PEG: left recursion by recursive ascent

Roman R. Redziejowski

Submitted for publication in Fundamenta Informaticae

Abstract

Parsing Expression Grammar (PEG) is extended to handle left recursion, and under specified conditions becomes a correct parser for left-recursive grammars in Backus-Naur Form (BNF).

1 Introduction

The idea of recursive-descent parsing can be traced back to Lucas [9] who suggested it for grammars later known as Backus-Naur Form (BNF). A recursive-descent parser consists of procedures that correspond to syntax rules. The procedures call each other recursively, each being responsible for recognizing input strings defined by its rule.

Procedures corresponding to certain syntax rules must decide which procedure to call next. This is easily solved by a class of languages that have the so-called LL(1) property: the decision can be made by looking at the next input symbol. When this is not possible, one can use brute force: trial-and-error. It means trying the alternatives one after another and backtracking after a failure. Making full search can require exponential time, so a possible option is limited backtracking: never return after a partial success. This approach has been used in actual parsers [4, 10] and is described in [1–3, 8]. It has been eventually formalized by Ford [6] under the name of Parsing Expression Grammar (PEG). PEG parsers can work in linear time by using the technique of "packrat parsing" [5].

The recursive-descent method, including its PEG version, encounters another problem: left-recursion. A procedure corresponding to left-recursive rule calls itself indefinitely on the same input. There is a strong bias towards using this pattern in the definition of programming languages, which prevents the use of PEG for them. Converting left recursion to iteration is possible, but is tedious, error-prone, and obscures the spirit of the grammar. Several ways of extending PEG to handle left recursion have been suggested [12, 15, 16]. They have their roots in the packrat technology. One repeatedly probes the input, gradually increasing the recursion depth. In this way one builds the recursive path by repeatedly starting at the top.

This paper suggests another procedure, one that reconstructs the path by proceeding upwards. It is called "recursive ascent", and may have something in common with that outlined in [7].

We consider a grammar in Backus-Naur Form and start by giving an informal description of our process. Then we show how the process can be implemented by a PEG that, under the usual conditions, is no longer left-recursive.

2 Basic concepts

We consider grammar G that is a version of BNF grammar over alphabet Σ. It is a set of named *expressions* defined by rules of the form $A \rightarrow e$ where e is an expression and A is its name. An expression is one of these:

– $a \in \Sigma$ ("terminal"),
– $e_1 \dots e_n$ with $n > 1$ ("sequence"),
– $e_1 | \dots | e_n$ with $n \geq 1$ ("choice"),

where each of e_i is either an expresion or expression name. The rule states that A is synonymous with the expression e. The set of all expressions is denoted by $\mathbb{E}_G$.

The grammar is a mechanism for generating strings of terminals. Such string is generated by repeated application of these operations:

– Replace expression $e = e_1 \dots e_n$ by sequence of expressions $e_1, \dots, e_n$.
– Replace expression $e = e_1 | \dots | e_n$ by one of the expressions $e_1, \dots, e_n$.

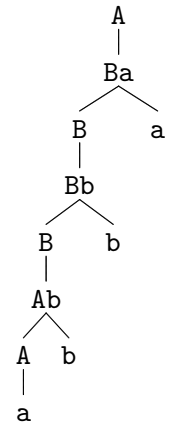
This is carried out until no more replacements are possible. The process is represented by a *syntax tree*. It is an ordered tree where the nodes are expressions and the leaves are terminals. The branch(es) emanating from node e represent replacement of e by one or more expressions.

The set of terminal strings generated from expression e by grammar G is called the *language* of e and is denoted by $\mathcal{L}_G(e)$.

Shown in Figure 1 are examples of two grammars and strings generated by them, together with their syntax trees. (Capital letters are names, small letters are terminals, and `Ab`, `EF`, etc. are temporary names for inner expressions.)

(a)

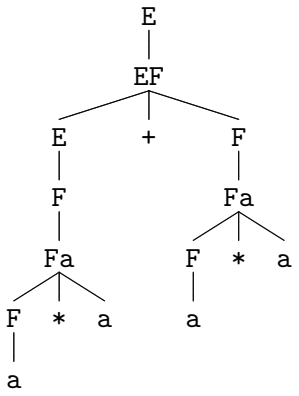
```
A → B a | a
B → A b | B b | b
```



Syntax tree of 'abba'

(b)

```
E → E + F | F
F → F * a | a
```



Syntax tree of 'a*a+a*a'

Figure 1: Examples of left-recursive grammars

For expression $e \in \mathbb{E}_G$ other than a terminal, define the relation $\xrightarrow{\text{firstG}}$ as follows:

- If $e = e_1 \dots e_n$, $e \xrightarrow{\text{firstG}} e_1$.
- If $e = e_1 | \dots | e_n$, $e \xrightarrow{\text{firstG}} e_i$ for $1 \leq i \leq n$.

