COMPUSOFT
An International Journal of Advanced Computer Technology

ISSN:2320-0790

Parser Generator for Parsing Expression Grammar

Madhavi Tota[1], Prof. P. Pradeep Kumar[2]

Vivekananda Institute of Technology & Science
Karimnagar (A.P), India

tota.madhavi@yahoo.co.in[1], pkpuram@yahoo.com[2]

Abstract: In the field of formal languages apart from context free grammar (CFG) a new approach is developed i.e. Parsing Expression Grammar (PEG). Parsing Expression Grammar (PEG) is a new way to specify recursive-descent parsers with limited backtracking. The use of backtracking lifts the *LL*(1) restriction usually imposed by top-down parsers. In addition, PEG can directly define the structures that usually require a separate "lexer" or "scanner". The parser has many useful properties, and with the use of memorization, it works in a linear time. This paper reports an experiment that consisted of defining PEG formalism, and literally transcribing the PEG definitions into parsing procedures.

Keywords: PEG, memorization, lexer, backtracking, CFG

I. INTRODUCTION

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. The great advantage of a recursive-descent parser is its simplicity and clear relationship to the grammar. For smaller grammars, the parser can be easily produced and maintained by hand. 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. The problem with constructing recursive-descent parsers from a classical context-free grammar is that the grammar must have the so-called *LL*(1) property. Forcing the language into the *LL*(1) mold can make the grammar – and the parser – unreadable. [1–3] introduced a language for writing recursive-descent parsers with limited backtracking. It is called Parsing Expression Grammar (PEG) and has the form of a grammar that can be easily transcribed into a set of recursive procedures.

II. PARSING EXPRESSION GRAMMAR

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

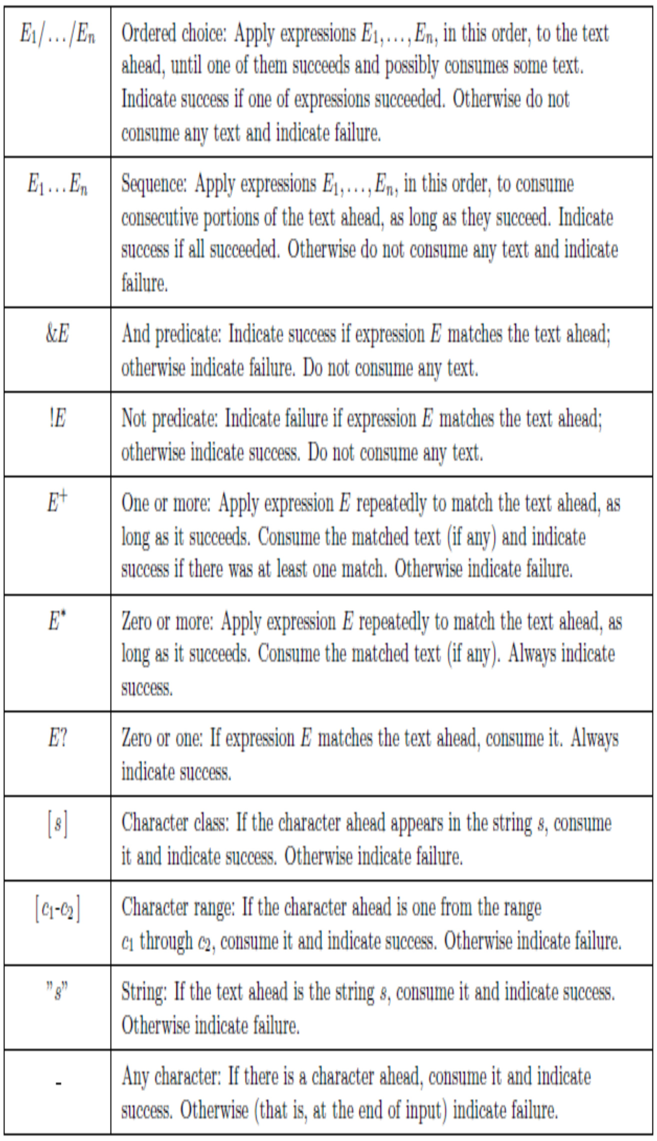
Language parsing expression is a procedure that carries out instruction. The expressions can call each other recursively, thus forming together a recursive-descent parser [5].
In general, parsing procedure is applied to a word from Σ and tries to recognize an initial portion of that word. If it succeeds, it "consumes" the recognized portion and returns "success"; otherwise, it returns "failure" and does not consume anything. The action of different procedures is as follows:

