Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking*

Roman R. Redziejowski

Abstract

Two recent developments in the field of formal languages are Parsing Expression Grammar (PEG) and packrat parsing. The PEG formalism is similar to BNF, but defines syntax in terms of recognizing strings, rather than constructing them. It is, in fact, precise specification of a backtracking recursive-descent parser. Packrat parsing is a general method to handle backtracking in recursive-descent parsers. It ensures linear working time, at a huge memory cost. This paper reports an experiment that consisted of defining the syntax of Java 1.5 in PEG formalism, and literally transcribing the PEG definitions into parsing procedures (accidentally, also in Java). The resulting primitive parser shows an acceptable behavior, indicating that packrat parsing might be an overkill for practical languages. The exercise with defining the Java syntax suggests that more work is needed on PEG as a language specification tool.

1 Introduction

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. It was suggested as early as in 1961 by Lucas [10], but for a long time the field was dominated by bottom-up parsers, their popularity enhanced by generators such as Yacc.

The structure of a recursive-descent parser follows closely a grammar defined in Backus-Naur Form (BNF) or Extended BNF (EBNF). Each procedure is associated with one symbol of the grammar and attempts to recognize an input string corresponding to that symbol. It either reports "success" and consumes the string, or reports "failure". A grammar in EBNF can be easily transcribed into a primitive recursive-descent parser, such as the Java program in Figure 1. Procedure `next(`c`)` in that program looks at the next input character. If that character is c, the procedure consumes it and returns `true`; otherwise it returns `false`. Procedures `letters()` and `digits()` accept respectively, a string of letters and a string of digits. Procedure `error()` throws an exception.

This is perhaps not the best coding style (work done by side-effects of "if" statements), but the great advantage of such a parser is its simplicity and clear relationship to the grammar. For smaller grammars, the parser can be easily produced and maintained by hand. For large grammars, parser generator is mainly a labor-saving device churning out procedures. This is contrary to bottom-up parsers, normally driven by large tables that have no obvious relationship to the grammar; these tables *must* be mechanically generated.

One can easily see that the procedure `sum()` in Figure 1 will terminate on any input (by either returning or throwing an exception), and will return `true` if and only if that input conforms to the grammar. In fact, a parser of this kind will always terminate if the grammar does not contain left recursion and the expression E in E^* or E^+ does not specify empty string. The top procedure returning `true` will always imply that input conforms to the grammar. The opposite, however, is not always true. The grammar may define alternative phrases that begin in the same way. Seeing correct initial portion, the parser may commit itself to a wrong alternative and throw an exception on finding incorrect continuation.

*Appeared in *Fundamenta Informaticae 79*, 3-4 (Sept 2007) 513–524.

```
Grammar:    sum  = term ("+" term)* ";"
            term = letters | digits


Parser:     boolean sum()                    // sum = term ("+" term)* ";"
              {
                if (!term())    return false;
                while (sum1());
                if (!next(';')  error();
                return true;
              }

            boolean sum1()                   // sum1 = "+" term
              {
                if (!next('+')) return false;
                if (!term())    error();
                return true;
              }

            boolean term()                   // term = letters | digits
              {
                if (letters())  return true;
                if (digits())   return true;
                return false;
              }
```

Fig.1.

The obvious solution is to move back in the input and explore other alternatives. This strategy, known as backtracking, was tried already in the early 1960's by Brooker and Morris [12]. The main problem with backtracking is that it may take time exponential in the length of input. Some solutions to that problem were suggested in the early 1970's [1–3], but were abandoned as exceeding the technology of that time. The attention was instead directed to grammars that do not require backtracking. The result was formulation of the notorious $LL(k)$ condition meaning that the parser can always make correct choice by looking at the next k input characters.

If the grammar is $LL(1)$, a primitive recursive-descent parser constructed from it will return `true` if and only if input conforms to the syntax. Unfortunately, most languages of practical interest do not have $LL(1)$ grammar. Sometimes, the problem can be handled by a preprocessor known as a "lexer" or "scanner". This means, at best, that we have to provide an additional processor, often using some ad-hoc technology. But at worst, it may not solve the problem, which happens to be the case for languages such as Java, C, and C++. The existing generators for top-down parsers, such as JavaCC, ANTLR [11], or Parse::RecDescent [5] extend EBNF with instructions to specify look-ahead. In some cases, the parser may be assisted by semantic procedures. There is also a report [4] on a generator that produces $LL(1)$ parsers from non-$LL(k)$ grammars by heavy transformations of the grammar. All these solutions go a long way from the simplicity and clarity of primitive parser.