Let $\xrightarrow{\text{FirstG}}$ be the transitive closure of $\xrightarrow{\text{firstG}}$. Define expression $e \in \mathbb{E}_G$ to be *left-recursive*, (shortly: *recursive*) to mean that $e \xrightarrow{\text{FirstG}} e$. Let $\mathbb{R} \subseteq \mathbb{E}_G$ be the set of all recursive expressions.

Define relation between recursive expressions e_1, e_2 that holds if $e_1 \xrightarrow{\text{FirstG}} e_2 \xrightarrow{\text{FirstG}} e_1$. This is an equivalence relation that partitions $\mathbb{R}$ into equivalence classes. We call them *recursion classes*. The recursion class of expression e is denoted by $\mathbb{C}(e)$.

The grammar of example (a) above has {`A,Ab,B,Ba,Bb`} as the set of recursive expressions. They all belong to the same class. Example (b) has two recursion classes: {`E,E+F`} and {`F,F*a`}.

In syntax tree, the leftmost path emanating from any expression is a chain of expressions connected by $\xrightarrow{\text{firstG}}$. Suppose expression e_1 and e_2 belonging to the same recursion class $\mathbb{C}$ appear on the same leftmost path. Any expression e between them must also belong to $\mathbb{C}$, which follows from the fact that $e_1 \xrightarrow{\text{FirstG}} e \xrightarrow{\text{FirstG}} e_2 \xrightarrow{\text{FirstG}} e_1$. It means that all expressions from $\mathbb{C}$ appearing on the same leftmost path must form an uninterrupted sequence. We call such sequence a *recursion path* of class $\mathbb{C}$.

In example (a), there is one such path: the whole leftmost path without final `a`. In example (b), we have recursion path (`E,E+F,E`) of the class containing `E` and two paths (`F,E*F,F`) of the class containing `F`.

Let $e = e_1 \dots e_n$ be on a recursion path. The next expression on the path must be e_1, and it must belong to $\mathbb{C}(e)$, because $e \xrightarrow{\text{FirstG}} e$ requires $e \xrightarrow{\text{firstG}} e_1 \xrightarrow{\text{FirstG}} e$. It follows that the last expression on a recursion path must be $e = e_1 | \dots | e_n$ where at least one of e_i is not a member of $\mathbb{C}(e)$.

The choice expression with alternatives outside its class $\mathbb{C}$ is called an *exit* of $\mathbb{C}$, and these alternatives are the *seeds* of $\mathbb{C}$. In example (a), both `A` and `B` are exits, and the seeds are `a` and `b`. In example (b), `E` is the exit of class {`E,E+F`} with seed `F`, and `F` is the exit of class {`F,F*a`} with seed `a`.

An expression that may appear as first in recursion path of class $\mathbb{C}$ is an *entry* of that class. The class in example (a) has entry `A`; the classes in example (b) have `E` respectively `F` as their entries.

3 Recursive ascent

Parsing is the process of reconstructing the syntax tree for a given string. In recursive-descent parsing, each expression has a procedure designed to consume the string generated by that expression. The procedure for a terminal just consumes that terminal. The procedure for $e_1 \dots e_n$ calls in turn the procedures for $e_1, \dots, e_n$. The procedure for $e_1 | \dots | e_n$ selects in some way one of the procedures for $e_1, \dots, e_n$ and calls it.

The parser explores the syntax tree top-down. The activated procedures correspond to nodes of the syntax tree, and the calls made by them to the branches. Procedures that explore the leftmost path are all applied to the same input. An attempt to explore a recursion path would mean procedures calling themselves on the same input in an infinite descent.

The avoid this, we suggest that procedure for $e \in \mathbb{R}$ encountered in recursive descent does not continue the descent. Instead, it reconstructs the syntax tree bottom-up, by a process that we call "recursive ascent". The recursion path starting with $e \in \mathbb{C}$ must end with a seed of $\mathbb{C}$. The procedure for e identifies the seed and then grows the syntax tree like a plant towards e.

The process is illustrated on grammar (a) from Figure 1, for input `'abba'`. It is the action of procedure for `A` and its goal is to reconstruct syntax tree with root `A`.

