any input.

The final unexplained part of Algorithm 1 is when left-recursion is occurring and R calls itself. In this case, the seed is immediately returned (line 4). Note again that this will be set to *null* (due to line 6) on the first left-recursive call; thus line 4 is the key point in preventing an infinite loop in the presence of left-recursion.

4.3 Practical considerations

Algorithm 1 is tailored for simplicity of presentation and analysis, not for performance. In particular, it imposes unnecessary checks for all rules. A practical implementation can trivially reduce this cost by statically determining which rules are potentially directly left-recursive, and only calling the APPLY-RULE in Algorithm 1 for those rules; normal rules can call a 'traditional' PEG function which does not impose such costs. Thus, non-left-recursive calls can trivially be relieved of any extra processing over-head related to left-recursion handling. Furthermore, many approaches will (for efficiency reasons) fold APPLY-RULE into a larger function. In such cases, the third and fourth arguments – P_{orig} and R_{orig} – are also trivially removed. The example implementation that accompanies this paper (see Section 7) demonstrates both aspects.

5 Incorrect parses with Algorithm 1

Unfortunately Algorithm 1 has a major, if subtle, flaw. Consider the following minor change to Warth *et al.*'s example from Section 3, where the right hand side of the first ordered choice in `expr` has been changed from `num` to *expr*:

```
expr <- expr "-" expr / num
num  <- [0-9]+
```

When we use this PEG and Algorithm 1 to parse the input `1-2-3` we get a parse tree tree equivalent to the input `(1-(2-(3)))`. As with traditional left-recursion removal, we have obtained a right-associative parse. It may not be immediately obvious that this is an incorrect result: after all, if the above PEG were a CFG, this would be an entirely valid parse. However this parse violates a fundamental aspect of PEGs, which is that rules match text greedily in order to ensure unambiguous parses [4]. In the above right-associative parse, one can clearly see that the right-recursive call of `expr` has matched more text (`2-3`) than the left-recursive call (`1`).

An important question is whether this error is a result of the move from Packrat parsing to pure PEGs in Algorithm 1. However exactly the same issue can be seen in faithful implementations of Warth *et al.*'s algorithm. Given the following PEG, Warth's OMeta tool [11] produces exactly the same incorrect parse for `1-2-3`:

```
ometa AT <: Parser {
  Expr = Expr:l "-" Expr:r -> (l - r)
       | Num:i -> (i),
  Num = digit+:ds  -> ds.join('')
}
```

A corollary of greedy matching is that, for a given input position P, later expressions in an ordered choice cannot affect the text matched by earlier expressions (i.e. in $\mathtt{e}_0/\mathtt{e}_1$, $\mathtt{e}_1$ cannot effect $\mathtt{e}_0$). Algorithm 1 also violates this corollary. Consider the PEG:

```
expr <- expr "-" num / expr "+" num / num
num  <- [0-9]+
```

When used to parse `1+2-3` the resulting parse tree is equivalent to the input `(((1)+2)-3)`. If we now make the second part of the ordered choice in `expr` right-recursive:

```
expr <- expr "-" num / expr "+" expr / num
num  <- [0-9]+
```

then the parse of `1+2-3` creates a parse tree equivalent to the input `(1+(2-(3)))`. In other words, the change to the second expression of the ordered choice has effected the first expression.

As these examples show, by unintentionally altering the rules on greedy matching, Algorithm 1 introduces ambiguity into PEG parsing, thus losing many of the formal guarantees of traditional PEGs. To the best of my knowledge, this is the first time that this problem with Warth *et al.*'s algorithm has been shown.

6 Identifying and mitigating the problem

Let us first consider the simplest solution to the problem in Section 5: disallowing all (direct or indirect) recursive calls in a left-recursive rule other than an initial left-recursive call. For example, `R <- R X` is allowed, but `R <- R X R` and `R <- R X R Y` are disallowed (assuming that `X` and `Y` are input consuming substitutions). Restricting the set of parseable PEGs to this is trivial – and easily done dynamically; static enforcement would require some cunning as we shall soon see – and will clearly solve the problem. However, while it does increase the set of parseable PEGs relative to the traditional approach, it is still rather restrictive.

In this section, I first show that right-recursion in left-recursive rules is the fundamental problem, allowing a wider class of PEGs to be parsed; by forbidding these, Algorithm 1 can be safely used (Section 6.1). I then show that all definitely direct left-recursive rules without potential right-recursion can be safely parsed too (Section 6.3).