An entirely new approach was recently taken by Ford [6, 7]. It is a continuation of ideas from [1–3]. The method developed by Ford guarantees linear parsing time – at a huge memory cost. The trick is to store the result of calling *each* parsing procedure at *each* position of the input, so you don't need to call it again after backtracking. The method is appropriately called the *packrat parsing*[1]. To the author's knowledge, there already exist two generators producing packrat parsers. One of them, called *Pappy*, is described in [6]. Another, called *Rats!*, is described in [9].

An integral component of Ford's approach is a new way to specify syntax. It is called *Parsing Expression Grammar* (PEG), and is a set of rules to parse input text. These rules can be easily transcribed into executable code. The result is a primitive parser similar to that in Figure 1, with the difference that it does some backtracking. The author had a feeling that such parser could be of practical use without a recourse to packrat technology. Existing programming languages have large parts that are $LL(1)$. Besides, a programming language is supposed to be understood by humans that can not do too much

[1] Merriam Webster Online Dictionary gives these definitions of "pack rat":
(1) a bushy-tailed rodent (Neotoma cinerea) of the Rocky Mountain area that has well-developed cheek pouches and hoards food and miscellaneous objects;
(2) one who collects or hoards especially unneeded items.

backtracking. In addition, Parsing Expression Grammar does only a limited backtracking; with language defined in the usual way as a sequence of statements, the grammar will never return farther back than the beginning of current statement. That means the parsing time may in the worst case be exponential in the length of a statement, times the number of statements.

This paper reports an experiment performed to verify that feeling. The experiment consisted of defining the grammar of Java 1.5 in PEG formalism, transcribing it into executable code, and applying the parser thus obtained to a number of Java source files.

The experiment confirmed the author's expectations. The parser exhibits a moderate backtracking activity. The parsing "time", expressed as the number of procedure calls, grows linearly with the input size. A significant improvement could be achieved by saving results of one or two most recent calls to each parsing procedure. (A thrifty mouse instead of a pack rat.) This modification could be comfortably hidden behind the scenes without destroying the clarity of the primitive parser.

The following Section 2 introduces the PEG formalism. Section 3 reports the practical experience of using that formalism to define Java 1.5 syntax. Sections 4 and 5 describe experiments with the parser.

$E_1/\ldots/E_n$	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.2.

2 Parsing Expression Grammar

Parsing Expression Grammar (PEG), as defined in [7], is a set of rules of the form $A = E$ where E is a *parsing expression* and A is a *grammar symbol* associated with 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 2 lists all forms of parsing expressions. Each of $E, E_1, \ldots, E_n$ is a parsing expression, specified either explicitly or via its symbol. 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.

The right-hand column of Figure 2 describes informally the actions of different expressions. Figure 3 defines these actions precisely in the form of Java procedures. The procedures act on a string of characters given as the array `input`, and try to match the portion of that string starting at position `pos`. They indicate success or failure by returning `true` or `false`, and "consume" the matched portion by advancing `pos`. Method `back(`p`)` resets `pos` to p and returns `false`. Procedures `next()` and `nextIn()` test next input character(s); only one is shown in detail.

```
// Ordered choice: A = E1/E2/.../En
   boolean A()
     {
       if (E1()) return true;
       if (E2()) return true;
          ...
       if (En()) return true;
       return false;
     }

// And predicate: A = &E
   boolean A()
     {
       int p = pos;
       if (!E()) return false;
       return !back(p);
     }

// One or more: A = E+
   boolean A()
     {
       if (!E()) return false;
       while (E());
       return true;
     }

// Zero or one: A = E?
   boolean A()
     {
       E();
       return true;
     }

// Character range [c1-c2]
   boolean nextIn(char c1, char c2)
     { ... }

// String "s"
   boolean next(String s)
     { ... }
```

```
// Sequence: A = E1E2 ... En
   boolean A()
     {
       int p = pos;
       if (!E1()) return false;
       if (!E2()) return back(p);
          ...
       if (!En()) return back(p);
       return true;
     }

// Not predicate: A = !E
   boolean A()
     {
     int p = pos;
       if (!E()) return true
       return back(p);      ;
     }

// Zero or more: A = E*
   boolean A()
     {
       while (E());
       return true;
     }

// Character class [s]
   boolean nextIn(String s)
     {
       if (pos>=endpos) return false;
       if (s.indexOf(input[pos])<0)
         return false;
       pos++;
       return true;
     }


// Any character
   boolean next()
     { ... }
```

Fig.3.