1. The seeds of recursion class of `A` are `a` and `b`. We decide to try `a`. Apply procedure `a` to `'abba'`. It consumes `'a'`, leaving `'bba'`. Put `a` as start of the plant.
2. The only possible predecessor of `a` in the syntax tree is `A`. As `a` is an alternative of `A`, there is no need to call the procedure `A`. It must consume `'a'`. Add `A` on top of the plant. We have now a syntax tree with root `A`, but decide to continue.
3. The only possible predecessor of `A` is `Ab`. We already know that procedure `A` consumed `'a'`. Apply procedure `b` to the remaining `'bba'`. It consumes `b`, so `Ab` consumed `'ab'`. Add `Ab` on top of the plant.
4. The only possible predecessor of `Ab` is `B`. As `Ab` is an alternative of `B`, there is no need to call the procedure `B` . It must consume `'ab'`. Add `B` on top of the plant.
5. The possible predecessors of `B` are `Ba` and `Bb`. Decide to try `Bb`. We already know that `B` consumed `'ab'`, leaving `'ba'`. Apply procedure `b` to `'ba'`. It consumes `b`, so `Bb` consumes `'abb'`. Add `Bb` on top of the plant.
6. The only possible predecessor of `Bb` is `B`. As `Bb` is an alternative of `B`, there is no need to call the procedure `B`. It must consume `'abb'`. Add `B` on top of the plant.
7. The possible predecessors of `B` are `Ba` and `Bb`. Decide to try `Ba`. We already know that `B` consumes `'abb'`, leaving `'a'`. Apply procedure `a` to `'a'`. It consumes `a`, so `Ba` consumes `'abba'`. Add `Ba` on top of the plant.
8. The only possible predecessor of `Ba` is `A`. As `Ba` is an alternative of `A`, there is no need to call the procedure `A`. It must consume `'abba'`. Add `A` on top of the plant. We have a syntax tree with root `A` and decide to stop.

(We made here five successful choices; the way of making them will be discussed later on.)

4 The dual grammar

The process illustrated above can be represented as recursive descent for another grammar.

In the repetitive action of the ascent we first complete processing of an expression R that was initiated by the preceding step, and add R to the plant. Then we select a possible predecessor P of R and repeat the same for P. Apart from growing the plant, this is exactly the recursive descent process for expression

$$\$R \to \texttt{rest}(R)\texttt{pre}(R). \tag{1}$$

Here $\texttt{rest}(R)$ consists of procedures to be called after the first one of R succeeded:

- $\texttt{rest}(e_1 \dots e_n) = e_2 \dots e_n$;
- $\texttt{rest}(e_1| \dots |e_n) = \varepsilon$.

The $\texttt{pre}(R)$ represents the choice of next procedure to call:

- $\texttt{pre}(R) = \$P_1| \dots |\$P_n$, or
- $\texttt{pre}(R) = \$P_1| \dots |\$P_n|\,\varepsilon$ if R is the entry of $\mathbb{C}(R)$,

where P_i are all members of $\mathbb{C}(R)$ such that $P_i \xrightarrow{\text{firstG}} R$.

We introduced above "ε" for procedure that always succeeds without consuming anything. To simplify the discussion, we assume that each recursion class has only one entry expression.

The procedure corresponding to (1) is referred to as *ascent procedure*. The whole process is started by identifying the seed and exploring its predecessor. This is exactly the action of procedure for expression

$$E \to S_1\,\texttt{pre}(S_1)| \dots |S_n\,\texttt{pre}(S_n) \tag{2}$$

where the S_i are all the seeds of $\mathbb{C}(E)$.

The choice expressions in (1) and (2) describe the five decisions that were made in our example. The choice between seeds `a` and `b` in step 1 implements the choice expression in (2). The decision whether

to continue with `Ba` or `Bb` in steps 5 and 7 implements the choice of $\$P_i$ in (1). The decision whether to end the process after arriving at `A` in steps 2 and 8 implements the choice of ε in (1).

All five decisions in our example could be made by looking at the symbol ahead (including "end of input"), but we are going to use limited backtracking for the general case. Before proceeding to that, we need to check some facts.

We are going to consider the "dual grammar" D obtained by transforming the grammar G as follows:

- Assign names to all unnamed inner expressions.
- For each $A \in \mathbb{R}$ replace $A \to e$ by $\$A \to \texttt{rest}(A)\texttt{pre}(A)$.
- For each entry E add $E \to S_1\,\texttt{pre}(S_1)|\ldots|S_n\,\texttt{pre}(S_n)$.

The dual grammars for our examples are shown in Figure 2.

(a)

```
A   → a $A | b $B
$A  → ε ($Ba | ε)
$Ba → b $B
$B  → ε ($Ab | $Bb)
$Ab → a $A
$Bb → b $B
```

(b)

```
E   → F $E
$E  → ε ($EF | ε)
$EF → + F $E
F   → a $F
$F  → ε ($Fa | ε)
$Fa → * a $F
```

Figure 2: Dual grammars for examples from Figure 1.

