More about left recursion in PEG

Roman R. Redziejowski

roman@redz.se

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 technique of parsing by recursive descent assigns to each grammatical construct a procedure that calls other procedures to process components of that construct. As grammars are usually defined in a recursive way, these calls are recursive. This method encounters two problems:

(1) Procedures corresponding to certain type of construct must chose which procedure to call next.

(2) If the grammar contains left recursion, a procedure may call itself indefinitely.

Problem (1) has been traditionally solved by looking at the next input symbol(s), which works if the grammar satisfies a condition known as LL(n). Another way is to try the alternatives one by one, backtracking in the input, until one succeeds (or all fail).

Making full search can require exponential time, so a possible option is limited backtracking: never return after a partial success. This method has been used in actual parsers [4,10] and is described in literature [1–3,7]. It has been eventually formalized by Ford [5] under the name of Parsing Expression Grammar (PEG).

Problem (2) is serious because of a strong tendency to present grammars in left-recursive form. Converting the grammar to right-recursive form is possible, but is tedious, error-prone, and obscures the spirit of the grammar.

The problem has been in the recent years solved by what can be called "recursive ascent". Each recursion must end up in a part of syntax tree that does not involve further recursion. It has been referred to as the "seed". After identifying the seed, one reconstructs the syntax tree upwards, in the process of "growing the seed". Extensions to PEG using this method have been described in [6,8,12,15–17].

The paper tries to find out under which conditions this process will work correctly. The idea of "working correctly" needs an explanation. One of the most common ways to define the syntax of a formal language is the Backus-Naur Form (BNF) or its extended version EBNF. We treat here PEG as a parser for BNF that implements recursive descent with limited backtracking. Because of limited backtracking, PEG may miss some strings that belong to the language defined by BNF. The author has previously tried to answer the question under which

conditions PEG will accept exactly the language defined by BNF (see [13, 14]). This paper tries to answer the same question for PEG equipped with recursive ascent technique to handle left recursion.

We look here at a process inspired by Hill [6]. Section 2 introduces a subset of BNF grammar with natural semantics that is a slight modification of that due to Medeiros [9, 11]. In Section 3 we develop some concepts needed to discuss left recursion. In Section 4 we describe the parsing process, check that it terminates, and state the conditions under which it reproduces the BNF syntax tree. The last section contains some comments. Proofs of the Propositions are found in Appendix.

2 The BNF grammar

We consider an extremely simplified form of BNF grammar over alphabet Σ. It is a set of *rules* of the form $A = e$ where A belongs to a set N of symbols distinct from the letters of Σ and e is an *expression*. Each expression is one of these:

$$\begin{array}{ll} \varepsilon \text{ (”empty”)}, & e_1 e_2 \text{ (”sequence”)}, \\ a \in \Sigma \text{ (”letter”)}, & e_1 | e_2 \text{ (”choice”)}. \\ A \in N \text{ (”nonterminal”)}. & \end{array}$$

where each of e_1, e_2 is an expression and ε denotes empty word. The set of all expressions is in the following denoted by $\mathbb{E}$. There is exactly one rule $A = e$ for each $A \in N$. The expression e appearing in this rule is denoted by $e(A)$.

Each expression $e \in \mathbb{E}$ has its language $\mathcal{L}(e) \subseteq \Sigma^*$ defined formally by natural semantics shown in Figure 1. String x belongs to $\mathcal{L}(e)$ if and only if $[e]x \overset{\text{BNF}}{\to} x$ can be proved using the inference rules from Figure 1. Note that if a proof of $[e]x \overset{\text{BNF}}{\to} x$ exists, one can prove $[e]xy \overset{\text{BNF}}{\to} x$ for every $y \in \Sigma^*$

$$\frac{}{[\varepsilon]w \overset{\text{BNF}}{\to} \varepsilon}\ (\textit{empty}) \qquad \frac{}{[a]aw \overset{\text{BNF}}{\to} a}\ (\textit{letter}) \qquad \frac{[e(A)]w \overset{\text{BNF}}{\to} x}{[A]w \overset{\text{BNF}}{\to} x}\ (\textit{rule})$$

$$\frac{[e_1]xw \overset{\text{BNF}}{\to} x \quad [e_2]w \overset{\text{BNF}}{\to} y}{[e_1 e_2]xw \overset{\text{BNF}}{\to} xy}\ (\textit{seq})$$

$$\frac{[e_1]w \overset{\text{BNF}}{\to} x}{[e_1|e_2]w \overset{\text{BNF}}{\to} x}\ (\textit{choice1}) \qquad \frac{[e_2]w \overset{\text{BNF}}{\to} x}{[e_1|e_2]w \overset{\text{BNF}}{\to} x}\ (\textit{choice2})$$

Fig. 1. Formal semantics of BNF

One can read $[e]w \overset{\text{BNF}}{\to} x$ as saying that expression e matches prefix x of string w. Thus, $x \in \mathcal{L}(e)$ means that e matches the string x. We say that expression e is *nullable* to mean that $\varepsilon \in \mathcal{L}(e)$.