A parsing expression grammar with its rules transcribed into Java according to Figure 3 becomes a primitive recursive-descent parser. It can be viewed as the executable form of the grammar. From Theorem in section 3.6 of [7] follows that each procedure will terminate on any input if the grammar does not contain left recursion, and the expression E in E^* or E^+ does not specify empty string. (Such a grammar is referred to as "well-formed".) The language defined by the grammar is the set of input strings for which the top procedure returns `true`.

3 Writing the grammar

This and the following sections report experiments performed by the author. They are easiest described using the first person – with apologies for breaking the etiquette.

The first stage of the experiment was to obtain parsing expression grammar for Java 1.5. It turned out that Java grammars published in connection to *Pappy* and *Rats!* are not exactly pure PEG. Besides, they were for earlier Java versions, so I decided to write the grammar from scratch.

The official source of the grammar is Java Language Specification (JLS) [8]. Almost all syntax definitions in that text are heavily left-recursive. In most cases, replacing left recursion by iteration is rather straightforward. But the most central part, the definition of "Primary", is an exception. The left recursion there involves several levels of definitions; after several attempts to disentangle it, I gave up and turned to the summary of the grammar in Chapter 18 of JLS. It does not have operator hierarchy and omits the lexical parts; I used for them definitions from JLS Chapters 3 and 15.

Quite naturally, the grammar was not correct at first, and had to be debugged. Debugging consisted of transcribing the grammar into executable form – a primitive parser – and applying that parser to Java source files. As the grammar has over 200 rules, I had to use a tool – a simple parser generator – to perform the transcription.

Writing the grammar from scratch and debugging it gave me some insights into the working of PEG. With the original specifications in EBNF, you notice immediately that, in spite of a similar form, PEG is *not* EBNF. Contrary to what you may first expect, PEG does not backtrack to check *all* possibilities. A parsing procedure will never let a lower-level procedure reconsider an already successful decision. If `E1` in the rule `A = E1 E2` succeeds, and then `E2` fails, the procedure for `A` does not invoke `E1` to try another parse.

As a specific example, consider the EBNF syntax `("+"|"++")[a-z]`. The string `++n` conforms to that syntax, but it is not accepted by parsing expression `("+"/"++")[a-z]`. The first alternative of `("+"/"++")` succeeds and consumes the first `+`, after which `[a-z]` fails on the second `+`. The whole expression fails without ever reconsidering the second alternative of `("+"/"++")`. To avoid such "prefix capture", you have to make sure that none of the strings accepted by `E1` in choice expression `E1/E2` is a prefix of any string accepted by `E2`. In the above example, the problem is easily seen and solved by reversing the order. It is not that simple when `E1` and `E2` are defined by several levels of rules.

One of the main ideas with PEG is to specify complete syntax, down to the lexical structure. That means the grammar must specify all tasks traditionally performed by lexers. One of them is ignoring white space and comments. The method used in [6,7,9] is to define "Spacing" that consumes the elements to be ignored. The grammar must then define each "token" to be followed by Spacing. That means each operator, such as "`+`", must be defined by a rule like this:

```
PLUS = "+" ![+=] Spacing?
```

Notice the predicate `![+=]` that prevents `PLUS` from capturing the prefix of operators `++` and `+=`. An "Identifier" can be defined like this:[2]

```
Identifier = !Keyword Letter LetterOrDigit* Spacing?
Keyword = ("abstract" / ... / "while") !LetterOrDigit
Letter = [a-z] / [A-Z] / [_$]
LetterOrDigit = Letter / [0-9]
```

Predicate `!Keyword` implements the rule that Identifier must not be any of the 53 "keywords" listed in JLS Section 3.9. Predicate `!LetterOrDigit` at the end of `Keyword` prevents prefix capture: without it, strings such as `abstractLine` would never be recognized as identifiers. Keywords themselves must be defined like this:

```
ABSTRACT = "abstract" !LetterOrDigit Spacing?
```

By trial and error, I found and corrected a number of hidden prefix captures and other problems. I also found some errors in JLS. Did I find all? I do not know. The grammar accepts 10 522 files of J2SE 5.0 JDK source code obtained from the official Java site, but this does not prove it correct.