The set of expressions of D is in the following denoted by $\mathbb{E}_D$. It consists of \$-expressions and expressions that are "the same" as in $\mathbb{E}_G$. The set of these "common" expressions is in the following denoted by $\mathbb{E}_C$. It consists of the set $\overline{\mathbb{R}}$ of non-recursive expressions of G and the set $\mathbb{A}$ of entry expressions. The expressions in $\mathbb{A}$ have the same name in the two grammars; the expressions in $\overline{\mathbb{R}}$ are the same, but refer to different entry expressions using the same name. The meaning should hopefully be clear from the context.

5 Left recursion in dual grammar

The dual grammars in Figure 2 are obviously not left-recursive. But this is not necessarily the case. For example, the dual grammar for `A → A|a` is:

```
A  → a $A
$A → $A | ε
```

which is left-recursive.

To investigate left recursion in the dual grammar, we look at the relation $\xrightarrow{\text{firstD}}$:

(a) For $e = e_1 \ldots e_n \in \overline{\mathbb{R}}$, $e \xrightarrow{\text{firstD}} e_1$.
(b) For $e = e_1|\ldots|e_n \in \overline{\mathbb{R}}$, $e \xrightarrow{\text{firstD}} e_i$ for $1 \le i \le n$.
(c) For $e \in \mathbb{A}$, $e \xrightarrow{\text{firstD}} S$ where S is a seed of $\mathbb{C}(e)$.
(d) For $R = e_1 \ldots e_n \in \mathbb{R}$, $\$R \xrightarrow{\text{firstD}} \texttt{rest}(R)$.
(e) For $R = e_1|\ldots|e_n \in \mathbb{R}$, $\$R \xrightarrow{\text{firstD}} \$P$ for each $P \xrightarrow{\text{firstG}} R$.

As before, we define $\xrightarrow{\text{FirstD}}$ as the transitive closure of $\xrightarrow{\text{firstD}}$; the dual grammar is left-recursive if $e \xrightarrow{\text{FirstD}} e$ for some $e \in \mathbb{E}_D$.

A *cycle* in the grammar is a sequence of choice expressions $R_1, R_2, \ldots, R_n$ with $R_n = R_1$ where R_i has R_{i+1} as one of the alternatives for $1 \le i < n$. The example above has cycle `A`, `A`.

Proposition 1. *The dual grammar is left-recursive if and only if G contains a cycle.*

Proof. ($\Rightarrow$ part:) Suppose G has a cycle $R_1, \ldots, R_n$. Clearly, $R_i \overset{\text{firstG}}{\longrightarrow} R_{i+1}$.
According to (e), $\$R_{i+1} \overset{\text{firstD}}{\longrightarrow} \$R_i$ for $1 \leq i < n$. Thus $\$R_1 \overset{\text{FirstD}}{\longrightarrow} \$R_1$.
($\Leftarrow$ part:) Suppose there exist expressions $e_1, e_2, \ldots, e_k \in \mathbb{E}_D$ such that $e_1 \overset{\text{firstD}}{\longrightarrow} \ldots \overset{\text{firstD}}{\longrightarrow} e_k$ and $e_k = e_1$.
Suppose $e_1 \in \mathbb{E}_C$. If $e_1 \in \overline{\mathbb{R}}$, then according to (a)-(b), e_2 is one of the components of e_1; and we have $e_1 \overset{\text{firstG}}{\longrightarrow} e_2$. It also belongs to $\mathbb{E}_C$.
If $e_1 \in \mathbb{A}$, then according to (c), e_2 is a seed S of class $\mathbb{C}(e_1)$, so $e_1 \overset{\text{FirstG}}{\longrightarrow} S$. Again, it belongs to $\mathbb{E}_C$.
In each case, $e_1 \overset{\text{FirstG}}{\longrightarrow} e_2$ and $e_2 \in \mathbb{E}_C$. This can be repeated for $e_2, \ldots e_{k-1}$ showing that all e_i are in $\mathbb{E}_C$ and $e_i \overset{\text{FirstG}}{\longrightarrow} e_{i+1}$ for $1 \leq i < k$. The latter means that none of E_i can be in $\overline{\mathbb{R}}$. Suppose they are all in $\mathbb{A}$. According to (c), they must all be seeds, and so in particular e_2. But, from $e_1 \overset{\text{FirstG}}{\longrightarrow} e_2 \overset{\text{FirstG}}{\longrightarrow} e_1$ follows that e_2 is both a seed and a member of $\mathbb{C}(e_1)$, which contradicts the definition.
Hence, all e_i must be expressions $\$R_i$. But we have $\$R_i \overset{\text{firstD}}{\longrightarrow} \$R_{i+1}$ only if R_i is a choice expression. We have then $R_{i+1} \overset{\text{firstG}}{\longrightarrow} R_i$, which means R_i is one of the alternatives of R_{i+1}. Thus, G contains cycle $R_k, R_{k-1}, \ldots, R_1$. □

6 Equivalence

The next thing to check is whether the dual grammar defines the same language as the original one. We understand it so that for each $e \in \mathbb{E}_C$, $\mathcal{L}_D(e) = \mathcal{L}_G(e)$, with e interpreted according to its grammar. The language $\mathcal{L}_D(e)$ is defined in the same way as $\mathcal{L}_G(e)$, with the addition that ε is replaced by nothing.