Figure 2 is an example of formal proof using the rules from Figure 1. It verifies that the string *baac* belongs to $\mathcal{L}(S)$ as defined by this grammar:

```
S = Ac    A = Aa|B    B = b
```

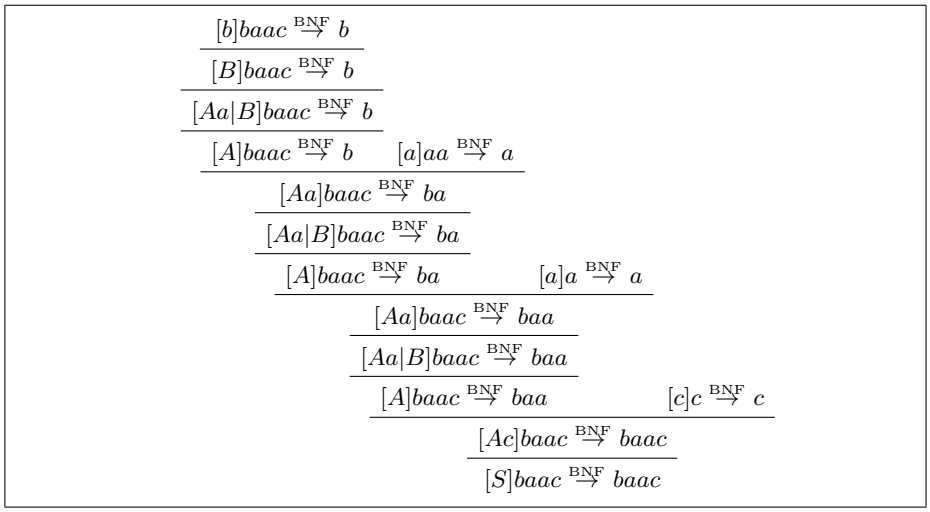

Fig. 2. Example of BNF proof

The proof can be represented in the inverted form shown in Figure 3. This seems more intuitive when speaking about "recursive descent". The diagram on the right is simplified to show only the expressions. It is the syntax tree of *baac*.

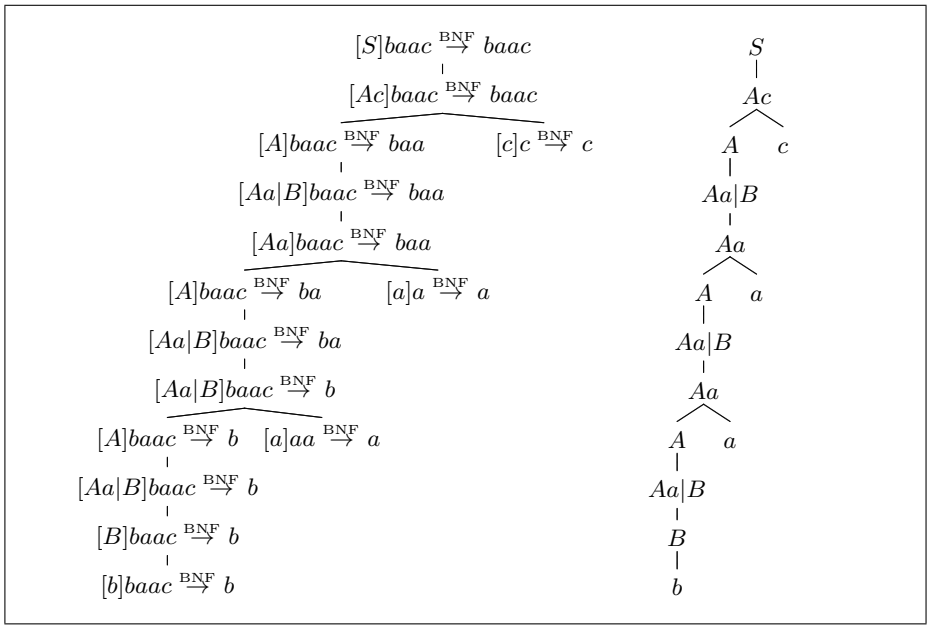

Fig. 3. BNF tree and syntax tree

3 Left recursion

3.1 Recursion classes

For $e \in \mathbb{E}$ define $\texttt{first}(e)$ as follows:

$$\texttt{first}(\varepsilon) = \texttt{first}(a) = \varnothing, \qquad \texttt{first}(A) = \{e(A)\},$$

$$\texttt{first}(e_1|e_2) = \{e_1, e_2\}, \qquad \texttt{first}(e_1e_2) = \begin{cases} \{e_1\} \text{ if } \varepsilon \notin \mathcal{L}(e_1), \\ \{e_1, e_2\} \text{ if } \varepsilon \in \mathcal{L}(e_1). \end{cases}$$

Define further $\texttt{First}$ to be the transitive closure of relation $\texttt{first}$. In the following, we write $e \xrightarrow{\text{first}} e'$ to mean that $e' \in \texttt{first}(e)$, and $e \xrightarrow{\text{First}} e'$ to mean that $e' \in \texttt{First}(e)$.

