A Primitive Recursive-Descent Parser with Backtracking

Roman R. Rędziejowski

roman.redz@swipnet.se

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. The present paper reports an experiment that consisted of defining the syntax of Java 1.5 in PEG formalism, and constructing a parser from it by 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 to better understand 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 [9], 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 the BNF or BNF-like grammar that defines the syntax. 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 extended BNF can be easily transcribed into a primitive recursive-descent parser, such as the Java program in Figure 1. Method `next(`c`)` in that program consumes the next input character and returns `true` if that character is c; otherwise it returns `false`. Method `error()` throws an exception.

Such parser will always terminate if the grammar does not contain left recursion and the expression E in $E*$ or $E+$ does not specify empty string. If, in addition, the grammar is $LL(1)$ and the expression E_1 in each concatenation $E_1E_2 \ldots E_n$ does not specify empty string, the top procedure returns `true` if and only if its input conforms to the grammar.

A great advantage of parser such as in Figure 1 is its simplicity and clear relationship to the grammar. For smaller grammars, it 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,

Grammar:

```
sum = term ("+" term)* ";"
term = name | number
```

Parser:

```
// sum = term ("+" term)* ";"
boolean sum()
  {
    if (!term())   return false;
    while (sum_1());
    if (!next(";") error();
    return true;
  }

// sum_1 = "+" term
boolean sum_1()
  {
    if (!next("+")) return false;
    if (!term())    error();
    return true;
  }

// term = name | number
boolean term()
  {
    if (name())   return true;
    if (number()) return true;
    return false;
  }
```

Fig.1.

normally driven by huge tables without obvious relationship to the grammar, that *must* be mechanically generated.

The grammar being $LL(1)$ means that `term()` in Figure 1 can test the next character and directly call `name()` or `number()` or return `false`, rather than delegate the test to the called procedures. This strategy is known as *predictive parsing*. It trades off simplicity for speed. But, if performance is not critical, the simplicity may be more desirable.

Unfortunately, most languages of practical interest do not have $LL(1)$ grammar. Some of this problem can be handled in a "lexer" that converts the input stream into a sequence of "tokens" given as input to the parser proper. 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 [10], or Parse::RecDescent [5] solve the problem by providing mechanisms to look at more than one character ahead. There is also a report [4] on generator that produces $LL(1)$ parsers from non-LL 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]: backtracking. If one alternative fails, go back in the input and try another. This is a continuation of ideas from the early 1970's [1–3] that were abandoned as exceeding the technology of that time. The problem with backtracking is that, in the worst case, it may take time that is exponential in the input length. 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 [8].

An integral component of Ford's approach is a new way to specify syntax. It is called *Parsing Expression Grammar* (PEG), and defines a language in terms of recognizing strings. This is contrary to context-free grammars and regular expressions that define languages in terms of constructing strings.

Parsing Expression Grammar is, in fact, formal specification for a backtracking recursive-descent parser. It can be easily transcribed into a parser similar to that of Figure 1. The present paper reports the author's experiments with such parser for Java 1.5. The experiments seem to confirm the author's feeling that packrat strategy is not really necessary for programming languages in practical use. The primitive parser exhibits a moderate backtracking activity; the parsing "time", measured in number of procedure calls, seems to grow 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. (Thrifty mouse instead of a pack rat.) This modification could be comfortably hidden behind the scenes without destroying the clarity of 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. Finally, Sections 4 and 5 describe experiments with the parser.

2 Parsing Expression Grammar

Parsing Expression Grammar (PEG) is a set of definitions of the form $A \leftarrow E$ where A is a (nonterminal) symbol of the grammar and E is a *parsing expression*.

Parsing expressions are similar in form to expressions in extended BNF, but have an entirely different meaning: they are instructions for *parsing* strings, rather than *constructing* them. 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 a nonterminal defined in the grammar.

Parsing expressions are applied to character strings. An expression may match initial portion of a string, and possibly "consume" the matched portion.

[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.

It may then indicate "success" or "failure". These actions are described informally in the Figure; precise formal definitions can be found in [7].

An expression is said to "accept" a string if it returns success when applied to that string. The language defined by the grammar is the set of strings accepted by a specified "starting expression".

$E_1/\ldots/E_n$	Ordered choice: Try expressions $E_1 \ldots E_n$, in this order, until one of them matches the text ahead. Consume the matched text and indicate success. Indicate failure if none of them matches.
$E_1 \ldots E_n$	Sequence: Apply expressions $E_1 \ldots E_n$ one after another, to match consecutive portions of the text ahead. If all succeed, consume all matched text and indicate success. If any of them fails, 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.
.	Any character: If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.
"s"	String: If the text ahead is the string s, consume it and indicate success. Otherwise indicate failure.
[c_1-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]	Character class: If the character ahead appears in the string s, consume it and indicate success. Otherwise indicate failure.

