Some Aspects of Parsing Expression Grammar*

Roman R. Redziejowski

Abstract

Parsing Expression Grammar (PEG) is a new way to specify syntax, by means of a top-down process with limited backtracking. It can be directly transcribed into a recursive-descent parser. The parser does not require a separate lexer, and backtracking removes the usual $LL(1)$ constraint. This is convenient for many applications, but there are two problems: PEG is not well understood as a language specification tool, and backtracking may result in exponential processing time. The paper consists of two parts that address these problems. The first part is an attempt to find out the language actually defined by a given parsing expression. The second part reports measurements of backtracking activity in a PEG-derived parser for the programming language C.

1 Introduction

Parsing Expression Grammar (PEG) is a new way to specify syntax, recently introduced by Ford [6–8]. The grammar is a formal description of a recursive-descent parser with limited backtracking.

Recursive-descent parsers have been around for a while. Already in 1961, Lucas [13] suggested the use of recursive procedures that reflect the syntax of the language being parsed. His design did not allow backtracking; an explicit assumption about the syntax (section 1.13211) was identical to what later became known as $LL(1)$. The great advantage of recursive-descent parsers is transparency: the code closely reflects the grammar, which makes it easy to maintain and modify. However, manipulating the grammar to force it into the $LL(1)$ mold can make the grammar itself unreadable. The use of backtracking removes the $LL(1)$ restriction. Complete backtracking, meaning an exhaustive search of all alternatives, may require an exponential time. A reasonable compromise is limited backtracking, also called "fast-back" in [12]. In that approach, we discard further alternatives once a sub-goal has been recognized.

Limited backtracking was adopted in at least two of the early top-down designs: the Atlas Compiler Compiler of Brooker and Morris [5, 20], and TMG (the TransMoGrifier) of McClure [15]. The syntax specification used in TMG was later formalized and analyzed by Birman and Ullman [3, 4]. It appears in [2] as "Top-Down Parsing Language" (TDPL) and "Generalized TDPL" (GTDPL). Parsing Expression Grammar is a development of this latter.

Parsing Expression Grammar is designed for a unified syntax definition, that does not require a separate "lexer" or "scanner". Together with the lifting of the $LL(1)$ restriction, this gives a very convenient tool when we need an ad-hoc parser for some application.

One problem with PEG is just its new approach to specify syntax. Although PEG looks very much like the Extended Backus-Naur Form (EBNF), it is *not* EBNF. It is an algorithm, and defines whatever that algorithm happens to accept. Writing PEG for a language specified in EBNF offers many surprises and is basically a trial-and-error process. It has to be better understood. After a brief introduction to PEG in Section 2, we try, in Section 3, to find out what is actually defined by a given parsing expression.

Another problem with PEG is just the backtracking. Even the limited backtracking may require a lot of time. In [6, 7], PEG was introduced together with a technique called *packrat parsing*. Packrat parsing handles backtracking by extensive *memoization*: storing all results of parsing procedures. It guarantees linear parsing time at a large memory cost[1].

*Appeared in *Fundamenta Informaticae 85*, 1-4 (2008) 441–454.

[1] "Packrat" comes from *pack rat* – a small rodent (*Neotoma cinerea*) known for hoarding unnecessary items; also a person that does the same. "Memoization", introduced in [16], is the technique of reusing stored results of function calls instead of recomputing them.

Excessive backtracking does not matter in small interactive applications such as [18], where the input is short and performance not critical. But, the author had a feeling that the usual programming languages do not require much backtracking. The feeling was based on the observation that these languages have large $LL(1)$ parts, and that limited backtracking prevents the parser from going back farther than to the beginning of a statement. An experiment reported in [19] indeed demonstrated a moderate backtracking activity in a PEG parser for Java 1.5. Section 4 reports similar results for the programming language C.

2 Parsing Expression Grammar

Parsing Expression Grammar is a set of named *parsing expressions.* They are specified by rules of the form $A = e$ where e is a parsing expression and A is the name given to it. Parsing expressions are instructions for parsing strings. When applied to a character string, parsing expression tries to match initial portion of that string, and may ”consume” the matched portion. It may then indicate ”success” or ”failure”.

Figure 1 lists all forms of parsing expressions. Each of $e, e_1, \ldots, e_n$ in the Figure is a parsing expression, specified either explicitly or by its name. Subexpressions may be enclosed in parentheses to indicate the order of applying the operators. In the absence of parentheses, the operators appearing lower in the table have precedence over those appearing higher. Note the backtracking involved in three constructions: the sequence and the two predicates.

$e_1/\ldots/e_n$	Ordered choice: Apply expressions $e_1, \ldots, e_n$, in this order, to the text ahead, until one of them succeeds and possibly consumes some text. Indicate success if one of expressions succeeded. Otherwise do not consume any text and indicate failure.
$e_1 \ldots e_n$	Sequence: Apply expressions $e_1, \ldots, e_n$, in this order, to consume consecutive portions of the text ahead, as long as they succeed. Indicate success if all succeeded. Otherwise do not consume any text and indicate failure.
$\&e$	And predicate: Indicate success if expression e matches the text ahead; otherwise indicate failure. Do not consume any text.
$!e$	Not predicate: Indicate failure if expression e matches the text ahead; otherwise indicate success. Do not consume any text.
e^+	One or more: Apply expression e repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any) and indicate success if there was at least one match. Otherwise indicate failure.
e^*	Zero or more: Apply expression e repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any). Always indicate success.
$e?$	Zero or one: If expression e matches the text ahead, consume it. Always indicate success.
$[\,s\,]$	Character class: If the character ahead appears in the string s, consume it and indicate success. Otherwise indicate failure.
$[\,c_1\text{-}c_2\,]$	Character range: If the character ahead is one from the range c_1 through c_2, consume it and indicate success. Otherwise indicate failure.
”s”	String: If the text ahead is the string s, consume it and indicate success. Otherwise indicate failure.
_	Any character: If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

Fig.1.