An expression e is *left-recursive* if $e \xrightarrow{\text{First}} e$. Let $\mathbb{R} \subseteq \mathbb{E}$ be the set of all left-recursive expressions. Define relation $\texttt{Rec} \subseteq \mathbb{R} \times \mathbb{R}$ so that $(e_1, e_2) \in \texttt{Rec}$ means $e_1 \xrightarrow{\text{First}} e_2 \xrightarrow{\text{First}} e_1$. It is an equivalence relation that defines a partition of $\mathbb{R}$ into equivalence classes; we refer to them as *recursion classes*. The recursion class containing expression e is denoted by $\mathbb{C}(e)$.

Let e be an expression belonging to recursion class $\mathbb{C}$. If $e = A \in N$ or $e = e_1e_2$ with non-nullable e_1, the expression $e' \in \texttt{first}(e)$ must also belong to $\mathbb{C}$. This is so because e' is the only element in $\texttt{first}(e)$, and we must have $e \xrightarrow{\text{first}} e' \xrightarrow{\text{First}} e$ to achieve $e \xrightarrow{\text{First}} e$. In $e = e_1|e_2$ or $e = e_1e_2$ with nullable e_1, one of expressions e_1 or e_2 may be outside $\mathbb{C}$. It is a *seed* of $\mathbb{C}$, and e is an *exit* of $\mathbb{C}$. For $e = e_1e_2$ in $\mathbb{C}$, e_2 is a *leaf* of $\mathbb{C}$. The set of all seeds of $\mathbb{C}$ is denoted by $\texttt{Seed}(\mathbb{C})$ and the set of all its leafs by $\texttt{Leaf}(\mathbb{C})$.

As an example, the grammar used in Figure 2 has $\mathbb{R} = \{A, Aa|b, Aa\}$. All these expressions belong to the same recursion class with exit $Aa|b$, seed b and leaf a.

To simplify the discussion, we asume in the following that all exits have the form $e_1|e_2$, that is, $e_1 \in \mathbb{R}$ in e_1e_2 is not nullable. As the ordering of BNF choice expression does not influence its language, we assume that e_2 in $e_1|e_2$ is always the seed.

3.2 Recursion sequence

Consider a node in the syntax tree and the leftmost branch emanating from it. It is a chain of nodes connected by $\xrightarrow{\text{first}}$. If the branch contains any left-recursive expressions, expressions belonging to the same recursion class $\mathbb{C}$ must form an uninterrupted sequence. Indeed, if e_1 and e_2 appearing in the branch belong to $\mathbb{C}$, we have $e_1 \xrightarrow{\text{First}} e \xrightarrow{\text{First}} e_2 \xrightarrow{\text{First}} e_1$ for any e in between. Such sequence is a *recursion sequence* of class $\mathbb{C}$. The same argument shows that the branch can contain at most one recursion sequence of a given class, and that sequences of different classes cannot be nested.

The last expression in the sequence must be an exit of $\mathbb{C}$. One can easily see that each recursion class must have at least one exit. We assume in the following that this is the case.
Let $e \xrightarrow{\text{first}} e'$ be two consecutive expressions in a recursion sequence. Let $[e]w \overset{\text{BNF}}{\to} x$ and $[e']w \overset{\text{BNF}}{\to} x'$. Expression e can be one of these:

- $A = e'$. Then $x = x'$ according to (*rule*).
- $e'|e_2$. Then $x = x'$ according to (*choice1*).
- $e_1|e'$. Then $x = x'$ according to (*choice2*).
- $e'e_2$. Then $x = x'y$ where $y \in \mathcal{L}(e_2)$ according to (*seq*).

Define $\texttt{adv}(e, e') = \{\varepsilon\}$ in each of the first three cases and $\texttt{adv}(e, e') = \mathcal{L}(e_2)$ in the last. For a recursion sequence $s = e_{[n]}, e_{[n-1]}, \ldots, e_{[2]}, e_{[1]}$ define

$$\texttt{adv}(s) = \texttt{adv}(e_{[n]}, e_{[n-1]}) \ldots \texttt{adv}(e_{[2]}, e_{[1]}).$$

For $e \in \mathbb{R}$ define $\texttt{Adv}(e)$ to be the union of $\texttt{adv}(s)$ for all recursion sequences s starting with e, ending with e, and not containing e. The following is easy to see:

Lemma 1. *If $[e]w \overset{BNF}{\to} x$ and $[e]w \overset{BNF}{\to} x'$ are two consecutive results for the same e in a recursion sequence, we have $x \in x'\texttt{Adv}(e)$.*

4 Parsing

Given a BNF grammar, we define for each $e \in \mathbb{E}$ a parsing procedure named $[e]$. The procedure may return "success" after possibly "consuming" some input, or "failure" without consuming anything. In the following, the result of calling $[e]$ for input w is denoted by $[e]w$; it is either `fail` or the consumed string x (possibly ε).