We shall carry out the proof in terms of syntax trees. Figure 3 shows syntax trees for our examples according to their dual grammars. We refer to them as D-syntax trees, while the original ones are G-syntax trees. In the following, $e \lhd_G x$ will stand for G-syntax tree of x with root e, and $e \lhd_D x$ for similar D-syntax tree. We write $e \lhd [e_1 \lhd x_1 + \cdots + e_n \lhd x_n] x$ to represent syntax tree with root e and branches $e_1 \lhd x_1, \ldots, e_n \lhd x_n$, where $x_1 \ldots x_n = x$.

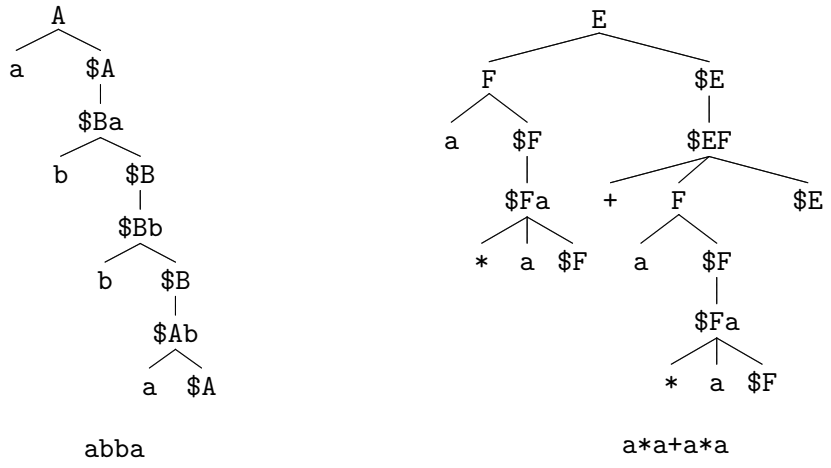


Figure 3: D-syntax trees for examples from Figure 1 (ε-branches omitted)

Proposition 2. $\mathcal{L}_D(e) = \mathcal{L}_G(e)$ *for each* $e \in \mathbb{E}_C$.

Proof. The proof is by induction on the height of syntax tree.

(Induction base:) Syntax tree of height 0 for a terminal a. It is identical for both grammars..

(Induction step:) Is handled by Lemmas 1 and 2 found in the Appendix. Assume that for each $e \lhd_G x$ (respectively $e \lhd_D x$) of height $h \geq 0$ where $e \in \mathbb{E}_C$ exists syntax tree $e \lhd_D x$ (respectively $e \lhd_G x$). Lemma 1 (respectively Lemma 2) shows that the same holds for each $e \lhd_G x$ (respectively $e \lhd_D x$) of height $h + 1$.

By induction, for each $e \lhd_G x$ where $e \in \mathbb{E}_C$ exists $e \lhd_D x$ and vice-versa. □

7 Dual grammar as PEG

We look now at the dual grammar as PEG. If the original grammar does not have cycles, D is not left-recursive and, as shown in [6, 13], each of its expressions will terminate when applied to an arbitrary input.

Because of limited backtracking, PEG may fail to explore some paths, and fail on input conforming to the BNF understanding of the grammar. Still, the limited backtracking may be "efficient" in the sense that it does not miss anything. According to [13], the backtracking is efficient if for every choice expression $e = e_1 | \ldots | e_n \in \mathbb{E}_D$ holds

$$\mathcal{L}_D(e_i)\Sigma^* \cap \mathcal{L}_D(e_j)\texttt{Tail}_D(e) = \varnothing \tag{3}$$

where $1 \leq i < j \leq n$, and $\texttt{Tail}_D(e)$ is the set of strings that may follow the application of expression e.

The dual grammar has only two kinds choice expressions in addition to those in the original grammar, namely those in (1) and (2). Because of the way the $\texttt{pre}(R)$-expressions are nested, the strings that may follow these choice expressions are exactly those that may follow the original entry expression. Thus, the $\texttt{Tail}_D(E)$ in their condition (3) is for the expression E that started the ascent.