Parsing Expressions:

```
sum     = number (sign number)* !_
number  = real / integer
real    = digits? "." digits space?
integer = digits space?
sign    = [+-] space?
digits  = [0-9]+
space   = [ ]*
```

Parsing Expressions translated to Java procedures:

```
boolean sum() {
  init();
  if (!number()) return reject();
  while (sum_1());
  if (next()) return reject();
  return accept(); }

boolean sum_1() { // sum_1 = sign number
  init();
  if (!sign()) return reject();
  if (!number()) return reject();
  return accept(); }

boolean number() {
  init();
  if (real()) return accept();
  if (integer()) return accept();
  return reject(); }

boolean real() {
  init();
  digits();
  if (!next('.')) return reject();
  if (!digits()) return reject();
  space();
  return accept(); }
```

```
boolean integer() {
  init();
  if (!digits()) return reject();
  space();
  return accept(); }

boolean sign() {
  init();
  if (!nextIn("+-")) return reject();
  space();
  return accept(); }

boolean digits() {
  init();
  if (!nextIn('0','9')) return reject();
  while (nextIn('0','9'));
  return accept(); }

boolean space() {
  init();
  while (next(' '));
  return accept(); }
```

Fig.2.

Figure 2 shows, highest up, a sample grammar in PEG formalism. It defines syntax for writing simple sums of numbers, with white space around, but not within, the numbers. Note that ”`!_`” means ”no character ahead”, that is, ”end of input”. The rest of the Figure shows the expressions translated to Java procedures. The procedures use a number of common methods. Method `init()` stacks an object representing a call to the procedure; the object contains current input position. Method `accept()` unstacks the call object and returns `true`; `reject()` resets input position from the call object, unstacks the call object and returns `false`. Methods `next(...)` and `nextIn(...)` look at the next character and either consume it returning `true`, or return `false`; `next()` without argument consumes the next character and returns `true`, or returns `false` at the end of input. Together, the procedures constitute a simple recursive-descent parser.

The procedures in Figure 2 do most of their job via side-effects of `if` statements, what is normally not considered the best programming practice. However, here it results in a very concise code that closely corresponds to the grammar. Notice that subexpressions of the form $e?$ and e^* can be inlined as unconditional calls and `while` statements. All this can be produced by a relatively simple parser generator.

In the following, a recursive-descent parser obtained by a similar transcription of PEG into executable procedures is referred to as a ”PEG parser”.

As explained in [19], the common methods `init()`, `accept()`, etc. can be modified to provide some memoization. They can be further extended to assist keeping of semantic information in call objects. All this can be done behind the scenes, with only minimal changes to the code shown in Figure 2.

3 What is actually defined by PEG?

The construction of parser in Figure 2 is quite appealing: no need to factor out `digits` from `real` and `integer`, as little backtracking will do the job. A quick step from the grammar to the parser, and no separate lexer to write (provided one has a ready package with common methods). The parser will always terminate; as shown in [8], this is guaranteed if the grammar does not contain left recursion and the expressions under * and $^+$ always consume some input. One can easily see which strings are accepted (in the sense of `sum()` returning `true`).

But, this latter is not always so easy. PEG may look like EBNF, but its meaning is quite different. For an illustration, consider the definitions `("a"|"aa")"a"` and `("aa"|"a")"a"` in EBNF. They both specify the language $\{aa, aaa\}$. However, the parsing expression `("a"/"aa")"a"` does not accept aaa: `"a"/"aa"` consumes the first a, letting the final `"a"` consume the second, and leaving out the third a. No attempt is ever made to go back and let `"a"/"aa"` try the second alternative. The expression `("aa"/"a")"a"` does not accept aa: `"aa"/"a"` accepts all of aa, leaving nothing for the final `"a"`. Again, the second alternative of `"aa"/"a"` is not tried.

An example that really defies intuition is the recursive expression `A = "a"A"a"/"aa"`. When applied to a string of eight a's, it consumes the entire string; when applied to a string of nine a's, it consumes only the first two. In general, the number $A(n)$ of a's consumed by `A` when applied to a string of n a's can be obtained by starting with $A(2) = 2$ and computing

$$A(n) = \begin{cases} A(n-1)+2 & \text{if } A(n-1)+2 \le n\,, \\ 2 & \text{otherwise,} \end{cases}$$

for $n = 3, 4$, etc.. One can verify that $A(n) = n$, that is, `A` consumes the whole string, if and only if n is a power of 2.[2]

If one wants to define a language using PEG, one must be able to see what is actually being defined. The effects of ordered choice and limited backtracking, such as illustrated above, may be hidden by many levels of rules. In [21], Schmitz suggested how one can detect the situations exemplified by `"a"/"aa"`. This corresponds, in his own words, to detection of dead code. The example of `("aa"/"a")"a"` shows that the reverse, `"a"/"aa"`, is also suspect. However, this situation is more subtle; it can still be safe and useful, as in the case of `number` in Figure 2.