The action of parsing procedure $[e]$ for $e \notin \mathbb{R}$ is the same as in PEG and is shown in Figure 4.

The procedure $[e]$ for $e \in \mathbb{R}$ is a "grower" for the recursion class $\mathbb{C}(e)$ and input string w. The grower has a "plant" $[e', w]$ for each expression $e' \in \mathbb{C}(e)$. The plant is a procedure with memory. The procedure emulates the action of parsing procedure $[e']$, but may use results from other plants instead of calls to parsing procedures. It computes a result as shown in Figure 5. The memory holds the result of procedure applied to w. It is denoted by $\langle e', w\rangle$. The grower initializes all its plants with $\langle e', w\rangle = \texttt{fail}$, and then repeatedly calls all plants in some order. If the result is better than already held by the plant, it replaces the latter. (A string is better than `fail`, and longer string is better than shorter one.) The grower stops when it cannot improve any result. The result in $[e, w]$ is then the result of parsing procedure $[e]$.

In order to create a formal record of parsing process, we represent actions of parsing procedures and plants by inference rules shown in Figure 6. A rule with conclusion $[e]w = X$ represents a call to $[e]$ returning X. One with conclusion $\langle e, w\rangle = x$ represents setting new result x in $\langle e, w\rangle$. A premise $[e']w = X$

represents call a to sub-procedure $[e']$ returning X; a premise $\langle e', w\rangle = X$ represents result obtained from $[e', w]$. With these conventions, parsing process is represented as a formal proof.

An example of such proof is shown in Figure 7. It represents parsing process for the string and grammar from example in Figure 3. Note that part of that tree was constructed by going down and part by going up.

$[\varepsilon]$	Indicate success without consuming any input.
$[a]$	If the text ahead starts with a, consume a and return success. Otherwise return failure.
$[A = e_1]$	Call $[e_1]$ and return result.
$[e_1\, e_2]$	Call $[e_1]$. If it succeeded, call $[e_2]$ and return success if $[e_2]$ succeeded. If $[e_1]$ or $[e_2]$ failed, backtrack: reset the input as it was before the invocation of $[e_1]$ and return failure.
$[e_1 \mid e_2]$	Call $[e_1]$. Return success if it succeeded. Otherwise call $[e_2]$ and return success if $[e_2]$ succeeded or failure if it failed.

Fig. 4. Actions of parsing procedures

$[A, w]$	Return $\langle e(A), w\rangle$.
$[(e_1\, e_2), w]$	If $\langle e_1, w\rangle = \texttt{fail}$, return `fail`. If $\langle e_1, w\rangle = x$, call $[e_2]$ on z where $w = xz$ and restore input to w. If $[e_2]z = \texttt{fail}$, return `fail`. Otherwise return $x\,[e_2]z$.
$[(e_1 \mid e_2), w]$	If $\langle e_1, w\rangle \neq \texttt{fail}$, return $\langle e_1, w\rangle$. If $\langle e_1, w\rangle = \texttt{fail}$ and there exists plant $[e_2, w]$, return $\langle e_2, w\rangle$. Otherwise call $[e_2]$, restore input to w, and return $[e_2]w$.

Fig. 5. Actions of plants

We say that "parsing procedure $[e]$ handles string w" to mean that the procedure applied to w terminates and returns either $x \in \Sigma^*$ or `fail`.

Proposition 1. *Each parsing procedure* $[e]$ *for* $e \in \mathbb{E}$ *handles all* $w \in \Sigma^*$.

Proof is found in the Appendix.

Proposition 2. *For each result* $[e]w = x$ *or* $\langle e, w\rangle = x$ *in a parse tree there exists a proof of* $[e]w \overset{BNF}{\rightarrow} x$.

Proof is by induction on height of the subtree.

The conditions under which each BNF tree has a corresponding parse tree depend on the context in which certain expression may be used. We assume that the grammar has a start expression S and use as context the BNF proof for $[S]u \overset{\text{BNF}}{\to} u$ for $u \in \Sigma^*$.