Fig.2.

The actions of different expressions can also be precisely defined by Java procedures shown in Figure 3. The procedures take a character string as input and indicate success or failure by returning `true` or `false`. Method `pos()` returns current position in the input. Method `back()` restores that position and returns `false`. Procedures `next()` and `nextIn()` test next input character(s), and are easy to figure out.

From Theorem in section 3.6 of [7] follows that all procedures 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 set of procedures constructed for a specific grammar using patterns from Figure 3 constitutes a primitive parser. The language defined by that grammar is the set of input strings for which the "starting procedure" returns `true`.

```
// A <- E1 / E2 / ... / En
boolean A()
  {
    if (E1()) return true;
    if (E2()) return true;
       ...
    if (En()) return true;
    return false;
  }

// A <- E1 E2 ... En
boolean A()
  {
    int p = pos();
    if (!E1()) return false;
    if (!E2()) return back(p);
       ...
    if (!En()) return back(p);
    return true;
  }


// A <- &E
boolean A()
  {
    int p = pos();
    if (!E()) return false;
    return !back(p);
  }

// A <- !E
boolean A()
  {
    int p = pos();
    if (!E()) return true;
    return back(p);
  }
```

```
// A <- E+
boolean A()
  {
    if (!E()) return false;
    while (E());
    return true;
  }

// A <- E*
boolean A()
  {
    while (E());
    return true;
  }

// A <- E?
boolean A()
  {
    E();
    return true;
  }

// .
boolean next()
  { ... }

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

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

// [s]
boolean nextIn(String s)
  { ... }
```

Fig.3.

According to [7], PEG formalism can define a large class of context-free languages, including all LR-class languages. It can also define some non-context free languages, but the question if it can define all context-free languages is stated as an open problem.

3 Writing the grammar

This and the following sections report experiments performed by the author, that are easiest described using the first person. (With apologies for breaking the etiquette.)

As the first stage, I had to create parsing expression grammar for Java 1.5. It turned out that the grammars supported by *Pappy* and *Rats!* are not exactly pure PEG. Besides, the Java grammars available for these generators are for earlier Java versions. Writing the grammar from scratch and testing it gave me some impressions of PEG as a language specification tool.

The official source of the grammar is Java Language Specification (JLS) at `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`. However, almost all syntax specifications scattered in the JLS text are left-recursive. The summary of the grammar in Chapter 18, intended to be $LL(1)$-close, has a very simplified syntax of expressions. At the end, it was possible to obtain much of the PEG by modifying the ANTLR Java grammar from `http://www.antlr.org/grammar/list`. Both JLS Chapter 18 and ANTLR grammar omit the lexical parts; I defined them by paraphrasing the JLS syntax definitions.

To test different versions of the grammar, I wrote a simple generator that converts PEG into a primitive parser according to Figure 3. It does some in-lining of procedures for $E+$, $E*$, and $E?$, in a similar way as in Figure 1.

One of the main ideas with PEG is to specify complete syntax, down to the lexical structure. This gives an integrated parser, without a separate lexer. 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 [6–8] is to define a nonterminal "Spacing" that recognizes the elements to be skipped. The grammar must then define each "token" to be followed by Spacing. That means each operator, such as `"+"`, and each keyword, such as `"new"`, must be redefined as a nonterminal, for example:

```
PLUS <- "+"   Spacing
NEW  <- "new" Spacing
```

A serious difficulty was the definition of identifiers. JLS defines an "Identifier" as "JavaLetter JavaLetterOrDigit*", but "JavaLetter" is much more than the usual characters 'a' through 'z' and 'A' through 'Z'. It is namely any Unicode character for which the method "Character.isJavaIdentifierStart()" returns `true`. Similarly, "JavaLetterOrDigit" is any Unicode character for which the method "Character.isJavaIdentifierPart()" returns `true`. This cannot be easily

defined by Parsing Expressions, but can be implemented in the parser by hand-coded procedures for "JavaLetter" and "JavaLetterOrDigit" – a simple task as long as the language used is Java. For my experiments, I chose the conservative definition:

```
Letter <- [a-z] / [A-Z] / [_]
LetterOrDigit <- Letter / [0-9]
Identifier <- Letter LetterOrDigit* Spacing
```

But this is not all. An "Identifier" must not be any of the character strings specified as "keywords". There are about 50 such keywords, from "abstract" through "while". This can be specified in PEG formalism as

```
Identifier <- !Keyword Letter LetterOrDigit* Spacing
Keyword <- "abstract" / ... / "while"
```

But this is still not all. With such definition, the string "newValue" is not recognized as "Identifier". The expression for "Keyword" accepts the prefix "new", and the predicate "!Keyword" fails. A correct definition seems to be:

```
Identifier <- !Keyword Letter LetterOrDigit* Spacing
Keyword <- ("abstract" / ... / "while") !LetterOrDigit
```

