Applying Classical Concepts to Parsing Expression Grammar*

Roman R. Redziejowski

Abstract

The paper is an attempt to see how much we can learn about a given Parsing Expression Grammar with the help of classical concepts used in the construction of predictive top-down parsers.

1 Introduction

This is a sequel to the author's papers [4, 5] on Parsing Expression Grammars. Parsing Expression Grammar (PEG), as introduced by Ford [2, 3], is a way to define a recursive-descent parser with limited backtracking. The parser does not require a separate "lexer" to preprocess the input, and the limited backtracking lifts the $LL(1)$ restriction usually imposed by top-down parsers. These properties are useful in many applications. However, PEG is not well understood as a language specification tool. One of the things attempted in [5] was to find the language defined by a given parsing expression. Unfortunately, it was only possible to obtain some approximations that became more and more difficult as the complexity of the expression increased.

In this paper, we take some classical concepts used in the construction of predictive top-down parsers, adapt them to the world of PEGs, and try to see how much we can learn with their help.

2 Parsing Expression Grammar

We recall that Parsing Expression Grammar is a set of *parsing expressions*, 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, written in a special language. In the following, we view parsing expression as a procedure that carries out such instruction. The expressions can call each other recursively, thus forming together a recursive-descent parser.

In general, parsing expression is applied to a character string – the "input" to be parsed – at a position indicated by some "cursor". It tries to recognize a portion of the string ahead of the cursor. If it succeeds, it "consumes" the recognized portion by advancing the cursor and indicates "success"; otherwise, it indicates "failure" and does not consume anything (does not advance the cursor). The following four kinds of parsing expressions work directly on the input string:

`"`s`"`	where s is a nonempty character string. If the text ahead starts with the string s, consume that string and indicate success. Otherwise indicate failure.
`[`s`]`	where s is a nonempty character string. If the text ahead starts with a character appearing in s, consume that character and indicate success. Otherwise indicate failure.
`[`c_1`-`c_2`]`	where c_1, c_2 are two characters. If the text ahead starts with a character from the range c_1 through c_2, consume that character and indicate success. Otherwise indicate failure.
`_`	(underscore character). If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

*Appeared in *Fundamenta Informaticae 93*, 1-3 (2009) 325–336.

These expressions are analogous to terminals of a classical context-free grammar, and are in the following referred to as "terminals". The remaining five kinds of parsing expressions call other expressions to do their job:

e_1/e_2	(*Choice*) Call expression e_1. Indicate success if it succeeded. Otherwise call expression e_2 and indicate success if e_2 succeeded or failure if it failed.
$e_1\, e_2$	(*Sequence*) Call expression e_1. If it succeeded, call expression e_2, and indicate success if e_2 succeeded. If e_1 or e_2 failed, reset cursor as it was before the invocation of e_1 and indicate failure.
$!e$	(*Lookahead*) Call expression e and then reset the cursor as it was before the invocation of e. Indicate success if e failed or failure if e succeeded.
$e*$	(*Iteration*) Call expression e repeatedly as long as it succeeds. Indicate success (even if e did not succeed a single time).
$e?$	(*Optional*) Call expression e and indicate success (even if e failed).

The expressions e, e_1, e_2 may be specified either explicitly or by name. Parentheses may be used to indicate the order of applying the operators. In the absence of parentheses, operators appearing lower in the above list have precedence. (The expressions shown here are simpler than those in [4, 5], but the missing forms can be constructed from them.)

Backtracking takes place in the Sequence expression. If e_1 succeeds and consumes some input, and then e_2 fails, the cursor is moved back to where it was before trying e_1. If this Sequence was called as the first alternative in a Choice expression, Choice has an opportunity to try another alternative on the same input. However, the opportunity to try another alternative is limited: once e_1 in the Choice e_1/e_2 succeeded, e_2 will never be tried on the same input, even if the parse fails later on.