For $e \in \mathbb{E}$, we define $\texttt{Tail}(e)$ as the set of all strings that may follow a string matched by e in the proof of $[S]u \overset{\text{BNF}}{\to} u$. More precisely, it is the set of strings z in all results $[e]xz \overset{\text{BNF}}{\to} x$ that appear in that proof. A possible method for estimating $\texttt{Tail}(e)$ can be found in [14]. For $e \in \mathbb{R}$, $\texttt{Tail}_R(e)$ excludes occurrences of e in a recursion path except the first one.

The ordering of choice expression is essential for rules (*choice2.p*), (*choice2.g*), and(*choice3.g*). But, the ordering does not affect the language defined by BNF rules. Therefore, we can always rearrange the choice as is best for the parsing.

Proposition 3. *If the grammar satisfies the following conditions (1)-(3) then for each proof of* $[S]u \overset{BNF}{\to} u$ *there exists parse tree with root* $[S]u = u$*. Moreover, for each subproof* $[e]w \overset{BNF}{\to} x$ *of that proof exists parse tree with root* $[e]w = x$ *or* $\langle e, w\rangle = x$*.*

$$\textit{For each}\ \ e = e_1|e_2 \in \mathbb{R},\ \ e_2 \notin \mathbb{C}(e); \tag{1}$$

$$\textit{For each}\ \ e = e_1|e_2,\ \ \mathcal{L}(e_1)\Sigma^* \cap \mathcal{L}(e_2)\mathit{Tail}(e) = \varnothing; \tag{2}$$

$$\textit{For each}\ \ e \in \mathbb{R},\ \ \mathit{Adv}(e)\Sigma^* \cap \mathit{Tail}_R(e) = \varnothing. \tag{3}$$

Proof is found in the Appendix.

5 Comments

We presented sufficient conditions under which the extended PEG is a correct parser for BNF grammars. Because we treat PEG as parser for BNF, it does not include syntactic predicates of classical PEG.

We chose here the scheme for handling left recursion inspired by [6] because it seems easy to analyze. The scheme where the grower scans all plants even if nothing changes is also easy to analyze; in a practical implementation the plant would be called only if its argument(s) change.

Checking (2) in the presence of left recursion is not simple. Without left recursion, one can use approximation by prefixes, with LL(1) as the extreme case. The languages defined by left recursion often have identical prefixes and differ only at the far end.

The chosen scheme required a rather severe restriction (1) to the grammar. It seems possible to replace it by a requirement that languages of different seeds of the same recursion class are disjoint, as well as languages of expressions in $\texttt{first}^{-1}(e)$ for $e \in \mathbb{R}$. This is the subject of further research, as well as attempts to analyze other schemes.

$$\frac{}{[\varepsilon]w = \varepsilon}\ (\mathit{empty.p}) \qquad \frac{[e(A)]w = X}{[A]w = X}\ (\mathit{rule.p}) \qquad \frac{\langle e, w\rangle = X}{[e]w = X}\ (\mathit{grow.p})$$

$$\frac{}{[a]aw = a}\ (\mathit{letter1.p}) \qquad \frac{b \neq a}{[b]aw = \mathtt{fail}}\ (\mathit{letter2.p}) \qquad \frac{}{[a]\varepsilon = \mathtt{fail}}\ (\mathit{letter3.p})$$

$$\frac{[e_1]xz = x \quad [e_2]z = y}{[e_1 e_2]xz = xy}\ (\mathit{seq1.p}) \qquad \frac{[e_1]w = \mathtt{fail}}{[e_1 e_2]w = \mathtt{fail}}\ (\mathit{seq2.p})$$

$$\frac{[e_1]xz = x \quad [e_2]z = \mathtt{fail}}{[e_1 e_2]xz = \mathtt{fail}}\ (\mathit{seq3.p})$$

$$\frac{[e_1]w = x}{[e_1 \,|\, e_2]w = x}\ (\mathit{choice1.p}) \qquad \frac{[e_1]w = \mathtt{fail} \quad [e_2]w = X}{[e_1 \,|\, e_2]w = X}\ (\mathit{choice2.p})$$

$$\frac{\langle e(A), w\rangle = X}{\langle A, w\rangle = X}\ (\mathit{rule.g}) \qquad \frac{\langle e_1, xz\rangle = x \quad [e_2]z = y}{\langle (e_1 e_2), xz\rangle = xy}\ (\mathit{seq1.g})$$

$$\frac{\langle e_1, xz\rangle = x \quad [e_2]z = \mathtt{fail}}{\langle (e_1 e_2), xz\rangle = \mathtt{fail}}\ (\mathit{seq2.g}) \qquad \frac{\langle e_1, w\rangle = \mathtt{fail}}{\langle (e_1 e_2), w\rangle = \mathtt{fail}}\ (\mathit{seq3.g})$$

$$\frac{\langle e_1, w\rangle = x}{\langle (e_1 \,|\, e_2), w\rangle = x}\ (\mathit{choice1.g}) \qquad \frac{\langle e_1, w\rangle = \mathtt{fail} \quad \langle e_2, w\rangle = X}{\langle (e_1 \,|\, e_2), w\rangle = X}\ (\mathit{choice2.g})$$

$$\frac{\langle e_1, w\rangle = \mathtt{fail} \quad [e_2]w = X}{\langle (e_1 \,|\, e_2), w\rangle = X}\ (\mathit{choice3.g}) \qquad \frac{e \in \mathbb{R}}{\langle e_1, w\rangle = \mathtt{fail}}\ (\mathit{init.g})$$

where X denotes x or $\mathtt{fail}$.