We shall try another approach: finding the language accepted by a given expression. But, we must first decide what do we mean by it. In [8], Ford defines the language accepted by an expression as the set of all inputs on which the expression succeeds – without necessarily consuming all of the input. That means the language accepted by each of `"aa"/"a"` and `"a"/"aa"` consists of all strings starting with a, and the language accepted by the expression `A` above of all strings starting with aa. This definition seems too coarse if we want to define semantics; we need to know exactly the strings accepted by different elements of the grammar.

In the following, we denote the input alphabet by Σ. For $x, y \in \Sigma^*$, we write $x \le y$ to mean that x is a prefix of y. For $X \subseteq \Sigma^*$, $\mathrm{Pref}(X)$ denotes the set of all prefixes of $x \in X$. The empty word is denoted by ε.

For a parsing expression e and $w \in \Sigma^*$, we write $c\,(e,\, w) = x$ to mean that e succeeds on input w and consumes x. We write $c\,(e,\, w) = \varphi$ to mean that e fails on w.
We consider two versions of a language accepted by expression e, "large" and "small":

$$\mathcal{L}(e) = \{x \in \Sigma^* \mid c\,(e,\, xy) = x \text{ for some } y \in \Sigma^*\}\,,$$
$$\mathcal{S}(e) = \{x \in \Sigma^* \mid c\,(e,\, xy) = x \text{ for all } y \in \Sigma^*\}\,.$$

One can easily see that $\mathcal{S}(e) \subseteq \mathcal{L}(e)$.[3] We note that the equality $\mathcal{S}(e) = \mathcal{L}(e)$ indicates a reliable behavior of e, independent of the text far ahead.

[2]This example appears as exercise 4.13 in [1] and in a number of earlier publications, originating with Theorem 6.2 in [3].
[3]The language $\mathcal{L}$ seems to be the same as $\mathcal{L}$ in Section 4 of [21].

Lemma 1. *For any parsing expression e and $x, y \in \Sigma^*$ holds:*
(1) $c\,(e,\,x) \neq \varphi \;\Rightarrow\; x \in \mathcal{L}(e)\Sigma^*\,,$
(2) $c\,(e,\,xy) \neq \varphi \;\Rightarrow\; x \in \mathcal{L}(e)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e))\,,$
(3) $x \in \mathcal{S}(e)\Sigma^* \;\Rightarrow\; c\,(e,\,xy) \neq \varphi\,.$

Proof. Let e be any parsing expression.
(1) Consider any x such that $c\,(e,\,x) \neq \varphi$, that is, $c\,(e,\,x) = u$ where $x = uv$ for some $v \in \Sigma^*$. By definition, $u \in \mathcal{L}(e)$, so $x \in \mathcal{L}(e)\Sigma^*$.
(2) Consider any x, y such that $c\,(e,\,xy) \neq \varphi$. According to (1), this implies $xy \in \mathcal{L}(e)\Sigma^*$. That means $xy = uv$ where $u \in \mathcal{L}(e)$ and $v \in \Sigma^*$. By Levi's lemma, exists $t \in \Sigma^*$ such that either (a) $x = ut$ and $v = ty$, or (b) $u = xt$ and $y = tv$. In case (a) we have $x \in \mathcal{L}(e)\Sigma^*$, so $xy \in \mathcal{L}(e)\Sigma^*\Sigma^* = \mathcal{L}(e)\Sigma^*$. In case (b) we have $xt \in \mathcal{L}(e)$, that is, $x \in \mathrm{Pref}(\mathcal{L}(e))$.
(3) Consider any $x \in \mathcal{S}(e)\Sigma^*$, that is, $x = uv$ for some $u \in \mathcal{S}(e)$ and $v \in \Sigma^*$. By definition, $u \in \mathcal{S}(e)$ means $c\,(e,\,uvy) = u$ for all v and y. Thus, $c\,(e,\,xy) \neq \varphi$. $\square$

One can easily see that for each of the expressions `[`s`]`, `[`c_1`-`c_2`]`, `"`s`"`, and `_` , the languages $\mathcal{S}(e)$ and $\mathcal{L}(e)$ are identical, and consist of the characters, or the string, defined by the expression. We proceed now to languages defined by other expressions.

Proposition 1. *For any expressions e_1, e_2 holds*

$$\begin{aligned}&\mathcal{S}(e_1) \;\cup\; (\mathcal{S}(e_2) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1))) \;\subseteq\; \mathcal{S}(e_1/e_2)\\ &\subseteq\; \mathcal{L}(e_1/e_2) \;\subseteq\; \mathcal{L}(e_1) \;\cup\; (\mathcal{L}(e_2) - \mathcal{S}(e_1)\Sigma^*)\,.\end{aligned}$$