3 Some notation

In the following, we consider a fixed grammar. The set of all its expressions is denoted by $\mathcal{G}$. All subexpressions are treated as separate members of $\mathcal{G}$. When we speak of "an expression", we always mean a member of $\mathcal{G}$. Some of the expressions must be terminals; their set is denoted by $\mathcal{T} \subseteq \mathcal{G}$.

Let Σ be the input alphabet. For $x, y \in \Sigma^*$, x is a *prefix* of y if $y = xu$ for some $u \in \Sigma^*$; y is a *tail* of x if $x = uy$ for some $u \in \Sigma^*$. We write $x \leq y$ 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 tail of $w \in \Sigma^*$ starting at position p is denoted by $w[\,p\,]$. The empty word is denoted by ε. If an expression succeeds without consuming any string, we say that it consumed ε.

For an expression e and $w \in \Sigma^*$, we write $c\,(e,\, w) = x$ to mean that e succeeds when called at the start of w and consumes x. We write $c\,(e,\, w) = \varphi$ to mean that e fails when called at the start of w. The language of expression e is defined as

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

Note that $x \in \mathcal{L}(e)$ does not necessarily mean $c\,(e,\, xy) = x$ for an arbitrary string y. However, $x \in \mathcal{L}(e) \;\Rightarrow\; c\,(e,\, xy) = x$ for all $e \in \mathcal{T}$ and $y \in \Sigma^*$.

A relation R on $\mathcal{G}$ is a subset of $\mathcal{G} \times \mathcal{G}$. As usual, we write $R(e)$ to mean the set of all e' such that $(e, e') \in R$, and $R(\mathcal{E})$ to mean the union of $R(e)$ for all $e \in \mathcal{E} \subseteq \mathcal{G}$. The identity relation is denoted by I. The transitive closure of R is denoted by R^+, the transitive and reflexive closure by R^*, and the product of relations R and S by $R \times S$.

4 The parsing process

We consider a parsing process that begins by calling some "starting expression" S with cursor at the beginning of an input string w.

We say that expression e was "called at position p" to mean that it was called with cursor at position p in w. If e succeeds, it may move the cursor to some position $q \geq p$. We say then that e "succeeded

at q". (We have $q = p$ if e consumed empty word.) If e fails, we say that e "failed at p". We say that e "terminated at p" to mean that e either succeeded at p or failed at p. If the expression that terminated is S, the parsing process ends. Otherwise the expression "returns to" its caller.

We are interested in movements of the cursor as the result of executing the expressions. Note that even if e terminates with unchanged cursor, the cursor could have changed in the process of evaluating e. For example: e_1 in $e = e_1\, e_2$ consumed a non-null string, after which e_2 failed. We include this possibility when we say that e moved the cursor. A failing terminal never moves the cursor.

5 Properties of expressions

The classical construction of predictive top-down parsers for BNF grammars uses two properties of grammar symbols: FIRST – the set of possible first letters in a string derived by the symbol, and FOLLOW – the set of letters that may follow a string derived by the symbol. (See, for example, [1] Section 4.4, [7] Section 6.2.) We are going to define analogous properties for parsing expressions.

5.1 Nullability, fallibility, and well-formedness

We shall need three properties of expressions defined by Ford in [3]. We call them here *nul* (nullable), *adv* (may advance), and *fail* (may fail). We treat them as subsets of $\mathcal{G}$ that contain expressions with the respective property. These sets are defined by means of recursive formulas and can be computed by iteration. The following is shown to hold for all $e \in \mathcal{G}$:

$$\text{If } c\,(e,\, w) = \varepsilon \text{ for some } w \in \Sigma^* \text{ then } e \in \mathit{nul}\,. \tag{1}$$

$$\text{If } c\,(e,\, w) = x \in \Sigma^+ \text{ for some } w \in \Sigma^* \text{ then } e \in \mathit{adv}\,. \tag{2}$$

$$\text{If } c\,(e,\, w) = \varphi \text{ for some } w \in \Sigma^* \text{ then } e \in \mathit{fail}\,. \tag{3}$$

Note that these implications can not, in general, be reversed.