[2] JLS defines letters and digits to be the Unicode characters recognized as such by special Java methods. This cannot be easily defined by Parsing Expressions, so I chose the conservative definition.

4 Evaluating the parser

To investigate the behavior of the parser, I included some instrumentation in the code generated from PEG. With this instrumentation, a typical parsing procedure looks as follows:

```
// A = E1 E2 ... En
   boolean A()
     {
       init(A);
       if (!E1()) return reject();
       if (!E2()) return back();
          ...
       if (!En()) return back();
       return accept();
     }
```

Each symbol of the grammar is represented by an object of class "Symbol", named after that symbol ("A" in this case). The object contains variables for tracking calls to the corresponding procedure. Method `init()` registers a call to the procedure and stacks an object of class "Call" identifying current input position and the symbol. Method `accept()` registers successful completion, unstacks the Call object and returns `true`. Method `reject()` registers failure, unstacks the Call object, and returns `false` without changing the input position. Method `back()` restores input position from the stacked Call object, registers backtracking, unstacks the object, and returns `false`. There are separate procedures and Symbol objects for nested expressions, with generated names (such as `sum1` in Figure 1). There are also Symbol objects for terminals; they are used for counting calls to `next()` and `nextIn()`.

I applied the instrumented parser to 80 files selected from: the J2SE 5.0 JDK source code, the ANTLR and *Rats!* libraries, an industrial application, and my own code. The files were randomly selected to cover the range of sizes from 0.1 kB to 476 kB.

The table in Figure 4 shows some numbers obtained for twelve of these 80 files. Column "calls" shows the processing "time" expressed as the number of calls to parsing procedures and to procedures that test for terminals. I chose this measure, rather than the clock time, because it is repeatable and independent of the environment. Column "calls/byte" shows the number of calls per byte of non-comment code. Its purpose is to asses the dependence between performance and file size. Because some files contained a large percentage of comments (column "comments"), and comments require only 3 calls per byte, dividing the total number of calls by total size does not give a fair measure. As one can see, the result is quite independent of the file size – a good approximation to linear time. The values for all 80 files varied from 7.9 to 36.2 calls per byte, with 22.2 as the median. Their distribution is shown by the histogram on the right.

file	size kB	comments	calls	calls/byte
1	0.1	0.0 %	2 540	21.0
2	0.3	21.1 %	2 715	11.6
3	1.1	67.3 %	9 032	21.5
4	2.8	67.9 %	23 654	23.2
5	3.1	24.0 %	63 635	26.4
6	6.4	33.7 %	86 065	19.2
7	14.1	35.9 %	212 180	22.4
8	39.8	49.7 %	456 615	21.3
9	57.6	27.3 %	883 689	20.2
10	140.3	42.0 %	1 694 212	19.6
11	257.8	18.9 %	5 005 436	23.1
12	424.7	18.5 %	8 340 972	23.3

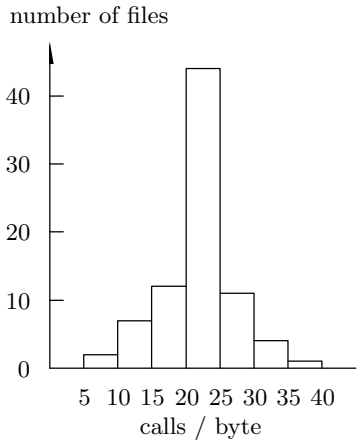

Fig.4.

Figure 5 contains more detailed statistics collected for the same twelve files. The last row contains totals for all 80 files. Columns "size" and "calls" are the same as in Figure 4. The next three columns break down the number of calls according to their result. Column "true" contains the number of calls that

returned `true`. These are the calls where the actual parser would do its real work in the sense of processing the semantics. Column "false" contains the number of calls that returned `false` without backtracking. These are failed probes for next character(s), often involving several levels of descent. Column "back" contains the number of calls that returned `false` after backtracking. Their remarkably low number may be partially explained by the fact that if a deeply nested call backtracks, all the enclosing calls may fail without backtracking. Another factor is that large number of calls are tests for terminals that never backtrack.