Proof. Consider any $x \in \mathcal{S}(e_1) \cup (\mathcal{S}(e_2) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1)))$ and $y \in \Sigma^*$. Apply e_1/e_2 to xy. If $x \in \mathcal{S}(e_1)$, e_1 succeeds and consumes x.
If $x \in \mathcal{S}(e_2) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1))$ we have, from Lemma 1 (2), $c\,(e_1,\,xy) = \varphi$, that is, e_1 fails and e_2 is applied. As $x \in \mathcal{S}(e_2)$, e_2 succeeds and consumes x.
In each case, e_1/e_2 applied to xy for arbitrary y succeeds and consumes x. This shows that $x \in \mathcal{S}(e_1/e_2)$ and proves the first inclusion.
The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}(e_1/e_2)$ That means $c\,(e_1/e_2,\,xy) = x$ for some $y \in \Sigma^*$. Either $c\,(e_1,\,xy) = x$, or $c\,(e_1,\,xy) = \varphi$ and $c\,(e_2,\,xy) = x$. Clearly, $x \in \mathcal{L}(e_1)$ in the first case, and $x \in \mathcal{L}(e_2)$ in the second. But $c\,(e_1,\,xy) = \varphi$ in the second case means, according to Lemma 1 (3), that $x \notin \mathcal{S}(e_1)\Sigma^*$. This proves the third inclusion. $\square$

Proposition 2. *For any expressions e_1, e_2 holds*

$$\mathcal{S}(e_1)\mathcal{S}(e_2) \;\subseteq\; \mathcal{S}(e_1e_2) \;\subseteq\; \mathcal{L}(e_1e_2) \;\subseteq\; \mathcal{L}(e_1)\mathcal{L}(e_2)\,.$$

Proof. Consider any $x \in \mathcal{S}(e_1)\mathcal{S}(e_2)$ and $y \in \Sigma^*$. We have $x = uv$ where $u \in \mathcal{S}(e_1)$ and $v \in \mathcal{S}(e_2)$. Apply e_1e_2 to xy. As $u \in \mathcal{S}(e_1)$, e_1 applied to uvy succeeds and consumes u, after which e_2 is applied to vy. As $v \in \mathcal{S}(e_2)$, e_2 applied to vy succeeds and consumes v. Thus, e_1e_2 applied to xy for arbitrary y succeeds and consumes $uv = x$. This shows that $x \in \mathcal{S}(e_1e_2)$ and proves the first inclusion. The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}(e_1e_2)$. That means e_1 and e_2 applied one after another to some $w \in \Sigma^*$ consume, respectively, u and v such that $uv = x$. We have, clearly, $u \in \mathcal{L}(e_1)$ and $v \in \mathcal{L}(e_2)$, so $x \in \mathcal{L}(e_1)\mathcal{L}(e_2)$. This proves the third inclusion. $\square$

Proposition 3. *For any expression e holds*

$$\begin{aligned}\varnothing \;\subseteq\; \mathcal{S}(\&e) \;\subseteq\; \mathcal{L}(\&e) \;\subseteq\; \{\varepsilon\}\,,\\ \varnothing \;\subseteq\; \mathcal{S}(!e) \;\subseteq\; \mathcal{L}(!e) \;\subseteq\; \{\varepsilon\}\,.\end{aligned}$$

Proof. The predicates either consume ε or fail. $\square$

Proposition 4. *For any expression e holds*

$$\mathcal{S}(e) \subseteq \mathcal{S}(e?) \subseteq \mathcal{L}(e?) \subseteq \mathcal{L}(e) \cup \{\varepsilon\}\,.$$

Proof. Consider any $x \in \mathcal{S}(e)$ and any $y \in \Sigma^*$. Apply e to xy. As $x \in \mathcal{S}(e)$, e succeeds and consumes x, meaning $e?$ succeeds and consumes x, so $x \in \mathcal{S}(e?)$. This proves the first inclusion. The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}(e?)$. It is either in $\mathcal{L}(e)$ (if e succeeds and consumes x on some input) or ε (if e fails on some input). This proves the third inclusion. □

Proposition 5. *For any expression e holds*

$$(\mathcal{S}(e))^* \subseteq \mathcal{L}(e^*) \subseteq (\mathcal{L}(e))^*\,.$$

Proof. Consider any $x \in (\mathcal{S}(e))^*$. If $x = \varepsilon$, it is in $\mathcal{L}(e^*)$. Otherwise $x = u_1u_2\dots u_n$ where $n \geq 1$ and $u_i \in \mathcal{S}(e)$ for $1 \leq i \leq n$. Apply e to $u_1u_2\dots u_n$. As $u_1 \in \mathcal{S}(e)$, e succeeds and consumes u_1; e is then applied to $u_2\dots u_n$. As $u_2 \in \mathcal{S}(e)$, e succeeds and consumes u_2, until the n-th application of e consumes u_n. Thus $x \in \mathcal{L}(e^*)$ also in this case. This proves the first inclusion.
(Note that this can not be applied to xy for any y: if y starts with any string in $\mathcal{S}(e)$, e^* will continue and consume more than x. We cannot conclude that $(\mathcal{S}(e))^* \subseteq \mathcal{S}(e^*)$.)
Consider now any $x \in \mathcal{L}(e^*)$. If $x = \varepsilon$, it is in $\mathcal{L}(e^*)$. Otherwise, e applied $n \geq 1$ times to some string w consumes x. The i-th application for $1 \leq i \leq n$ consumes $u_i \in \mathcal{L}(e)$, Clearly $x = u_1u_2\dots u_n \subseteq \mathcal{L}(e^*)$. This proves the second inclusion. □

The result stated by Proposition 3 is not very useful for finding the effect of predicates. Combined with Proposition 2, it does not give much information about constructions such as $\&e_1e_2$ or $e_1\&e_2$. The reason is that $\mathcal{S}(e)$ and $\mathcal{L}(e)$ do not depend on the input ahead. One can obtain better estimates by considering a predicate together with expression that follows.