The evaluation of an expression e may never terminate, because of infinite recursion or iteration. Such e may have none of the above properties, but may have some of them (if it terminates for some input strings).

Termination is handled in [3] by another property of an expression, denoted by WF (well-formed). This property is again defined by means of recursive formulas and can be evaluated by iteration. It is shown that if all expressions are well-formed, the evaluation of every expression always terminates. The grammar is then said to be well-formed. A well-formed grammar is not left-recursive and does not contain any expression e^* with $e \in \mathit{nul}$.

In the following, we assume $\mathcal{G}$ to be well-formed.

5.2 Relation FIRST

A non-terminal expression E processes input by calling its subexpressions as specified in Section 2. A subexpression e thus called is said to be "directly called" by E. If e then calls its subexpression e_1, this latter is "indirectly called" by E; the same applies to a subexpression e_2 called by e_1, etc..

Define the relation First on $\mathcal{G}$ to consist of all pairs (E, e) such that one of the following holds:

- $E = e/e_1$ for some e_1 ;
- $E = e_1/e$ for some $e_1 \in \mathit{fail}$;
- $E = e\, e_1$ for some e_1 ;
- $E = e_1 e$ for some $e_1 \in \mathit{nul}$;
- $E = !\, e$;
- $E = e^*$;
- $E = e?$.

One can easily verify:

Proposition 1. *If expression E called at position p directly calls e at the same position, then $e \in \mathrm{First}(E)$.*

By induction on the number of intermediate calls, we have:

Proposition 2. *If expression E called at position p calls e at the same position, either directly or indirectly, then $e \in \mathrm{First}^+(e)$.*

Define $\mathrm{FIRST} = \mathrm{First}^* \cap \mathcal{T}$.

Proposition 3. *If expression e called at p moved the cursor, there exists $t \in \mathrm{FIRST}(e)$ and $u \in \mathcal{L}(t)$ such that $u \leq w[\,p\,]$.*

Proof. Moving the cursor means that some non-null string $x \leq w[\,p\,]$ was consumed during evaluation of e. A prefix $u \leq x$ of that string must have been consumed by some terminal $t \in \mathcal{T}$ called at p. That t is either e itself or was called by e, directly or indirectly. By Proposition 2, we have $t \in \{e\} \cup \mathrm{First}^+(e)$, that is, $t \in \mathrm{First}^*(e)$. Obviously $u \in \mathcal{L}(t)$ and $u \leq w[\,p\,]$. □

5.3 Disjoint expressions

Define terminals t_1 and t_2 to be *disjoint*, denoted by $t_1 \asymp t_2$, if for every $x \in \mathcal{L}(t_1)$ and $y \in \mathcal{L}(t_2)$ neither $x \leq y$ nor $y \leq x$.

We have, for example,`"ab"` $\asymp$ `"ac"`, but `"ab"` $\not\asymp$ `[a-z]` and `"ac"` $\not\asymp$ `[a-z]`. Obviously, $t_1 \not\asymp t_2$ whenever $\mathcal{L}(t_1) \cap \mathcal{L}(t_2) \neq \varnothing$.

Define expressions e_1 and e_2 to be disjoint, written $e_1 \asymp e_2$, if $t_1 \asymp t_2$ for every $t_1 \in \mathrm{FIRST}(e_1)$ and $t_2 \in \mathrm{FIRST}(e_2)$.

Define expression e to be disjoint with a set $\mathcal{E}$ of expressions, written $e \asymp \mathcal{E}$, if $e \asymp e_1$ for each $e_1 \in \mathcal{E}$.