Fig. 6. Formal semantics of parser

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\langle Aa, baac\rangle = \mathtt{fail} \quad \cfrac{[b]baac = b}{[B]baac = b}}{\langle Aa|B, baac\rangle = b}}{\langle A, baac\rangle = b} \quad [a]aac = a}{\langle Aa, baac\rangle = ba}}{\langle Aa|B, baac\rangle = ba}}{\langle A, baac\rangle = ba} \quad [a]ac = a}{\langle Aa, baac\rangle = baa}}{\langle Aa|B, baac\rangle = baa}}{\langle A, baac\rangle = baa}}{[A]baac = baa} \quad [c]c = c}{[Ac]baac = baac}}{[S]baac = baac}$$

Fig. 7. Example of parse tree

A Appendix

A.1 Proof of Proposition 1

The proof is by induction on the length of w with Lemma 3 as induction base and Lemma 2 as induction step. □

Lemma 2. *If each parsing procedure handles all words of length n or less, each parsing procedure handles all words of length $n+1$.*

Proof. Let the *rank* of expression e, denoted $\rho(e)$, be defined as follows:

- $\rho(\varepsilon) = \rho(a) = 0$, and otherwise:
- For $e \notin \mathbb{R}$, highest $\rho(e')$ for $e' \in \texttt{First}(e)$ plus 1;
- For $e \in \mathbb{R}$, highest $\rho(e')$ for $e' \in \texttt{Seed}(\mathbb{C}(e)) \cup \texttt{Leaf}(\mathbb{C}(e))$ plus 1.

One can easily see that this definition is not circular.

Assume that each procedure handles all words of length n or less, and consider a word w of length $n+1$. We use induction on rank of e to show that each $[e]$ handles w.

(Induction base:) Obviously, each procedure of rank 0 handles w.

(Induction step:) Assume that each procedure of rank m or less handles all words of length $n+1$ or less. Take any procedure $[e]$ of rank $m+1$.
If $e \notin \mathbb{R}$, e can be one of these:

- $A \in N$. We have $\rho(e(A)) = m$; thus $e(A)$ handles w, and so does A.
- $e_1 e_2$ with nullable e_1. We have $\rho(e_1) \leq m$ and $\rho(e_2) \leq m$. Each of them handles words of length $n+1$ or less, so $e_1 e_2$ handles w.
- $e_1 e_2$ with non-nullable e_1. We have $\rho(e_1) = m$, so e_1 handles w. If it fails, so does $e_1 e_2$. Otherwise it consumes $x \neq \varepsilon$ and e_2 is applied to the rest w' of w, with length n or less. Thus, e_2 handles w', so $e_1 e_2$ handles w.
- $e_1 | e_2$. We have $\rho(e_1) \leq m$ and $\rho(e_2) \leq m$. Each of them handles words of length $n+1$ or less, so $e_1 | e_2$ handles w.

If $e \in \mathbb{R}$, the result of $[e]$ is obtained by grower for the recursion class $\mathbb{C}(e)$ and input w. The grower stops when it cannot improve any result. Each improvement means consuming more of w. Since w is finite, the grower must eventually stop. Thus, $[e]$ handles w. □

Lemma 3. *Each parsing procedure handles word of length 0.*

Proof. The proof is essentially the same as that of Lemma 2, with w replaced by ε, and simplified case of $e_1 e_2$ with non-nullable e_1. □

A.2 Proof of Proposition 3

Suppose we are given a BNF proof of $[S]u \overset{\text{BNF}}{\to} u$. We are going to show that each partial proof of that proof (and the final proof) has the corresponding parse tree. In the following, we say "subtree $[e]w \overset{\text{BNF}}{\to} x$" to mean the partial proof with result $[e]w \overset{\text{BNF}}{\to} x$. Define the *level* of subtree $[e]w \overset{\text{BNF}}{\to} x$ as follows:

– For $e = \varepsilon$ and $e = a$ the level is 1.
– For $e \notin \mathbb{R}$, other than above, the level is 1 plus the highest level of subtrees for components of e.
– Each result with $e \in \mathbb{R}$ belongs to some recursion sequence. All subtrees in that sequence have the same level, equal to 1 plus the highest level of subtrees for the seed and leafs of that recursion sequence.