Proposition 6. *For any expressions e_1, e_2 holds*

$$\mathcal{S}(e_2) \cap \mathcal{S}(e_1)\Sigma^* \subseteq \mathcal{S}(\&\,e_1e_2) \subseteq \mathcal{L}(\&\,e_1e_2) \subseteq \qquad \mathcal{L}(e_2) \cap (\mathcal{L}(e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1)))\,.$$

Proof. Consider any $x \in \mathcal{S}(e_2) \cap \mathcal{S}(e_1)\Sigma^*$ and $y \in \Sigma^*$. Apply $\&\,e_1e_2$ to xy. As $x \in \mathcal{S}(e_1)\Sigma^*$, we have $xy \in \mathcal{S}(e_1)\Sigma^*\Sigma^* = \mathcal{S}(e_1)\Sigma^*$. From $xy \in \mathcal{S}(e_1)\Sigma^*$ follows, by Lemma 1(3), that e_1 succeeds on xy, so e_2 is applied to xy. As $x \in \mathcal{S}(e_2)$, e_2 succeeds and consumes x. Thus, $\&\,e_1e_2$ applied to xy for arbitrary y succeeds and consumes x. This proves the first inclusion. The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}(\&\,e_1e_2)$. That means e_2 and e_1 applied to the same $w \in \Sigma^*$ both succeed and consume, respectively, x and some u. We have thus $x \in \mathcal{L}(e_2)$ and $u \in \mathcal{L}(e_1)$. Both x and u are prefixes of w. We have either $x \in u\Sigma^*$ or $x \in \mathrm{Pref}(u)$, meaning respectively, that $x \in \mathcal{L}(e_1)\Sigma^*$ or $x \in \mathrm{Pref}(\mathcal{L}(e_1))$. Hence, $x \in \mathcal{L}(e_2) \cap (\mathcal{L}(e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1)))$. This proves the third inclusion. □

Proposition 7. *For any expressions e_1, e_2 holds*

$$\mathcal{S}(e_2) - (\mathcal{L}(e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1))) \subseteq \mathcal{S}(!\,e_1e_2) \subseteq \mathcal{L}(!\,e_1e_2) \subseteq \mathcal{L}(e_2) - \mathcal{S}(e_1)\Sigma^*\,.$$

Proof. Consider any $x \in \mathcal{S}(e_2) - (\mathcal{L}(e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1)))$ and $y \in \Sigma^*$. Apply $!\,e_1e_2$ to xy. From $x \notin (e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1))$ follows, by Lemma 1 (2), that e_1 fails on xy, so e_2 is applied to xy. As $x \in \mathcal{S}(e_2)$, e_2 succeeds and consumes x. Thus, $!\,e_1e_2$ applied to xy for arbitrary y succeeds and consumes x. This proves the first inclusion. The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}(!\,e_1e_2)$. That means e_2 applied to some $w \in \Sigma^*$ consumes its prefix x, so $x \in \mathcal{L}(e_2)$. But this happens only after e_1 failed when applied to the same w. We have then, by Lemma 1 (3), $w \notin \mathcal{S}(e_1)\Sigma^*$. Suppose $x \in \mathcal{S}(e_1)\Sigma^*$. But $w = xu$ for some $u \in \Sigma^*$, which would mean $w \in \mathcal{S}(e_1)\Sigma^*$. Hence, $x \notin \mathcal{S}(e_1)\Sigma^*$. This proves the third inclusion. □

Including the expression ahead improves other results as well.

Proposition 8. *For any expression e_1, e_2, e_3 holds*

$$\begin{aligned}
&\mathcal{S}(e_1)\mathcal{S}(e_3) \;\cup\; (\mathcal{S}(e_2)\mathcal{S}(e_3) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1))) \;\subseteq\; \mathcal{S}((e_1/e_2)\, e_3)\\
&\quad\subseteq\; \mathcal{L}((e_1/e_2)\, e_3) \;\subseteq\; \mathcal{L}(e_1)\mathcal{L}(e_3) \;\cup\; (\mathcal{L}(e_2)\mathcal{L}(e_3) - \mathcal{S}(e_1)\Sigma^*)\,.
\end{aligned}$$

Proof. Consider any $x \in \mathcal{S}(e_1)\mathcal{S}(e_3) \cup (\mathcal{S}(e_2)\mathcal{S}(e_3) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1)))$. and any $y \in \Sigma^*$. Apply $(e_1/e_2)\, e_3$ to xy. Two cases are possible:

(a) $x \in \mathcal{S}(e_1)\mathcal{S}(e_3)$. From Proposition 1 follows $\mathcal{S}(e_1) \subseteq \mathcal{S}(e_1/e_2)$, and from Proposition 2 follows $\mathcal{S}(e_1/e_2)\mathcal{S}(e_3) \subseteq \mathcal{S}((e_1/e_2)\, e_3)$, so $x \in \mathcal{S}((e_1/e_2)\, e_3)$.