6.1 Pinpointing right-recursion in left-recursive rules

An intuitive analysis of the examples in Section 5 and Algorithm 1 highlights the underlying problem. While Algorithm 1 contains both a top-down and bottom-up parser, the two are not on an equal footing—the top-down parser gets the first opportunity to run to completion. When the bottom-up parser calls APPLY-RULE on line 8 of Algorithm 1, it allows the top-down parser opportunity to match the maximum possible text without giving the bottom-up parser an opportunity to wrest back control. Thus in any battle between left and right-recursion in a single rule, right-recursion (using the top-down parser) always wins over left-recursion (using the bottom-up parser).

That rules that are both left and right-recursive is the underlying issue can be seen with a minor modification to the running example, placing an arbitrary input consuming 'end marker' after the right-recursive call to `expr` (thus making it merely recursive, not right-recursive):

```
expr <- expr "-" expr "m" / num
num  <- [0-9]+
```

If we now parse `1-2m-3m` (i.e. our running example text with end markers inserted), we get a parse tree equivalent to the input `(((1)-2)-3)`—the end markers have prevented

right-recursion matching an arbitrary amount of text, thus allowing left-recursion to work as expected, giving us a left-associative parse.

It thus seems that we can now state a fairly liberal, but still easily implementable, solution which prevents the problem noted in Section 5 from arising: PEGs which contain rules which are both left and right recursive are not permitted. However there is one further subtlety which we must consider, which is best highlighted by the following modification to the running example, making end markers optional:

```
expr <- expr "-" expr "m"? / num
num  <- [0-9]+
```

If we parse the text `1-2-3` against this PEG, we will once again receive an incorrect right-associative parse. Until now I have implicitly assumed that right-recursive rules are those which 'look obviously' right-recursive. In fact, some rule calls are right-recursive, or merely recursive, depending on the input string and whether they are followed by other expressions which can match against the empty string; the same idea is also true for left-recursion. We can use the standard *nullables* computation from parsing theory [10], which is trivially transferable to PEGs, to statically calculate potentially left or *potentially* right-recursive calls[1].

Therefore a precise, fairly liberal, solution to the problem noted in Section 5 is to statically forbid any potentially left-recursive rule which contains potentially right-recursive calls. Non-left-recursive rules can still safely include potentially (direct or indirect) right-recursive calls.

6.2 The case for right-recursion in left-recursive rules

In Section 6.1 I showed that forbidding potentially right-recursive calls in potentially left-recursive rules solved the problem of Section 5. For many uses, this may well allow a sufficiently large class of PEGs to be parsed. There are, however, two reasons why we might wish to remove this restriction. First, this sort of (seemingly arbitrary) restriction is arguably one of the reasons why parsing is considered a black art by many otherwise capable people; the classically liberally minded have every right to ask whether we can allow some form of right-recursion in left-recursive rules. Second, grammars with both left and right recursion are the most natural way to write and evolve many grammars; their absence can complicate development.

For example, a standard expression grammar encoding correct precedences for `+`, `-`, `*`, and `-` looks as follows in traditional PEGs:

```
S <- Term ! .
Term <- Fact "+" Term / Fact "-" Term / Fact
Fact <- Int "*" Fact / Int "/" Fact / Int
Int <- [0-9]+
```

Using Warth *et al.*'s algorithm we can rewrite this to be left-recursive as follows:

```
S <- Term ! .
Term <- Term "+" Fact / Term "-" Fact / Fact
Fact <- Fact "*" Int / Fact "/" Int / Int
Int <- [0-9]+
```

Although this rewriting has simply swapped the left and right-recursion, many people find the left-recursive version more natural. However, in both of the above grammars, we are forced to be careful about making sure the recursion terminates consistently: in both the `Term` and `Fact` rules, note that the non-recursive alternative at the end

[1] In some parsing formalisms this is referred to as 'hidden' recursion.

of the rules (`Fact` and `Int` respectively) has to be encoded in each of the preceding alternatives too. This means that some seemingly local alterations to the grammar in fact require much larger chunks to be altered. For example, if we wish to allow bracketed expressions in the above grammar, we not only have to add a new rule, but change three references to it, as shown in the following (where changes are highlighted in italics):

```
S <- Term ! .
Term <- Term "+" Fact / Term "-" Fact / Fact
Fact <- Fact "*" Prim / Fact "/" Prim / Prim
Prim <- "(" Term ")" / Int
Int <- [0-9]+
```

Consider if, instead, we had been able to write the original expression grammar using both left and right-recursion:

```
S <- Term ! .
Term <- Term "+" Term / Term "-" Term / Fact
Fact <- Fact "*" Fact / Fact "/" Fact / Int
Int <- [0-9]+
```

Adding bracketed expressions to this grammar would then require changing only a single reference:

```
S <- Term ! .
Term <- Term "+" Term / Term "-" Term / Fact
Fact <- Fact "*" Fact / Fact "/" Fact / Prim
Prim <- "(" Term ")" / Int
Int <- [0-9]+
```

Though there are other arguments one could advance for justifying why right-recursion in left-recursive rules is useful, I hope that the above example of increased ease of grammar evolution provides sufficient motivation.