The proof is by induction on the level.

(Induction base:) The parse trees for subtrees on level 1 are $[\varepsilon]w = \varepsilon$ respectively $[a]az = a$. They represent calls to parsing procedures $[\varepsilon]$ and $[a]$.

(Induction step:) Assume that there exists parse tree for each subtree on level n or less. We show that there exists parse tree for each subtree on level $n+1$.
For a subtree $[e]w \overset{\text{BNF}}{\to} x$ on level $n+1$ where $e \notin \mathbb{R}$ we construct parse tree from parse trees of subtrees. This is done in Lemma 4 and represents calls from $[e]$ to its subprocedures.
The root of each subtree $[e]w \overset{\text{BNF}}{\to} x$ on level $n+1$ where $e \in \mathbb{R}$ belongs to some recursion sequence (perhaps degenerated to length 1). We construct parse trees for all results in the sequence from parse trees for the leafs and the seed. This is done in Lemma 5 and represents work done by the grower. □

Lemma 4. *Assume there exists parse tree for each subtree of* $[e]w \overset{BNF}{\to} x$. *There exists parse tree for* $[e]w \overset{BNF}{\to} x$.

Proof. The parse tree for $[e]w \overset{\text{BNF}}{\to} x$ is constructed in the way that depends on e:

– $A = e'$. $[A]w \overset{\text{BNF}}{\to} x$ is derived from $[e']w \overset{\text{BNF}}{\to} x$ according to (*rule*).
As assumed, there exists parse tree $[e']w = x$ for $[e']w \overset{\text{BNF}}{\to} x$. The parse tree $[A]w = x$ is constructed from it using (*rule.p*).
– e_1e_2. $[e_1e_2]xz \overset{\text{BNF}}{\to} xy$ is derived from $[e_1]xz \overset{\text{BNF}}{\to} x$ and $[e_2]z \overset{\text{BNF}}{\to} y$ according to (*seq*).
As assumed, there exist parse trees $[e_1]xz = x$ and $[e_2]z = y$. for $[e_1)]xz \overset{\text{BNF}}{\to} x$ and $[e_2]z \overset{\text{BNF}}{\to} y$. The parse tree $[e_1e_2]xz = xy$ is built from them using (*seq1.p*).
– $e_1|e_2$ with $[e_1|e_2]w \overset{\text{BNF}}{\to} x$ derived from $[e_1]w \overset{\text{BNF}}{\to} x$ according to (*choice1*).
As assumed, there exists parse tree $[e_1]w = x$ for $[e_1]w \overset{\text{BNF}}{\to} x$. The parse tree $[e_1|e_2]w = x$ is constructed from it using (*choice1.p*).
– $e_1|e_2$ with $[e_1|e_2]w \overset{\text{BNF}}{\to} x$ derived from $[e_2]w \overset{\text{BNF}}{\to} x$ according to (*choice2*).
As assumed, there exists parse tree $[e_2]w = x$ for $[e_2]w \overset{\text{BNF}}{\to} x$. $[e_2]w \overset{\text{BNF}}{\to} x$ means $w \in \mathcal{L}(e_2)\texttt{Tail}(e)$. As specified in Figure 4, parser calls the procedure $[e_1]$ on z. According to Proposition 1, the call returns either `fail` or prefix y of w. By Proposition 2, this latter would mean $w \in \mathcal{L}(e_1)\Sigma^*$, which contradicts (2). Therefore must be $[e_1]y = \texttt{fail}$. The parse tree $[e_1|e_2]w = x$ is constructed from $[e_1]w = \texttt{fail}$ and $[e_2]w = x$ using (*choice2.p*).

□

Lemma 5. *If there exist parse tree for each seed and each leaf of recursion sequence $e_{[k]}, \ldots, e_{[2]}, e_{[1]}$, there exists parse tree for each result in the sequence and in particular for $e = e_{[k]}$.*

Proof. Suppose the grower is called to handle e for input w. We start by showing that after the n-th round of the grower, $\langle e_{[n]} \rangle$ contains the root of parse tree for the subtree $[e_{[n]}]w \overset{\text{BNF}}{\rightarrow} x_{[n]}$.

As the grower checks all plants on each round, it will call $\langle e_{[n]} \rangle$ on round n. The proof is by induction on n.

(Induction base:) Expression $e_{[1]}$ is an exit $e_1|e_2$ of the sequence, with e_2 as seed. The result $[e_{[1]}]w \overset{\text{BNF}}{\rightarrow} x$ is derived from $[e_2]w \overset{\text{BNF}}{\rightarrow} x$ using (*choice2*).
When the grower calls $[e_{[1]}, z]$ in its first round, $[e_1, z]$ contains `fail`. As e_2 is the seed of the sequence, there exists parse tree with root $[e_2]w = x$. The parse tree for $\langle e_{[1]}, w \rangle = x$ is constructed from it using (*choice3.g*).