(b) $x \in \mathcal{S}(e_2)\mathcal{S}(e_3) - \mathcal{L}(e_1)\Sigma^* - \mathrm{Pref}(\mathcal{L}(e_1))$. Take any $y \in \Sigma^*$ and apply $(e_1/e_2)\, e_3$ to xy. From $x \notin \mathcal{L}(e_1)\Sigma^* \cup \mathrm{Pref}(\mathcal{L}(e_1))$ follows, by Lemma 1 (2), that e_1 fails when applied to xy, so e_2 is applied. We have $x = uv$ where $u \in \mathcal{S}(e_2)$ and $v \in \mathcal{S}(e_3)$. That means $c\,(e_2,\, uvy) = u$ and $c\,(e_3,\, vy) = v$. Thus, $(e_1/e_2)\, e_3$ applied to xy for arbitrary y consumes x.

This proves the first inclusion. The second inclusion follows from definitions.
Consider now any $x \in \mathcal{L}((e_1/e_2)\, e_3)$. That means $c\,((e_1/e_2)\, e_3,\, xy) = x$ for some y. Two cases are possible:

(a) $c\,(e_1,\, xy) = u$ for some u, where $x = uv$ for some v, and $c\,(e_3,\, vy) = v$. We have $u \in \mathcal{L}(e_1), v \in \mathcal{L}(e_3)$ so $x \in \mathcal{L}(e_1)\mathcal{L}(e_3)$.

(b) $c\,(e_1,\, xy) = \varphi$, $c\,(e_2,\, xy) = u$ for some u, where $x = uv$ for some v, and $c\,(e_3,\, vy) = v$. We have $u \in \mathcal{L}(e_2), v \in \mathcal{L}(e_3)$ so $x \in \mathcal{L}(e_2)\mathcal{L}(e_3)$. But, according to Lemma 1 (3), we have then also $x \notin \mathcal{S}(e_1)\Sigma^*$.

This proves the third inclusion. □

Note that Proposition 8 gives an exact result $\mathcal{S}(e) = \mathcal{L}(e) = \{aaa\}$ for $e =$ `("aa"/"a")"a"`, while Proposition 1 combined with Proposition 2 gives only the estimate $\{aaa\} \subseteq \mathcal{S}(e) \subseteq \mathcal{L}(e) \subseteq \{aa, aaa\}$. Unfortunately, no similar results seem possible for expressions of the form $e_1\&e_2$ and $e_1!e_2$.

The last three Propositions show that we do not always get the best results by just combining those for subexpressions operation by operation. They may be improved by considering more complex structures as a whole. But how far one needs to go? Finding the languages for recursive expressions is still another story. As the inclusions in Proposition 1 involve both $\mathcal{L}$ and $\mathcal{S}$, we would have to solve systems of "equations" (or rather "inequalities") in two "unknowns". All of it seems rather hopeless.

To close this section, let us look at the relation of $\mathcal{L}$ and $\mathcal{S}$ to two other possible "languages of e":

$$\begin{aligned}
\mathcal{F}(e) &= \{x \in \Sigma^* \mid c\,(e,\, x) \neq \varphi\} \;\; \text{(the language of } e \text{ according to Ford [8])}\,,\\
\mathcal{C}(e) &= \{x \in \Sigma^* \mid c\,(e,\, x) = x\} \;\; \text{(the input strings completely consumed by } e)\,.
\end{aligned}$$

One can easily see that $\mathcal{S} \subseteq \mathcal{C} \subseteq \mathcal{L}$. For the expression `A` discussed earlier, we have $\mathcal{S}(\mathtt{A}) = \varnothing$, $\mathcal{C}(\mathtt{A}) = \{a^{2^n} \mid n \geq 1\}$, and $\mathcal{L}(\mathtt{A}) = \{a^{2n} \mid n \geq 1\}$, showing that both inclusions may be proper. One can also easily see that $\mathcal{S}(e)\Sigma^* \subseteq \mathcal{F}(e) \subseteq \mathcal{L}(e)\Sigma^*$, and that both inclusions may be proper. Finally, for an expression $e!\_$, required to consume all of its input, we have:

$$\varnothing = \mathcal{S}(e!\_) \subseteq \mathcal{F}(e!\_) = \mathcal{C}(e) = \mathcal{L}(e!\_)\,.$$

4 Backtracking in a PEG parser for C

In an earlier paper [19], the author reported an experiment to measure the effects of backtracking in a PEG parser for Java 1.5. The experiment consisted of expressing the Java syntax in PEG formalism, transcribing it into a parser (accidentally, also in Java), applying it to a large number of source files, and collecting various statistics. The parsing time was found to be linear over a large range of file sizes. The number of procedure calls repeated at the same position because of backtracking was only 16 % of all procedure calls. A very mild memoization (remembering the results of two most recent calls to each procedure) reduced the number of such redundant calls to about 1 %.

This successful experiment raised a question whether the same would apply to other programming languages. To find this out, the author performed another experiment, with a PEG parser for C. The syntax used was that defined by the International Standard ISO/IEC 9899:TC2, Annex A, with left recursion replaced by iteration. The preprocessing facilities defined under A.3 were not included, so the experiment was carried out on preprocessed files. The available source files contained some features not specified by the standard, namely function specifier `_stdcall`, type qualifier `__declspec()`, and type specifier `__attribute__()`. They had to be added to the syntax.

A problem with parsing C is its `typedef` declaration that dynamically introduces new keywords. Recognizing these keywords is essential for parsing. What makes the things worse is that identifiers defined by `typedef` do not become fully reserved, and may be, in some contexts, used as ordinary names. In order to handle `typedef`, it was necessary to introduce some semantic processing into the parser.