Proposition 4. *If each of two expressions independently called at the same position moves the cursor, these expressions are not disjoint.*

Proof. Suppose e_1 called at p moved the cursor. According to Proposition 3 exist $t_1 \in \mathrm{FIRST}(e_1)$ and $u \in \mathcal{L}(t_1)$ such that $u \leq w[\,p\,]$. Suppose e_2 called at p also moved the cursor. According to Proposition 3 exist $t_2 \in \mathrm{FIRST}(e_2)$ and $v \in \mathcal{L}(t_2)$ such that $v \leq w[\,p\,]$. From $u \leq w[\,p\,]$ and $v \leq w[\,p\,]$ follows $u \leq v$ or $v \leq u$, so $t_1 \not\asymp t_2$, and thus $e_1 \not\asymp e_2$. □

5.4 Relations FOLLOW

Consider the situation when an expression terminated and returns to its caller. The caller may then terminate without moving the cursor, or backtrack, or call another expression. The possible scenarios are detailed in the Appendix. There may be a chain of nested expressions that return to their callers without moving the cursor.

Define the following four relations on $\mathcal{G}$:

- Relation Last_{ss} consisting of all pairs (e, E) such that one of the following holds:
  - $E = e/e_1$ for some e_1 ;
  - $E = e_1/e$ for some $e_1 \in \mathit{fail}$;
  - $E = e\,e_1$ for some $e_1 \in \mathit{nul}$;
  - $E = e_1\,e$ for some e_1 ;
  - $E = e^*$;
  - $E = e?$.
- Relation Last_{sf} consisting of all pairs (e, E) such that one of the following holds:
  - $E = e\,e_1$ for some $e_1 \in \mathit{fail}$, and $e \in \mathit{nul}$;
  - $E = !\,e$, and $e \in \mathit{nul}$.
- Relation Last_{fs} consisting of all pairs (e, E) where $e \in \mathit{fail}$ and one of the following holds:
  - $E = e/e_1$ for some $e_1 \in \mathit{nul}$;
  - $E = !\,e$;
  - $E = e^*$;
  - $E = e?$.

- Relation Last_{ff} consisting of all pairs (e, E) where $e \in \mathit{fail}$ and one of the following holds:
  - $E = e/e_1$ for some $e_1 \in \mathit{fail}$;
  - $E = e_1/e$ for some $e_1 \in \mathit{fail}$;
  - $E = e\, e_1$;
  - $E = e_1\, e$ for some $e_1 \in \mathit{nul}$.

By inspection of scenarios in the Appendix, one can verify:

Proposition 5. *Whenever e returned to its caller E and E terminated without moving the cursor, then:*

- *If both e and E succeeded, $E \in \mathrm{Last}_{ss}(e)$;*
- *If e succeeded and E failed, $E \in \mathrm{Last}_{sf}(e)$;*
- *If e failed and E succeeded, $E \in \mathrm{Last}_{fs}(e)$;*
- *If both e and E failed, $E \in \mathrm{Last}_{ff}(e)$.*

These four relations can be summarized by one relation Last on the set of pairs (e, o) where $e \in \mathcal{G}$ and $o \in \{\texttt{succ},\texttt{fail}\}$. Namely, define:

- $((e, \texttt{succ}), (E, \texttt{succ})) \in \mathrm{Last}$ iff $(e, E) \in \mathrm{Last}_{ss}$,
- $((e, \texttt{succ}), (E, \texttt{fail})) \in \mathrm{Last}$ iff $(e, E) \in \mathrm{Last}_{sf}$, etc..

After taking the transitive closure of Last, we can recover four relations, LAST_{ss}, LAST_{sf}, LAST_{fs}, and LAST_{ff} having this property:

Proposition 6. *Whenever a return from e resulted in a termination of E without moving the cursor, either immediately or via a chain of returns that did not move the cursor, then:*

- *If both e and E succeeded, $E \in \mathrm{LAST}_{ss}(e)$;*
- *If e succeeded and E failed, $E \in \mathrm{LAST}_{sf}(e)$;*
- *If e failed and E succeeded, $E \in \mathrm{LAST}_{fs}(e)$;*
- *If both e and E failed, $E \in \mathrm{LAST}_{ff}(e)$.*

Define now the relation Next_s on $\mathcal{G}$ to consist of all pairs (e, e_1) such that one of the following holds:

- $E = e\, e_1$ for some $E \in \mathcal{G}$,
- $E = e^*$ for some $E \in \mathcal{G}$ and $e = e_1$,

and the relation Next_f consisting of all pairs (e, e_1) such that:

- $E = e/e_1$ for some $E \in \mathcal{G}$ and $e \in \mathit{fail}$.