For exactly the same reason, the definition of "NEW" given above as an example is not correct. It recognizes the initial portion of identifier "newValue" as keyword "new". In the tests I ran, no amount of backtracking could then find the correct interpretation. A correct definition is, of course:

```
NEW <- "new" !LetterOrDigit Spacing
```

These definitions of "Identifier" and "NEW" do not clearly convey the idea of an identifier or specific keyword. Intuitively, both are special cases of "word": a maximal string starting with "Letter" followed by zero or more "LetterOrDigit"s. In fact, the grammars available in connection with *Pappy* and *Rats!* follow this intuition by defining "word" in PEG and identifying the special cases by means outside PEG.

The "prefixing problem" is not limited to identifiers and keywords. Mechanically replacing the BNF alternative "`|`" by PEG ordered choice "`/`", I obtained `"<" / ">" / "<=" / ">="` as the definition of "relational operator". Of course, the last two alternatives are never recognized, causing the parser to fail on simple tests. The problem here is immediately visible, but in many cases it may be hidden by many levels of recursion. Sometimes the parser can get the correct interpretation after backtracking, but it looks more like good luck; in most cases backtracking does not help.

As a protection against unexpected prefixing, I had to define operators like this:

```
PLUS <- !("++"/"+=") "+" Spacing
```

which is not the most obvious way to define `"+"`. Again, the grammars for *Pappy* and *Rats!* handle the prefixing of operators outside PEG by defining nonterminal "Symbol" to include all operators, and testing for specific value.

By trial and error, I found a number of hidden prefixing problems. I was able to eliminate them by changing the order in some choice expressions. (For example, putting "FloatLiteral" before "IntegerLiteral".) But, did I find all? The parser constructed from my grammar accepts twenty or so Java programs, but this does not prove it correct.

4 Evaluating the parser

A primitive parser constructed from PEG can, in principle, use time that is exponential in the length of input. But this does not seem the case for most programming languages. The popular languages such as Java or C have large parts that are $LL(1)$. Besides, one can identify in them frequent "points of no return". Their top definitions follow this pattern:

```
Program <- Unit+ EndOfText
```

where "Unit" is some basic syntactic unit such as declaration or statement. Any failure in Unit may cause backtracking to at most the start of that unit. The time is at worst exponential in the length of Unit, times the number of Units. If the length of Unit is moderate and limited (which is usually the case), the time will be approximately linear.

To test this observation in practice, I included some instrumentation in my parser. With instrumentation, a typical parsing procedure looks like this:

```
// 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 procedure. Method `init()` registers 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 input position. Method `back()`

restores input position from the stacked Call object, registers backtracking, unstacks the object, and returns `false`.

(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. Note that even in this form, the procedure retains its simplicity and direct relation to the grammar.)

There are separate procedures and Symbol objects for nested expressions, with generated names. There are also Symbol objects for terminals; the methods `next()` and `nextIn()` are modified to update them. My Java grammar had altogether 491 nonterminals (of both kinds) and 116 terminals.

Figure 4 shows statistics from applying the parser to twelve Java source files of different sizes. The files were "randomly" selected from ANTLR and *Rats!* libraries, the Cryptix library, and from my own programs. The main criterion was to cover a range of sizes from small to large.

case	size	calls	rescans	true	false	back	avg	max
1	0.6	12 247	2 716	2 490	9 726	31	4.4	18
2	2.6	18 398	3 544	8 542	9 816	40	4.9	10
3	6.3	75 499	13 129	22 311	53 020	168	3.3	23
4	14.4	415 600	64 976	86 885	326 994	1 721	2.4	18
5	29.4	264 717	54 247	92 259	172 079	379	5.3	18
6	60.8	1 232 483	203 591	310 995	913 056	8 432	2.7	23
7	62.8	1 522 774	244 124	372 172	1 139 638	10 964	3.0	23
8	86.9	2 086 659	480 125	413 829	1 668 766	4 064	6.7	28
9	134.5	2 795 232	625 668	626 015	2 163 387	5 830	5.9	36
10	141.0	4 032 661	983 662	739 447	3 279 757	13 457	4.9	40
11	257.8	5 353 601	1 156 189	1 148 727	4 194 566	10 308	6.6	45
12	424.7	8 929 886	1 954 651	1 908 279	7 004 463	17 144	6.6	43
total	1221.7	26 739 757	5 786 622	5 731 951	20 935 268	72 538	5.1	45

Fig.4.

The table shows file sizes in kB. Column "calls" contains the number of calls for parsing procedures, including both terminals and nonterminals. Because of backtracking, some calls were repeated for the same input position. Column "rescans" shows how many calls shown in the preceding column were such repetitions. The next three columns count different exits from the procedures reported under "calls". These columns show, respectively, the number of "true" exits, the number of "false" exits without backtracking, and the number of "false" exits with backtracking. The rightmost two columns show, respectively, the average and maximum length of a backtrack. Notice that the "true" exits are the cases where the actual parser does its real work in the sense of processing the semantics. The "false" exits without backtrack represent unsuccessful tests of the next input character(s).

The table shows an approximately linear dependence between file size and the number of calls, and a relatively constant backtrack length. It is interesting that while very few calls (ca 0.3%) end up in backtrack, the resulting number of re-scans is large, about 22% of all calls.

The instrumented parser collects statistics separately for each Symbol. According to these detailed statistics, the bulk of backtracking and rescanning occurred in parts of the parser dealing with lexical structures. An obvious explanation is that non-lexical parts of my grammar were modeled after the ANTLR and JLS Chapter 18 grammars, formulated to be $LL(1)$-close, while the lexical parts were taken raw from JLS.

The largest part of rescanning was concerned with recognizing identifiers and keywords.

5 Some improvements

5.1 Saving results

The results in Figure 4 show that backtracking usually caused re-scanning of a very short portion of input. That means a specific parsing procedure was very likely called to re-scan the same input that it processed the last time. To exploit that, I modified my parser 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 like this:

```
// 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`.

Running the modified parser on the same twelve files as before, I saw a significant reduction in the number of rescans. They were reduced, on the average, to 5% of all calls; the total number of calls has been reduced by about 17%.

Encouraged by this effect, I modified my parser to keep the two most recent results. This reduced the number of rescans to only 1.2% of all calls. Figure 5 shows total numbers for versions with 0, 1, and 2 results saved. It seems that keeping one or at most two recent results is quite satisfactory for this grammar.

save	calls	rescans	true	false	back	avg	max
0	26 739 757	5 786 622	5 731 951	20 935 268	72 538	5.1	45
1	22 097 443	1 144 308	5 255 565	16 772 082	69 796	5.2	45
2	21 212 292	259 157	5 055 711	16 087 040	69 541	5.2	45

Fig.5.

5.2 Identifiers and keywords

As mentioned before, much of the backtracking and rescanning activity was related to identifiers and keywords. 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 6 shows the same kind of results as Figure 5, obtained with the modified parser. The significant reduction of the number of calls shows that the formalism of PEG was not very good for the purpose. The ease of manually modifying the parser once more demonstrated for me the advantages of simple and transparent construction.

save	calls	rescans	true	false	back	avg	max
0	19 262 528	2 913 924	5 639 623	13 553 215	69 690	5.2	45
1	16 786 950	438 346	5 204 613	11 514 426	67 911	5.2	45
2	16 437 812	89 208	5 014 795	11 355 134	67 883	5.2	45

Fig.6.

6 Conclusions and further work

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

The exercise with constructing the grammar gave some feeling of PEG as language specification tool. Apparently, the main purpose of introducing PEG in [6] was to provide a formal description of backtracking parser, making possible rigorous proofs of different facts. In [7], PEG is advanced as a language specification tool, better than BNF and regular expressions. One of the arguments is that the specification is unambiguous. True, it is an unambiguous specification of a *parser*. But, the *language* specified by it is whatever this 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. This is why a program is normally not considered to be its own specification (although it specifies very exactly what it does!). As we have seen in

Section 3, PEG contains pitfalls in the form of unexpected prefixing. A tool to detect such prefixing may help to debug the grammar. But, in some cases, like keywords and operators, removing the problem destroys readability, or leads outside PEG. A better mechanism to handle these cases would be desirable.

Apparently, most of the specification problems, and most of backtracking, occurred in the lexical parts of the language. These parts are traditionally specified by regular expressions and parsed bottom-up. Do these methods better reflect the working of human mind, or are we just used to them, and the problems indicate the need to better learn the new approach?

It will be interesting to repeat the described experiments with C. The difficulty was so far the diversity of C grammars and finding test cases that fit the grammar in hand. There is also an inherent problem, as the language may define new keywords by means of `typedef` statement. The author has so far conducted an experiment with one grammar and one test case (without `typedef`). Statistics for this case look similar to those for Java, although backtracks are somewhat longer.

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.
   Also as `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.
   Also as `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.
   Also as `http://pdos.csail.mit.edu/papers/parsing:popl04.pdf`.
8. Grimm, R. Practical packrat parsing. Tech. Rep. TR2004-854, Dept. of Computer Science, New York University, March 2004.
   Also as `http://www.cs.nyu.edu/rgrimm/papers/tr2004-854.pdf`.
9. Lucas, P. The structure of formula-translators. *ALGOL Bulletin Supplement 16* (September 1961), 1–27.
10. Parr, T., and Quong, R. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810.
    Also as `http://www.antlr.org/article/1055550346383/antlr.pdf`.