In the absence of any other method to verify the PEG definition, the resulting parser was applied to a number of source files and problems fixed until all files were accepted. The encountered problems were mostly the same as reported in [19].

file	size kB	calls	calls/byte
1	0.3	15 600	59.3
2	0.5	18 334	38.6
3	0.8	32 171	41.5
4	1.7	58 485	33.2
5	2.7	144 165	53.1
6	5.4	271 817	49.4
7	8.6	519 303	58.8
8	15.1	1 016 279	65.5
9	26.7	1 195 305	43.8
10	46.1	2 716 447	57.6
11	81.8	4 586 314	54.8
12	141.9	9 317 937	64.1

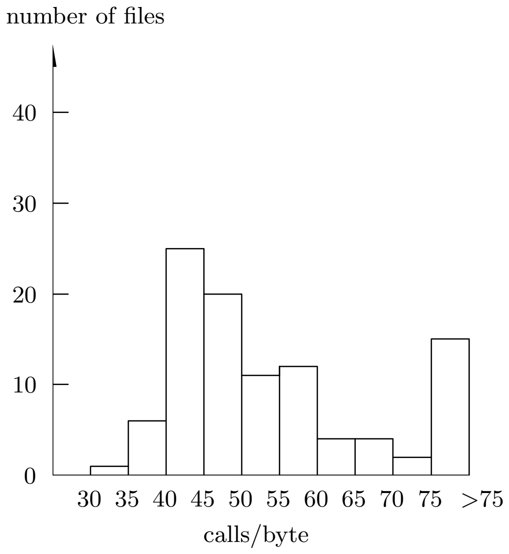

Fig.3.

To measure the effects of backtracking, the parser was applied to 100 preprocessed files with sizes from 0.3 to 265 kB, taken from [9, 10, 14, 17]. The processing time was estimated by counting calls to parsing procedures. This measure was used instead of the clock time because it is repeatable and independent of the environment. The number of calls divided by file size was used to asses the dependence between file size and processing time. Figure 3 shows the values obtained for twelve selected files. The values of calls/byte for all 100 files varied between 32.2 and 215.5, with 49.4 as the median. Their distribution is shown by the histogram in Figure 3. As one can see, there is no apparent correlation with file size – a reasonable approximation to linear time. (The actual correlation factor for all 100 files was 0.21.)

The number of redundant calls (a procedure called again at the same input position) was 68.5 % of all calls. The experiment was subsequently repeated with memoization of one, and then two, most recent results of each procedure. Figure 4 shows the effect of memoization on the number of redundant calls. (The numbers are totals for all 100 files.)

memo	calls	redundant	%
0	265 008 314	181 497 529	68.5
1	125 891 830	42 381 045	33.7
2	106 634 119	23 123 334	21.7

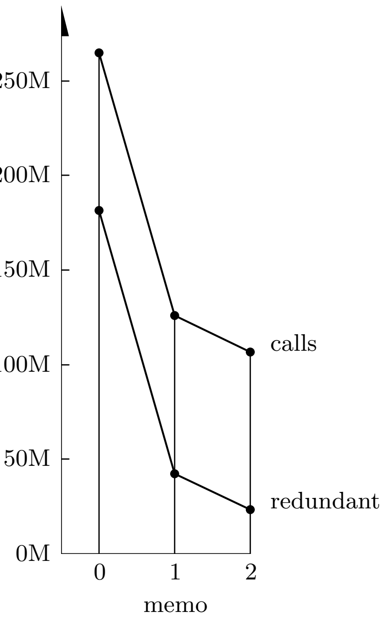

Fig.4.

These results are less optimistic than those obtained for Java. The parsing "time" is still linear in file size, but is much higher, and redundant calls are not eliminated by memoization. One way to explain this difference was that Java experiment used a simplified grammar from Chapter 18 of Java Language Specification [11]. As stated there, the grammar "in many cases minimizes the necessary look ahead".

To see how much such simplifications matter, the C experiment was repeated with a slightly modified grammar. The modification consisted of reversing the order of FunctionDefinition and Declaration in the rule for ExternalDeclaration, and relaxing the syntax of left-hand side of AssignmentExpression. Figures 5 and 6 show the results obtained with the modified syntax. They are very close to those for Java in [19]. The values of calls/byte ranged from 17.5 to 71.8, with 24.1 as the median. The correlation with file size was 0.03, and redundant calls were practically eliminated by memoizing the two most recent results.

file	size kB	calls	calls/byte
1	0.3	9 663	36.7
2	0.5	11 207	23.6
3	0.8	17 438	22.5
4	1.7	31 072	17.6
5	2.7	72 639	26.8
6	5.4	130 426	23.7
7	8.6	239 709	27.1
8	15.1	415 030	26.8
9	26.7	575 391	21.1
10	46.1	1 013 214	21.5
11	81.8	2 023 736	24.2
12	141.9	4 073 702	28.0

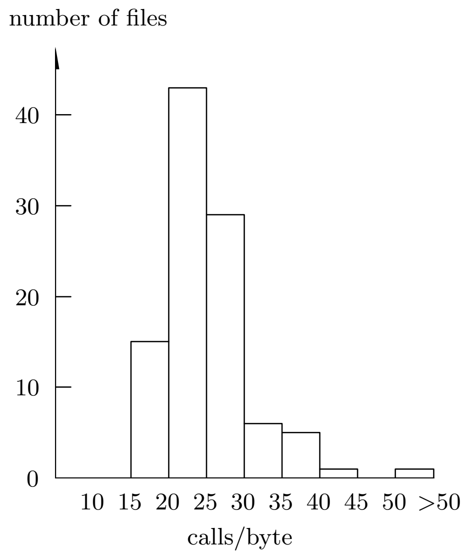

Fig.5.