(Induction step:) Consider the round $n+1$ and assume that $[e_{[n]}, w]$ contains the root $\langle e_{[n]}, w \rangle = x$ of parse tree for subtree $[e_{[n]}]w \overset{\text{BNF}}{\rightarrow} x$. What happens when the grower calls $[e_{[n+1]}, w]$ depends on expression $e_{[n+1]}$. It can be one of these:

- $A = e'$ with $[e_{[n+1]}]w \overset{\text{BNF}}{\rightarrow} x$ derived from $[e']w \overset{\text{BNF}}{\rightarrow} x$ according to (*rule*).
  The e' here is $e_{[n]}$ with parse tree $\langle e', w \rangle = x$. The parse tree for $\langle e_{[n+1]}, w \rangle$ is constructed from it using (*rule.g*).
- $e_1 e_2$ with $[e_{[n+1]}]xz \overset{\text{BNF}}{\rightarrow} xy$ derived from $[e_1]xz \overset{\text{BNF}}{\rightarrow} x$ and $[e_2]z \overset{\text{BNF}}{\rightarrow} y$ according to (*seq*).
  The e_1 here is $e_{[n]}$ with parse tree $\langle e_1, xz \rangle = x$. As e_2 is a leaf of the sequence, there exists parse tree with root $[e_2]z = y$. The parse tree for $[e_{[n+1]}, w]$ is constructed using (*seq1.g*).
- $e_1|e_2$. By (1), $e_2 \notin \mathbb{C}$. The BNF result $[e_{[n+1]}]w \overset{\text{BNF}}{\rightarrow} x$ is derived from or $[e_2]w \overset{\text{BNF}}{\rightarrow} x$ according to (*choice2*).
  As $e_{[n]}$ is in $\mathbb{C}$, it must be e_1 with parse tree $\langle e_1, w \rangle = x$. The parse tree for $\langle e_{[n+1]}, w \rangle$ is constructed from it using (*choice1.g*).

Note that from (3) follows $\varepsilon \notin \texttt{Adv}(e)$. Thus, according to Lemma 1, a new result is stored in $\langle e_{[n]}, w \rangle$ in every turn.

Suppose the grower does not stop after $e_{[k]}$ because it finds a better result for plant $[e_{[k]}, w]$. If this better result is x', we have by Lemma 1 $x' = \langle e_{[k]}, w \rangle \texttt{Adv}(e_{[k]})$ which means $\texttt{Adv}(e_{[k]})$ is a prefix of $\texttt{Tail}_R(e)$, and contradicts (3). Thus, the grower stops at $e_{[k]}$.

The parse tree $[e]w = x$ is constructed from $\langle e_{[k]}, w \rangle = x$ using (*grow.p*). □

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles,Techniques, and Tools. Addison-Wesley (1987)
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, 1–34 (1973)
4. Brooker, P., Morris, D.: A general translation program for phrase structure languages. J. ACM 9(1), 1–10 (1962)
5. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004. pp. 111–122. ACM, Venice, Italy (14–16 January 2004)
6. Hill, O.: Support for Left-Recursive PEGs (2010), https://github.com/orlandohill/peg-left-recursion
7. Hopgood, F.R.A.: Compiling Techniques. MacDonalds (1969)
8. Laurent, N., Mens, K.: Parsing expression grammars made practical. CoRR abs/1509.02439 (2015), http://arxiv.org/abs/1509.02439
9. Mascarenhas, F., Medeiros, S., Ierusalimschy, R.: On the relation between context-free grammars and Parsing Expression Grammars. Science of Computer Programming 89, 235–250 (2014)
10. McClure, R.M.: TMG – a syntax directed compiler. In: Winner, L. (ed.) Proceedings of the 20th ACM National Conference. pp. 262–274. 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 (Aug 2010)
12. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: Left recursion in Parsing Expression Grammars. Science of Computer Programming 96, 177–190 (2014)
13. Redziejowski, R.R.: From EBNF to PEG. Fundamenta Informaticae 128, 177–191 (2013)
14. Redziejowski, R.R.: Trying to understand PEG. Fundamenta Informaticae 157, 463–475 (2018)
15. Sigaud, P.: Left recursion. Tech. rep. (2017), https://github.com/PhilippeSigaud/Pegged/wiki/Left-Recursion
16. Tratt, L.: Direct left-recursive parsing expression grammars. Tech. Rep. EIS-10-01, School of Engineering and Information Sciences, Middlesex University (Oct 2010)
17. Warth, A., Douglass, J.R., Millstein, T.D.: Packrat parsers can support left recursion. In: 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. pp. 103–110 (2008)