The last three columns contain more information about backtracking. Columns "avg" and "max" show, respectively, the average and maximum length of a backtrack in bytes. As a curiosity, one may note that the longest backtrack was over a 309-character comment in the middle of an expression. Column "repeated" shows how many calls among those reported under "calls" were repetitions: the same procedure called again at the same input position. It is interesting that while very few calls ended up in backtrack, the resulting number of repetitions is large.

file	size kB	calls	true	false	back	avg	max	repeated
1	0.1	2 540	388	2 149	3	11.7	15	764
2	0.3	2 715	754	1 957	4	1.0	1	188
3	1.1	9 032	3 455	5 544	33	4.2	25	1 499
4	2.8	23 654	8 361	15 245	48	2.8	17	3 373
5	3.1	63 635	11 031	52 480	124	4.6	34	10 757
6	6.4	86 065	19 580	66 333	152	4.7	32	15 148
7	14.1	212 180	47 675	164 000	505	5.7	28	47 860
8	39.8	456 615	119 598	336 311	706	4.1	34	72 320
9	57.6	883 689	192 927	688 712	2 050	2.9	46	148 552
10	140.3	1 694 212	437 885	1 253 816	2 511	3.9	58	246 796
11	257.8	5 005 436	872 825	4 126 644	5 967	5.0	35	841 054
12	424.7	8 340 972	1 447 565	6 883 528	9 879	5.0	33	1 426 231
all 80	5 728.1	78 022 302 (100 %)	18 176 015 (23.3 %)	59 694 657 (76.5 %)	151 630 (0.2 %)	3.8	309	12 552 070 (16.1 %)

Fig.5.

5 Some improvements

5.1 Saving results

The results in Figure 5 show that backtracking usually caused re-scanning of a very short portion of input. That means repeated calls to a procedure at the same input position were most likely not separated by a call at a different position. To exploit that, I modified the generated code to save the last result of each parsing procedure, and reuse it if the procedure is called again for the same input. With this modification, a typical parsing procedure looks as follows:

```
// A = E1 E2 ... En
   boolean A()
     {
       if (saved(A)) return reuse();
       if (!E1()) return reject();
       if (!E2()) return back();
          ...
       if (!En()) return back();
       return accept();
     }
```

Every time a Call object is put on stack, a reference to it is placed in the corresponding Symbol object, and is left there after the Call has been unstacked. The Call object has an additional field, the "ending position". Method `accept()` enters there the current input position before unstacking. Methods `reject()`

and `back()` enter −1. Method `saved()` checks if the Symbol has a saved Call with the current starting position. If not, it stacks a new one, registers it in Symbol (replacing the older one), and returns `false`. After this, processing continues as before. If there is a saved Call with current position, `saved()` returns `true`. This gives control to `reuse()` that, depending on the ending position in Call, either returns `false` or advances to the ending position and returns `true`.

(Note that even in this form, the procedure retains its simplicity and direct relation to the grammar. This is, in fact, how the procedure may appear in a practical parser; the stacked objects would then include semantic information. A call to semantic procedure just before the last line would process that information.)

Running the modified parser on the same 80 files as before, I saw a significant reduction in the number of repeated calls. They were reduced to 3.3% of all calls; the total number of calls has been reduced by about 13%.

Encouraged by this effect, I modified my parser to keep the two most recent results. This reduced the number of repeated calls to only 1.1% of all calls. Figure 6 shows total numbers for versions with 0, 1, and 2 results saved, and illustrates them on a diagram. It seems that keeping more than two recent results is not needed for this grammar.

saved	calls	repeated	%
0	78 022 302	12 552 070	16.1
1	67 772 649	2 269 372	3.3
2	66 223 061	719 784	1.1

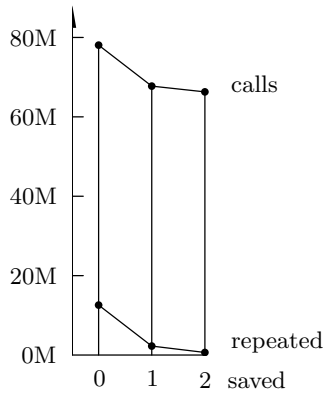

Fig.6.

5.2 Identifiers and keywords

The instrumented parser collected statistics separately for each Symbol. According to these detailed statistics, much of backtracking and repetitions occurred in parts of the parser dealing with lexical structures. It appears that PEG is not the best tool to specify such structures. In particular, the `!Keyword` predicate in the definition of Identifier means that the corresponding procedure always starts by comparing the text ahead with all the 53 "keywords". Only after all these comparisons failed, the procedure continues to look for the first letter of the identifier.