The difference between PEG and BNF is that choice expressions in PEG are ordered, and this can affect the validity of (3). The BNF choices can be freely reordered, which may help to satisfy (3). However, the empty alternative present in $\texttt{pre}(E)$ for entry E must be last, for otherwise it would block other alternatives.

From the way the ascent procedures are nested follows that the only way for successful termination is application of that empty alternative. In PEG this means that all the remaining alternatives failed. In the example in Section 3, there would be another step failing to try the predecessor `Ab` of `A`. It is a "greedy" behavior that stops when any further ascent fails.

One can easily see that (3) always holds if the grammar has the LL(1) property, which was discovered in [11]. But, checking (3) in the general case is not simple because emptiness of intersection is undecidable for context-free languages. In many practical cases, it can be done by inspection. Outlined in [14] is an interactive tool that allows to successively replace elements of expressions in (3), to emulate the process of generating $\mathcal{L}_D(e_i)\Sigma^*$ and $\texttt{Tail}_D(e)$. This often helps to decide whether or not these languages are disjoint.

8 Computing the Tail

In the preceding section, $\texttt{Tail}_D(e)$ was defined informally as the set of strings that may follow the application of expression $e \in \mathbb{E}_D$. A precise definition requires that e is considered in a specific context. The following applies to any BNF grammar, so we shall skip for the moment the subscripts G and D.

We assume that one expression, S, is designated as the *start expression*, and does not appear in any other expression. The application of e is considered in the context of strings derived from S, that is,

in the syntax tree with root S. If the syntax tree $s \lhd w$ contains e, there exists a subtree $e \lhd y$ such that $w = xyz$ for some $x, z \in \Sigma^*$. $\texttt{Tail}(e)$ is defined as the set of all strings z that can occur in such context. For expression E containing e, define $\texttt{next}(e, E)$, meaning "next after e in E" as follows:

- For $E = e_1 \dots e_n$ and $e = e_i$, $\texttt{next}(e, e') = e_{i+1} \dots e_n$ (with ε if $i = n$).
- For $E = e_1 | \dots | e_n$, $\texttt{next}(e, E) = \varepsilon$.

Then introduce new expressions:

$$\tau e \to \texttt{next}(e, E_1)\, \tau E_1 \mid \dots \mid \texttt{next}(e, E_m)\, \tau E_m \quad \text{for } e \neq S, \tag{4}$$

$$\tau S \to \vdash, \tag{5}$$

where $E_1, \dots, E_m$ are all expressions containing e, and $\vdash$ stands for "end of input".

Proposition 3. *$\texttt{Tail}(e)$ is the language generated by τe.*

Proof. ($\Rightarrow$ part) Suppose $z \in \texttt{Tail}(e)$, that is, exists syntax tree $S \lhd xyz$ containing subtree $e \lhd y$. There is path $e_0, \dots, e_n$ with $e_0 = e$ and $e_n = S$. Each of e_i generates part z_i of z. This part is generated by branches of the tree $e_i \lhd x_i y z_i$ that belong to $\texttt{next}(e_i, e_{i+1})$. One can easily see that $z_{i+1} \dots z_n \in \texttt{Tail}(e_i)$. We check by induction that τe_i generates $\texttt{Tail}(e_i)$.
($\Leftarrow$ part) Suppose exists $\tau e \lhd z$. Denote $e = e_0$. z must be generated from $\texttt{next}(e_0, e_1)\, \tau e_1$ where e_1 contains e_0, so $z = z_1 y_1$ where z_1 is generated by $\texttt{next}(e_0, e_1)$ and y_1 by τe_1. This can be repeated to find that $z = z_1 z_2 ... z_i y_i$ where z_i is generated by $\texttt{next}(e_{i-1}, e_i)$ and y_i by τe_i where e_i contains e_{i-1}. This chain ends on $e_n = S$ that is not contained in any expression. We have thus a syntax tree that contains nested trees $e_n, \dots, e_0$ of the form $e_i[\dots + e_{i-1} + \dots] x_n \dots x_1 y z_1 \dots z_n$ from which it follows that $z_1 \dots z_n \in \texttt{Tail}(e_0)$. □

Take as an example grammar (b) from Figure 2 with `E` as starting expression. From (5) we have:

```
τF → $E τE | $E τ$EF
τE →  ⊢
τ$EF → τ$E
τ$E → τE | τ$EF
```

The only string the can be finitely generated by τ`$E` is $\vdash$, so after some replacements we have:

```
τF →  + F $E ⊢ | ⊢ .
```

9 Practical Implementation

The described approach has been implemented in the author's parser generator "Mouse", found in https://sourceforge.net/projects/mousepeg. The theory presented here used some simplifying assumptions that do not apply to the implementation.