By inspection of scenarios in the Appendix, one can verify:

Proposition 7. *Whenever e returned to its caller E after terminating at position p and E called an expression e_1 that moved the cursor, then:*

- *If e succeeded, $e_1 \in \mathrm{Next}_s(e)$;*
- *If e failed, $e_1 \in \mathrm{Next}_f(e)$.*

Finally, define:

$$\mathrm{FOLLOW}_s = (\mathrm{LAST}_{ss} \cup \mathrm{I}) \times \mathrm{Next}_s \cup \mathrm{LAST}_{sf} \times \mathrm{Next}_f;$$
$$\mathrm{FOLLOW}_f = (\mathrm{LAST}_{ff} \cup \mathrm{I}) \times \mathrm{Next}_f \cup \mathrm{LAST}_{fs} \times \mathrm{Next}_s\,.$$

Proposition 8. *After an expression e terminated at p, one of the following must occur after one or more returns that do not move the cursor:*

(a) The parser terminates at p.
(b) The parser backtracks from p.
(c) An expression e_1 called at p moves the cursor. Then, if e succeeded, $e_1 \in \mathrm{FOLLOW}_s(e)$; if e failed, $e_1 \in \mathrm{FOLLOW}_f(e)$.

Proof. Clearly, one of (a)–(c) must eventually happen. It remains to check the conditions stated for (c). Consider first the case where e succeeded. Suppose e returned to its caller E and E called e_1. Then, by Proposition 7, $e_1 \in \mathrm{Next}_s(e)$, so $e_1 \in (\mathrm{I} \times \mathrm{Next}_s)(e)$. Suppose now e returned to its caller E_1 which returned to its caller E_2 without moving cursor, which, etc., etc., returned to E_n without moving cursor,

which returned to E_{n+1} without moving cursor, after which E_{n+1} called e_1. If E_n was successful, we have, by Proposition 6, $E_n \in \mathrm{LAST}_{ss}(e)$ and, by Proposition 7, $e_1 \in \mathrm{Next}_s(E_n)$, so $e_1 \in (\mathrm{LAST}_{ss} \times \mathrm{Next}_s)(e)$. Similarly, if E_n failed, we have $e_1 \in (\mathrm{LAST}_{sf} \times \mathrm{Next}_f)(e)$. In each case $e_1 \in \mathrm{FOLLOW}_s(e)$. The proof for failed e is analogous. □

6 Using the properties

6.1 Safe choice

Define the expression $E = e_1/e_2$ to be *safe* if it satisfies these four conditions:

$$e_1 \notin \mathit{nul}\,; \tag{4}$$

$$e_1 \asymp e_2\,; \tag{5}$$

$$e_2 \in \mathit{nul} \;\Rightarrow\; e_1 \asymp \mathrm{FOLLOW}_s(E)\,; \tag{6}$$

$$e_2 \in \mathit{fail} \;\Rightarrow\; e_1 \asymp \mathrm{FOLLOW}_f(E)\,. \tag{7}$$

Proposition 9. *Suppose a safe expression $E = e_1/e_2$ is called at p. If e_1 failed after moving the cursor, the parser will not move the cursor before it backtracks from p or terminates at p.*