6.3 Allowing direct definite right-recursion in left-recursive rules

In order to begin tackling the problem of allowing right-recursion in left-recursive rules, we first have to solve a challenge: can we ensure that the top-down parser gets sufficient opportunity to run when right-recursion is present in left-recursive rules? A simple observation is key: when we are in left-recursion and then encounter right-recursion, the bottom-up parser allows the right-recursion to continue many recursive levels deep, gobbling up text. Instead, we want the right-recursion to go only one recursive level deep, and then to return control to the bottom-up parser.

The solution, at a high level, therefore seems reasonably straight forward. When we are in left-recursion on a rule R and then right-recursively call R, we need to make all further right recursive calls to R fail, until the top-level right-recursion has returned control to the bottom-up parser. Intuitively, for a rule such as `expr <- expr "+" expr / num`, the right-recursive call of `expr` would then match only `num`, since the first part of the ordered choice would fail.

At a low level, we are forced to consider the issue of potential right-recursion. The high-level solution implicitly relies on statically identifying right-recursive calls; however when executed, potential right-recursion, by definition, may sometimes be simply normally-recursive or definitely right-recursive. To make matters worse, we can not dynamically determine whether a potentially right-recursive call is definitely right-recursive or not until its containing rule is completely evaluated. With PEGs this is not very useful since – unlike traditional CFGs, where one can backtrack arbitrarily – PEGs commit themselves to certain parses which backtracking can not undo. The issue can be seen in this extension of `expr`:

```
expr <- expr "-" expr ("-" expr)? / num
num  <- ["0-9"]+
```

Given this PEG, we would hope that `1-2-3` would parse as `(((1)-2)-3)`. In order for this to be the case, we would first need to evaluate the sub-expression `("-" expr)?` to determine whether it matched any of the input; depending on whether it did, or did not, we would then know whether to treat the middle call of `expr` as normal recursion (not needing special treatment) or right recursion (and requiring the special rule mentioned in the high-level solution). Since we can not evaluate `("-" expr)?` before evaluating the middle call of `expr`, this means that the high-level solution outlined above would mean the above PEG is ambiguous, since the potential right-recursion could lead to two different parses.

The problem with the high-level solution and potential right-recursion seems inherent. The only way to avoid it is to ban potential right-recursion in left-recursive rules. For the avoidance of doubt, we can state the following two rules: non-left recursive rules are allowed to have potential right-recursion; but left-recursive rules must either have no right-recursion or definite right-recursion.

There is one last issue to consider: potential left-recursion. As with right-recursion, one can create rules such as `e <- e?  "x" e` where left-recursion occurs, or not, solely depending on the input. One could therefore wait until run-time to discover whether a rule is called left-recursively, and, if the rule was also statically determined to be potentially right-recursive, complain. The algorithm in the following section supports this dynamic approach. The accompanying implementation, on the other hand, recognises that this is unlikely to be the desired behaviour, and statically rejects potentially right-recursive calls in potentially left-recursive rules.

6.4 An algorithm for definite right-recursion in left-recursive rules

Algorithm 2 shows, in the style of Warth *et al.*, how PEGs can cope with both direct left-recursion and direct definite right-recursion. Presenting the algorithm in this fashion does involve some compromises; however, the advantage conferred by continuity outweighs the distaste incurred by ugliness. In particular, there is a need to convey the information about whether a rule call is right-recursive or not. For the sake of simplicity, I assume that the R argument to the APPLY-RULE function has a boolean attribute *drr* (Definitely Right-Recursive) which is set to *true* if that rule is definitely right-recursive.

The first thing to note about Algorithm 2 is that it is really just an extension of Algorithm 1—lines 14–26 inclusive (the bottom-up parser) are unchanged.

The most complex part of Algorithm 2 relates to calling a right-recursive rule (lines 5–13). We maintain a set *limit* which contains the rule names of all current right-recursive calls (note that *limit* is global only to maintain symmetry with *growing*; it could as easily be a fifth argument to APPLY-RULE). Therefore when calling a definitely right-recursive call R, we first need to see whether we're already in right-recursion on R; if so we fail (line 6), in order to ensure that right-recursion never goes more than one level deep. If we're not in right-recursion on a rule, but are in left-recursion (which is implied if P_{orig} is in $growing[R]$, meaning that the bottom-up parser is in operation), then we add R to *limit* and evaluate the rule as normal (lines 8–10). If we're not in right or left-recursion, we evaluate the rule purely as normal (line 12).