One such assumption is absence of empty word ε among the expressions of G (which was then added in a limited way in D). Allowing ε means that the language of an expression may contain ε; the expression is the said to be "nullable". The presence of nullable expressions complicates the definition of $\xrightarrow{\text{firstG}}$: for $e = e_1 \dots e_n$ with nullable $e_1, \dots, e_i$ we have $e \xrightarrow{\text{firstG}} e_{i+1}$.

The implementation in "Mouse" allows nullable expressions, but keeps the restriction that the first expression in sequence $e \in \mathbb{R}$ is not nullable. Removing this restriction complicates the method in a way not explored yet.

The second simplifying assumption is that each recursion class has only one entry. This is not the case for many grammars. It is not feasible to have a separate set of ascent procedures for each entry. The solution in "Mouse" uses the fact that ascent processes are nested.

A Appendix

Lemma 1. *Assume that for each $e\lhd_G x$ of height $h \geq 1$ where $e \in \mathbb{E}_C$ exists $e\lhd_D x$. Then the same holds for each $e\lhd_G x$ of height $h+1$.*

Proof. Suppose $e \in \mathbb{E}_C$ and take any $e\lhd_G x$ of height $h+1$.

(Case 1) $e \in \overline{\mathbb{R}}$. We have $e\lhd_G[e_1\lhd_G x_1 + \cdots + e_m\lhd_G x_m]x$ where $m \geq 1$ depends on e. Each e_i is in $\mathbb{E}_C$ and each $e_i\lhd_G x_i$ has height h or less. As assumed, exists $e_i\lhd_D x_i$ for $i = 1, \ldots, m$. As e is the same in both grammars, exists $e\lhd_D[e_1\lhd_D x_1 + \cdots + e_m\lhd_D x_m]x$.

(Case 2) $e \in \mathbb{A}$. The tree $e\lhd_G x$ has recursion path $R_n \xrightarrow{\text{firstG}} \ldots \xrightarrow{\text{firstG}} R_1 \xrightarrow{\text{firstG}} R_0$ where $R_n = e$ and R_0 is a seed of $\mathbb{C}(e)$. We start by constructing $\texttt{rest}(R_i)\lhd_D r_i$ for $i = 1, \ldots, n$.
If $R_i = e_1 e_2 \ldots e_m$, we have $\texttt{rest}(R_i) = e_2 \ldots e_m$. Each of e_j for $j = 2, \ldots, m$ is in $\mathbb{E}_C$, and has its tree $e_j\lhd_G z_j$ with height less than h. As assumed, there exists $e_j\lhd_D z_j$.
We construct $\texttt{rest}(R_i)\lhd_D[e_2\lhd_D z_2 + \cdots + e_m\lhd_D z_m]r_i$.
If $R_i = e_1|\ldots|e_m$, we have $\texttt{rest}(R_i) = \varepsilon$, so there exists $\texttt{rest}(R_i)\lhd_D\varepsilon$.

We check now by induction that there exists $\$R_i\lhd_D x_i$ for $i = n, \ldots, 1$.
Induction base: For entry R_n we have $\$R_n = \texttt{rest}(R_n)\,\varepsilon$
so there exists $\$R_n\lhd_D[\texttt{rest}(R_n)\lhd_D r_n + \varepsilon\lhd_D\varepsilon]x_n$.
Induction step: Assume there exists $\$R_{i+1}\lhd_D x_{i+1}$ for $n > i \geq 1$. As $\$R_i = \texttt{rest}(R_i)\,\$R_{i+1}$, there exists $\$R_i\lhd_D[\texttt{rest}(R_i)\lhd_D r_i + \$R_{i+1}\lhd_D x_{i+1}]x_i$.
By induction exists $\$R_1\lhd_D x_1$.
In grammar D we have $e = R_0\$R_1$. The seed R_0 is in $\mathbb{E}_C$ and has G-tree of height h or less, so, by assumption, exists $R_0\lhd_D x_0$, and we have $e\lhd_D[R_0\lhd_D x_0 + \$R_1\lhd_D x_1]x$. □

Lemma 2. *Assume that for each $e\lhd_D x$ of height $h \geq 1$ where $e \in \mathbb{E}_C$ exists $e\lhd_G x$. Then the same holds for each $e\lhd_D x$ of height $h+1$.*

Proof. Suppose $e \in \mathbb{E}_C$ and take any $e\lhd_D x$ of height $h+1$.