Proof. Let E be called as stated. Suppose e_1 failed after advancing from p. Expression e_2 is now called at p. According to (5) and Proposition 4, e_2 will not advance, meaning it either consumes ε or fails. If e_2 consumes ε, E succeeds at p. If e_2 fails, E fails at p. According to Proposition 8, one of the following must now occur after one or more returns that do not move the cursor:
(a) The parser terminates at p.
(b) The parser backtracks from p.
(c) An expression e called at p moves the cursor.
If E succeeded, $e \in \mathrm{FOLLOW}_s(e)$. By (1) we have $e_2 \in \mathit{nul}$, so this is excluded by (6) and Proposition 4. If E failed, $e_1 \in \mathrm{FOLLOW}_f(e)$. By (3) we have $e_2 \in \mathit{fail}$, so this is excluded by (7) and Proposition 4. Thus, only (a) and (b) are possible. □

In other words, the property of being safe ensures that once e_1 failed and backtracked, the input rejected by it will not be reprocessed until the parser backtracks from the start of E. The only possible actions on the way are failing tests by terminals.

A phenomenon that causes much difficulty in understanding a PEG is what was in [4] loosely called "prefix capture", but could be better described as "language hiding". It is the case where a greedy first expression in e_1/e_2 prevents e_2 from being applied to inputs that it could otherwise accept (e_2 is not tried once e_1 succeeded). As a result, $\mathcal{L}(e_1/e_2)$ is not, as one could expect, $\mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, but has some, or all, of $\mathcal{L}(e_2)$ missing. Classical examples are `"a"/"aa"` where the second alternative will never succeed, and `"aa"/"a"` where the second alternative is not exercised on input beginning with aa. Note that hiding in this latter case depends on the input ahead. Constructions like this may be intentional; for example, defining comparison operator as `"=<"/"="` ensures that the first `"="` in `"=<"` will never be recognized separately. However, the effect may be unexpected and difficult to detect in a complex grammar. Some warning is thus desirable.

Let us define "language hiding" more precisely as the situation where e_1 in $E = e_1/e_2$ succeeds on some input but parsing fails later on; applying e_2 at the same position would result in a successful parse that would consume at least some input.

Proposition 10. *Language hiding does not occur in a safe $E = e_1/e_2$.*

Proof. Suppose a safe expression $E = e_1/e_2$ is called at input position p. That means expression e_1 is called at position p. Suppose it succeeds; according to (4) and (1), it consumes a nonempty word, thus moving the cursor. Suppose e_2 is applied instead at the same position. According to Proposition 9, the parser will not advance until it either terminates at p or backtracks from p. □

Safety is a very strong condition that results in many false warnings. Schmitz formulated in [6] a weaker, but still not necessary, condition for absence of language hiding. It is called "general semi-disjointness" and can be somewhat informally expressed as

$$\mathcal{L}(e_1) \cap \mathrm{Pref}(\mathcal{L}(e_2)\mathcal{L}(\mathrm{follow}(E))) = \varnothing\,, \tag{8}$$

where follow(E) is any sequence of expressions that can be called after E. Unfortunately, its mechanical verification is not simple.

6.2 Safe iteration

Another case of language hiding is by a greedy star operator. For example, `"ab"` in the expression `"a"* ("ab"/"c")` will never succeed. We can define this case as the situation where e^* succeeds after calling e for $n > 1$ times and the subsequent parse fails; however, the subsequent parse succeeds and consumes some input if e is called only $m < n$ times.

Define expression $E = e^*$ to be *safe* if:

$$e \asymp \mathrm{FOLLOW}_s(E)\,. \tag{9}$$

Proposition 11. *Language hiding does not occur in a safe $E = e^*$.*

Proof. Suppose $E = e^*$ succeeded after $n > 1$ calls to e. Suppose now that E instead returned success at p after calling e only $m < n$ times. According to Proposition 8, one of the following must now occur after one or more returns that do not move the cursor:
(a) The parser terminates at p.
(b) The parser backtracks from p.
(c) An expression $e_1 \in \mathrm{FOLLOW}_s(E)$ called at p moves the cursor.
Because $\mathcal{G}$ is well-formed, we have $e \notin nul$; according to (1), the $(m+1)$-th call to e consumed $x \neq \varepsilon$, thus moving the cursor. The case (c) is thus excluded by (9) and Proposition 4. □

Again, one can formulate a weaker condition for the absence of language hiding:

$$(\mathcal{L}(e))^+ \cap \mathrm{Pref}(\mathcal{L}(\mathrm{follow}(E))) = \varnothing\,, \tag{10}$$

but its mechanical verification is not simple.