To asses the cost of this process, I manually modified the parser so that the procedure for Identifier first recognizes any `Letter LetterOrDigit*` and then uses hash table to check if it is a keyword. Figure 7 shows the same kind of results as Figure 6, obtained with the modified parser. For comparison, the dotted lines in the diagram show the results without modification. As one can see, using PEG to specify reserved words costs about 20% of the total effort. It is not surprising that the grammars published in connection with *Pappy* and *Rats!* go outside PEG to define identifiers, keywords, and operators.

The ease of making the modification demonstrated for me once more the advantages of a primitive parser.

saved	calls	repeated	%
0	61 796 491	6 388 248	10.3
1	56 316 593	908 350	1.6
2	55 721 435	313 192	0.6

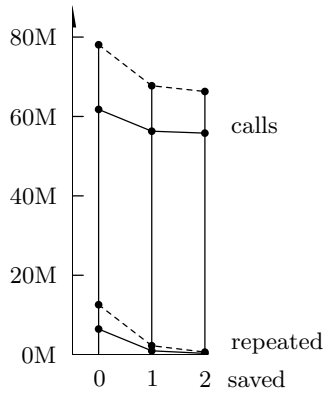

Fig.7.

6 Conclusions

The described experiments show that a primitive recursive-descent parser with limited backtracking and integrated lexing is a reasonable possibility for parsing Java 1.5 where performance is not too critical.

It should be clear that all measurements were made on average, correct, industrial-quality programs. No attempt was made to invent worst-case constructions, possibly with incorrect syntax. The exact values of backtrack lengths, repetitions, and calls per byte may be somewhat optimistic because the grammar in JLS Chapter 18 has been simplified so that "in many cases it minimizes the necessary look ahead" (quote from the Chapter's opening paragraph). However, the linear-time result does not seem affected by this simplification. It will be interesting to repeat the experiment with untainted grammar, and then do the same for C. The problem with C is that it exists in many versions and involves a preprocessor. It also has the `typedef` statement that introduces new keywords.

The exercise with constructing the grammar from scratch gave some feeling of PEG as a language specification tool. In [7], PEG is advanced as a tool for describing syntax, better than context-free grammars and regular expressions. One of the arguments is that the grammar is unambiguous. True, it is an unambiguous specification of a *parser*. But, the *language* specified by it is whatever that parser happens to accept. Can we easily see that it is the language we want? The idea of "specification" is that its meaning must be clear to a human reader. As shown in Section 3, PEG contains pitfalls in the form of "prefix capture" that are not immediately visible. Any usable parser generator for PEG must be able to detect prefix capture in addition to left recursion. (A recent paper [13] suggests how to do this.)

What other conditions must be detected to make sure the grammar defines the language we want? Or, should we simply learn to think in terms of Parsing Expressions? Does BNF seem easier to understand because it better reflects the working of human mind, or because we are just used to it?

Finally: the definitions of identifiers, keywords, and operators shown in Section 3 do not clearly convey their meaning. As shown in Section 5.2, they also have a significant performance cost. Can one modify PEG to better specify these elements?

Acknowledgements

The author thanks two anonymous referees for a number of useful suggestions and improvements.

References

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

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

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

[4] Christopher, T. A strong LL(k) parser generator that accepts non-LL grammars and generates LL(1) tables: Extended abstract. Dept. of Computer Science, Illinois Institute of Technology, `http://www.iit.edu/~tc/llkppr.ps`.

[5] Conway, D. The man (1) of descent. *The Perl 3*, 4 (1998), 46–58. `http://search.cpan.org/src/DCONWAY/Parse-RecDescent-1.94/tutorial/tutorial.html`.

[6] Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002. `http://pdos.csail.mit.edu/papers/packrat-parsing:ford-ms.pdf`.

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

[8] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, The 3rd Edition*. Addison-Wesley, 2005. `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`.

[9] Grimm, R. Practical packrat parsing. Tech. Rep. TR2004-854, Dept. of Computer Science, New York University, March 2004. `http://www.cs.nyu.edu/rgrimm/papers/tr2004-854.pdf`.

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

[11] Parr, T., and Quong, R. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810. `http://www.antlr.org/article/1055550346383/antlr.pdf`.

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

[13] Schmitz, S. Modular syntax demands verification. Tech. Rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, Oct. 2006. `http://www.i3s.unice.fr/~mh/RR/2006/RR-06.32-S.SCHMITZ.pdf`.