(Case 1) $e \in \overline{\mathbb{R}}$.
We have $e\lhd_D[e_1\lhd_D x_1 + \cdots + e_m\lhd_D x_m]x$ where $m \geq 1$ depends on e. Each e_i is in $\mathbb{E}_C$ and each $e_i\lhd_D x_i$ has height h or less. As assumed, exists $e_i\lhd_G x_i$ for $i = 1, \ldots, m$. As e is the same in both grammars, exists $e\lhd_G[e_1\lhd_G x_1 + \cdots + e_m\lhd_G x_m]x$.

(Case 2) $e \in \mathbb{A}$. W have $e = R_0\,\$R_1$.
Its D-tree is $e\lhd_D[R_0\lhd_D x_0 + \$R_1\lhd_D x_1]x$, where R_0 is a seed of $\mathbb{C}(e)$. The seed belongs to $\mathbb{E}_C$ and the tree $R_0\lhd_D x_0$ has height h or less, so, by assumption, exists $R_0\lhd_G x_0$.
The tree $\$R_1\lhd_D y_1$ consists of subtrees with roots $\$R_i$ for $1 \leq i \leq n$. such that $R_i \xrightarrow{\text{firstG}} R_{i-1}$ and $R_n = e$.

We check now by induction that there exists $R_i\lhd_G x_i$ for $i = 1, \ldots, n$.
Induction base: Exists $R_0\lhd_G x_0$.
Induction step: Assume there exists tree $R_{i-1}\lhd_G x_{i-1}$, where $1 \leq i \leq n$.
If $R_i = e_1|\ldots|e_m$, from $R_i \xrightarrow{\text{firstG}} R_{i-1}$ follows that R_{i-1} is one of e_j. We construct $R_i\lhd_G[R_{i-1}\lhd_G x_{i-1}]x_i$ where $x_i = x_{i-1}$.
If $R_i = e_1 \ldots e_m$, from $R_i \xrightarrow{\text{firstG}} R_{i-1}$ follows that $R_{i-1} = e_1$ and $R_i = R_{i-1}e_2 \ldots e_m$. There exists tree $\texttt{rest}(R_i)\lhd_D[e_2\lhd_D z_2 + \cdots + \lhd_D z_m]y_i$. Each of e_j is in $\mathbb{E}_C$ and each tree $e_j\lhd_D z_j$ has height less than h. By assumption, there exist $e_2\lhd_G z_2, \ldots, \lhd_G z_m$.
We construct $R_i\lhd_G[R_{i-1}\lhd_G x_{i-1} + e_2\lhd_G z_2 + \cdots + \lhd_G z_m]x_i$.
By induction exists $R_n\lhd_G x_n$. □

References

[1] Aho, A. V., Ullman, J. D.: *The Theory or Parsing, Translation, and Compiling. Vol. I: Parsing*, Prentice-Hall, 1972.

[2] Birman, A.: *The TMG Recognition Schema*, Ph.D. Thesis, Princeton University, February 1970.

[3] Birman, A., Ullman, J. D.: Parsing Algorithms with Backtrack, *Information and Control*, **23**, 1973, 1–34.

[4] Brooker, P., Morris, D.: A General Translation Program for Phrase Structure Languages, *Journal of the ACM*, **9**(1), 1962, 1–10.

[5] Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002* (M. Wand, S. L. P. Jones, Eds.), ACM, 2002.

[6] Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004* (N. D. Jones, X. Leroy, Eds.), ACM, Venice, Italy, 14–16 January 2004.

[7] Hill, O.: Support for Left-Recursive PEGs, 2010, `https://github.com/orlandohill/peg-left-recursion`.

[8] Hopgood, F. R. A.: *Compiling Techniques*, MacDonalds, 1969.

[9] Lucas, P.: The Structure of Formula-Translators, *ALGOL Bulletin Supplement*, **16**, September 1961, 1–27.

[10] McClure, R. M.: TMG – a syntax directed compiler, *Proceedings of the 20th ACM National Conference* (L. Winner, Ed.), ACM, 24–26 August 1965.

[11] Medeiros, S.: *Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto*, Ph.D. Thesis, Pontifícia Universidade Católica do Rio de Janeiro, August 2010.

[12] Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: Left Recursion in Parsing Expression Grammars, *Science of Computer Programming*, **96**, 2014, 177–190.

[13] Redziejowski, R. R.: More about converting BNF to PEG, *Fundamenta Informaticae*, **133**(2-3), 2014, 177–191.

[14] Redziejowski, R. R.: Trying to understand PEG, *Fundamenta Informaticae*, **157**, 2018, 463–475.

[15] Tratt, L.: *Direct left-recursive parsing expression grammars*, Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, October 2010.

[16] Warth, A., Douglass, J. R., Millstein, T. D.: Packrat parsers can support left recursion, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8*, 2008.