7 Experiments

To see in practice how much can be learned about a given PEG from the properties defined above, the author implemented a PEG analyzer to compute them. The relations FIRST and FOLLOW were easy to obtain using binary matrices.

Applied to a number of grammars, the analyzer clearly identified areas free from backtracking (in the sense of reprocessing the input). The resulting diagnostics indicated places that had to be further analyzed. Two typical cases are illustrated below.

The following is an example of PEG from [5], together with the message produced for it by the analyzer:

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

Conflicting alternatives in number: real / integer
  [0-9] in digits and [0-9] in digits
```

The message says that the expression for `number` is not safe: it does not satisfy (5) because both `real` and `integer` start with the terminal `[0-9]`, which is the first subexpression in `digits`. This is the only possibility of backtracking in the whole grammar. One can see by inspection that `number` satisfies (8): any string $x \in \mathcal{L}$(`real`) starts with a decimal point, or with digits followed by a decimal point. Any string $y \in \mathcal{L}$(`integer`)$\mathcal{L}$(follow(`number`))) starts with digits followed by a plus, minus, or semicolon, so x cannot be a prefix of y. No part of $\mathcal{L}$(`integer`) is thus hidden by `real`.

The second example is a grammar for simple polynomials with plus for sum and juxtaposition for product. As in the previous example, white space is allowed around the variable names and the plus.

```
poly  = space term (plus term)* ";"
term  = name name*
name  = [a-z][a-z]* space
plus  = "+" space
space = [ ]*

Conflict with follow in name: [a-z]*
  [a-z] in [a-z]* and [a-z] in name
```

The analyzer message says in this case that `[a-z]*` in `name` does not satisfy (9) and may gobble up `[a-z]` in something that follows `name`. By inspection one can see this "something" to be another `name` in `term`, because `space` may be an empty string. Indeed, if two `name`s are not separated by white space, the parser will read them as a single `name`. The question is if one should do anything about that. Probably nothing, because the parser will understand the text in the same way as a human reader.

The experiments revealed one serious problem, namely treatment of the lookahead operator. The problem is encountered on a large scale in grammars that define reserved words. The words used as identifiers must be distinct from reserved words. This can be specified in the grammar as shown below:

```
phrase = key / name
name   = !key [a-z][a-z]*
key    = ("plus"/"minus") ![a-z]
```

The lookahead `!key` prevents a `key` from being recognized as a `name`. The lookahead `![a-z]` prevents `key` from consuming a prefix of a longer word; otherwise `"plusTwo"` would not be recognized as a `name`. The analyzer produced these messages for this grammar:

```
Conflicting alternatives in phrase: key / name
  "plus" in key and [a-z] in name, "plus" in key
  "minus" in key and [a-z] in name, "minus" in key