memo	calls	redundant	%
0	103 150 973	23 899 850	23.2
1	81 465 462	2 214 339	2.7
2	80 333 011	1 081 888	1.3

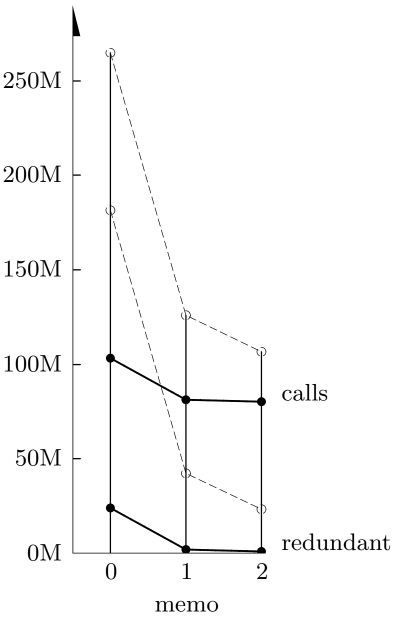

Fig.6.

5 Summary

In Section 3, we tried to find out the language that is actually defined by a given parsing expression. We used two different definitions of such language, but were at best able to obtain some estimates. The results are complicated, and involve operations that may be computationally impossible, such as obtaining prefix set of a language, or intersection of two languages.

It follows that PEG *is not good* as a language specification tool. The most basic property of a specification is that one can clearly see what it specifies. And this is, unfortunately, not true for PEG. On the other hand, PEG is useful as a parser. Instead of trying to find the meaning of given parsing expressions, one should ask how to construct a correct, comprehensible, and efficient PEG parser for a language specified in a traditional way. It has been shown (Theorem 6.1 in [2], Theorem 3.1 in [3]) that each deterministic context-free language can be defined in TDPL, and thus in PEG. However, the construction shown there is via a push-down automaton, and does not seem likely to result in a readable grammar. On the other hand, the Java and C grammars constructed ad-hoc for the experiments described in Section 4 are very close to the EBNF grammars used in the specifications. Perhaps the right way is to identify a discipline that would restrict the use of PEG to well-understood constructions. This is, after all, how we once saved ourselves from the quagmire of "spaghetti programming". And one should remember that EBNF is not ideal either.

In Section 4, we checked if the encouraging results obtained with a PEG parser for Java apply also to C. The answer is yes, with a modification. For the C grammar taken directly from the ISO standard, performance was worse than for Java, but not disastrous. The processing time was still linear in the source size. The number of redundant calls due to backtracking was rather high, but dropped by more than a half with memoizing of one latest result. However, it was still high with memoizing of two results.

The experiment was repeated with a slightly modified grammar. The results were then almost identical to those for Java. Which is not surprising, as the Java grammar used in [19] was, in fact, optimized for top-down processing.

One may note that the modification was rather simple, and did not affect readability of the grammar. As much can be gained by such manipulations, the question is how to identify them? The ones made in the experiment just followed a simple intuition. But, is it possible to do it in some systematic way, other than running tests and collecting statistics?

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles,Techniques, and Tools.* Addison-Wesley, 1987.

[2] Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation and Compiling, Vol. I, Parsing.* Prentice Hall, 1972.

[3] Birman, A. *The TMG Recognition Schema.* PhD thesis, Princeton University, February 1970.

[4] Birman, A., and Ullman, J. D. Parsing algorithms with backtrack. *Information and Control 23* (1973), 1–34.

[5] Brooker, P., and Morris, D. Some proposals for the realization of a certain assembly program. *The Computer Journal 3*, 4 (1961), 220–231.

[6] Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[7] Ford, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming* (October 2002).

[8] Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy, 14–16 January 2004), N. D. Jones and X. Leroy, Eds., ACM, pp. 111–122.

[9] GNU. Bison parser generator. `http://www.gnu.org/software/bison`.

[10] GNU. M4 macro processor. `http://www.gnu.org/software/m4`.

[11] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, The 3rd Edition.* Addison-Wesley, 2005.

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

[13] Lucas, P. The structure of formula-translators. *ALGOL Bulletin Supplement 16* (September 1961), 1–27.

[14] Mariano, A. GNU units 1.86. `http://ftp.gnu.org/gnu/units`.

[15] McClure, R. M. TMG – a syntax directed compiler. In *Proceedings of the 20th ACM National Conference* (24–26 August 1965), L. Winner, Ed., ACM, pp. 262–274.

[16] Michie, D. Memo functions and machine learning. *Nature 218* (April 1968), 19–22.

[17] Noll, L. C. Calc: C-style arbitrary precision system. `http://sourceforge.net/projects/calc`.

[18] Redziejowski, R. R. Computing with units – user's manual. `http://units-in-java.sourceforge.net/UnitsDoc.htm`.

[19] Redziejowski, R. R. Parsing Expression Grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae 79*, 3–4 (2007), 513–524.

[20] Rosen, S. A compiler-building system developed by Brooker and Morris. *Commun. ACM 7*, 7 (July 1964), 403–414.

[21] Schmitz, S. Modular syntax demands verification. Tech. Rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, Oct. 2006.