The final subtlety in Algorithm 2 relates to when right-recursion is happening in a rule R and we non-right-recursively call R. In this case, we need to ensure that the non-right-recursive call of R evaluates as normal, without limiting recursion to just one level. Fortunately we do not need to be as explicit as this might suggest; it is sufficient

Algorithm 2 An update of Algorithm 1 which correctly deals with direct right-recursion in left-recursive rules

```
1: growing ← <R̄ : <>>
2: limit ← { }
3: function APPLY-RULE(R, P, R_orig, P_orig)
4:     if R = R_orig ∧ R.drr then
5:         if R ∈ limit then
6:             return null
7:         else if P_orig ∈ growing[R] then
8:             ADD(limit, R)
9:             traditional PEG rule application
10:            DEL(limit, R)
11:        else
12:            traditional PEG rule application
13:        end if
14:    else if R = R_orig ∧ P ∈ growing[R] then
15:        return growing[R][P]
16:    else if R = R_orig ∧ P = P_orig then
17:        growing[R][P] ← null
18:        while true do
19:            result ← APPLY-RULE(R, P, R_orig, P_orig)
20:            seed ← growing[R][P]
21:            if result = null ∨ (seed ≠ null ∧ result.pos < seed.pos) then
22:                remove P from growing[R]
23:                return seed
24:            end if
25:            growing[R][P] ← result
26:        end while
27:    else
28:        if R ∈ limit then
29:            DEL(limit, R)
30:            traditional PEG rule application
31:            ADD(limit, R)
32:        else
33:            traditional PEG rule application
34:        end if
35:    end if
36: end function
```

to temporarily remove R from *limit* (if it exists) whenever R is called and it does not fit into the cases listed above (lines 28–34).

Given the following PEG and the string `1-2-3` we get the left-associative parse we originally desired:

```
expr <- expr "-" expr / num
num  <- ["0-9"]+
```

7 Conclusions

This paper first presented a 'pure PEG' adaption of Warth *et al.*'s left-recursive Packrat algorithm. I then showed cases where this algorithm failed, before identifying a safe

subset of left-recursive PEGs which can be used with this algorithm. I then extended the algorithm, allowing left-recursive rules with definite right-recursion to work as expected. As this paper has shown, in order to safely parse right-recursive PEGs, a number of subtle issues need to be considered, and the class of right-recursive PEGs safely parseable is smaller than might originally have been hoped for. While the obvious next step is to extend the solutions presented in this paper to deal with indirect left and indirect right recursion, this may well prove to be quite challenging and to impose further restrictions on valid PEGs. This therefore raises a deeper philosophical question: are PEGs really suited to allowing left-recursion? That question is left to others to ponder.

Example code which implements the algorithms in this paper can be found at `http://tratt.net/laurie/research/publications/files/direct_left_recursive_pegs`.

My thanks to Tony Sloane for verifying the problem noted in Section 5 with the Packrat implementation in his Kiama tool and providing comments on a draft of this paper; to David Hazell, Franco Raimondi, and Chris Hyuck for providing comments on a draft of this paper; and to Alessandro Warth for answering questions. Any errors and infelicities are my own.

References

[1] Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2) (February 1970)

[2] Tratt, L.: Domain specific language implementation via compile-time meta-programming. TOPLAS **30**(6) (2008) 1–40

[3] Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)

[4] Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proc. POPL, ACM (January 2004) 111–122

[5] Grimm, R.: Better extensibility through modular syntax. Volume 14 of Proc. Programming language design and implementation. (June 2006) 38–51

[6] Seaton, C.: A programming language where the syntax and semantics are mutable at runtime. Master's thesis, University of Bristol (May 2007)

[7] Warth, A., Douglass, J., Millstein, T.: Packrat parsers can support left recursion. In: Proc. PEPM, ACM (January 2008) 103–110

[8] Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time. In: International Conference on Functional Programming. (October 2002) 36–47

[9] Redziejowski, R.R.: Some aspects of parsing expression grammar. Fundamenta Informaticae **85**(1-4) (2008) 441–451

[10] Grune, D., Jacobs, C.J.: Parsing techniques. (1998)

[11] Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: Proc. Dynamic Languages Symposium, ACM (2007) 11–19