Conflict with followf in key: "plus" / "minus"
  "plus" in key and [a-z] in name, "plus" in key
```

We have FIRST(`!key`) = {`"plus"`, `"minus"`} and FIRST(`![a-z]`) = {`[a-z]`}. Since `!key` is nullable, FIRST(`name`) contains `"plus"`, `"minus"`, and `[a-z]`. As `"plus"` $\not\asymp$ `[a-z]` and `"minus"` $\not\asymp$ `[a-z]`, the choice `key/name` violates (5), and the choice `"plus"/"minus"` violates (7). The analyzer listed these facts, but they are false warnings.

Running the analyzer for the entire Java grammar produced 227 messages, almost all of this kind. Any real diagnostics were drowned by them.

8 Conclusions

The classical properties like FIRST and FOLLOW can be redefined for PEG and are simple to obtain even for a large grammar. One difference is that instead of letters we have here terminal expressions, which may mean sets of letters, or strings. Another is a more complicated definition of FOLLOW.

We have used FIRST and FOLLOW to define conditions for choice and iteration that are analogous to the classical $LL(1)$ conditions, although they have a different form and meaning.

Checking these conditions produces useful information like the absence of reprocessing or language hiding. This helps to locate places that need further examination.

Unfortunately, most results obtained here have the form of implications that cannot, in general, be reversed. The properties FIRST and FOLLOW are kind of upper bounds, and conditions using them are sufficient, but not necessary. This results in false warnings. In particular, the lookahead operator "`!`" may trigger a whole avalanche of them. A proper handling of this operator is thus an obvious target for further work.

References

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

[2] Ford, B.: *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master Thesis, Massachusetts Institute of Technology, September 2002, `http://pdos.csail.mit.edu/papers/packrat-parsing:ford-ms.pdf`.

[3] 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.

[4] Redziejowski, R. R.: Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking, *Fundamenta Informaticae*, **79**(3–4), 2007, 513–524.

[5] Redziejowski, R. R.: Some Aspects of Parsing Expression Grammar, *Fundamenta Informaticae*, **85**(1–4), 2008, 441–454.

[6] Schmitz, S.: *Modular Syntax Demands Verification*, Technical Report I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, October 2006, `http://www.i3s.unice.fr/~mh/RR/2006/RR-06.32-S.SCHMITZ.pdf`.

[7] Tremblay, J.-P., Sorenson, P. G.: *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

Appendix

What happens after expression e terminated at p and returns to its caller E.
(*) indicates the cases when E terminates without moving the cursor from p.

Scenario 1. Expression e succeeded. Then:

If $E = e/e_1$, E succeeds at p. (*)
If $E = e_1/e$, E succeeds at p. (*)
If $E = e\ e_1$, E calls e_1 at p. These are possible outcomes:

- e_1 succeeds:
  - If e_1 moved the cursor, E succeeds at $q \geq p$.
  - If e_1 did not move the cursor, E succeeds at p. (*)
- e_1 fails:
  - If e moved the cursor, E fails at $q \leq p$ (backtracks if $q < p$).
  - If e did not move the cursor, but e_1 did, E fails at p.
  - If neither e nor e_1 moved the cursor, E fails at p. (*)

If $E = e_1\ e$, E succeeds at p. (*)
If $E = !e$, E fails:

- If e moved the cursor, E fails at $q \leq p$ (backtracks if $q < p$).
- If e did not move the cursor, E fails at p. (*)

If $E = e^*$, E calls e again. These are possible outcomes:

- The new e succeeds. E keeps calling e until e fails.
  Eventually E succeeds at $q > p$ (note: $e \notin nul$).
- The new e fails after moving the cursor. E succeeds at p.
- The new e fails without moving the cursor. E succeeds at p. (*)

If $E = e?$, E succeeds at p. (*)

Scenario 2. Expression e failed. Then:

If $E = e/e_1$, E calls e_1 at p. These are possible outcomes:

- e_1 succeeds.
  - If e_1 moved the cursor, E succeeds at $q \geq p$.
  - If e_1 did not move the cursor, E succeeds at p. (*)
- e_1 fails.
  - If e_1 moved the cursor, E fails at p.
  - If e_1 did not move the cursor, E fails at p. (*)

If $E = e_1/e$, E fails at p. (*)
If $E = e\ e_1$, E fails at p. (*)
If $E = e_1\ e$, E fails.

- If e moved the cursor, E fails at $q \leq p$ (backtracks if $q < p$).
- If e did not move the cursor, E fails at p. (*)

If $E = !e$, E succeeds at p. (*)
If $E = e^*$, E succeeds at p. (*)
If $E = e?$, E succeeds at p. (*)