– '': Indicate success without consuming any input.

– $a \in \Sigma$: If the text ahead starts with a, consume it and return success. Otherwise return failure.

– $A = e1\ e2$: Call $e1$. If it succeeded, call $e2$ and return success if $e2$ succeeded.
If $e1$ or $e2$ failed, backtrack: reset the input as it was before the invocation of $e1$ and return failure.

– $A = e1|e2$: Call $e1$. Return success if it succeeded. Otherwise call expression $e2$ and return success if $e2$ succeeded or failure if it failed. Note the limited backtracking: $A = e1|e2$ has never a chance to try $e2$ once $e1$ succeeded.



$E_1/\ldots/E_n$	Ordered choice: Apply expressions $E_1,\ldots,E_n$, in this order, to the text ahead, until one of them succeeds and possibly consumes some text. Indicate success if one of expressions succeeded. Otherwise do not consume any text and indicate failure.
$E_1\ldots E_n$	Sequence: Apply expressions $E_1,\ldots,E_n$, in this order, to consume consecutive portions of the text ahead, as long as they succeed. Indicate success if all succeeded. Otherwise do not consume any text and indicate failure.
$\&E$	And predicate: Indicate success if expression E matches the text ahead; otherwise indicate failure. Do not consume any text.
$!E$	Not predicate: Indicate failure if expression E matches the text ahead; otherwise indicate success. Do not consume any text.
E^+	One or more: Apply expression E repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any) and indicate success if there was at least one match. Otherwise indicate failure.
E^*	Zero or more: Apply expression E repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any). Always indicate success.
$E?$	Zero or one: If expression E matches the text ahead, consume it. Always indicate success.
$[s]$	Character class: If the character ahead appears in the string s, consume it and indicate success. Otherwise indicate failure.
$[c_1\text{-}c_2]$	Character range: If the character ahead is one from the range c_1 through c_2, consume it and indicate success. Otherwise indicate failure.
"s"	String: If the text ahead is the string s, consume it and indicate success. Otherwise indicate failure.
-	Any character: If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

Fig 1: Parsing Expressions

Figure 1 lists all forms of parsing expressions. Each of *E;E*1…..*En* 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.

Figure 2 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[1-2]. 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  Sequence: A = E1E2 ...
En
boolean A()
                                        boolean A()
{                                       {
if (E1()) return true;                  int p = pos;
if (E2()) return true;                  if (!E1()) return false;
...                                      if     (!E2())     return
back(p);
if (En()) return true;                   ...
return false;                           if     (!En())     return
back(p);
}                                       return true;
// And predicate: A = &E
boolean A()                           // Not predicate: A = !E
{                                        boolean A()
int p = pos;                            {
if (!E()) return false;                 int p = pos;
return !back(p);                        if (!E()) return true
}                                        return back(p);    ;

// One or more: A = E+
boolean A()                             // Zero or more: A =
E*
{                                        boolean A()
if (!E()) return false;                              {
while (E());                            while (E());
return true;                            return true;
}                                        }
// Zero or one: A = E?              // Character class [s]
boolean A()                        boolean nextIn(String s)
{                                       {
E();                           if     (pos>=endpos)     return
false;
return true;                   if (s.indexOf(input[pos])<0)
}                                  return false;
                                        pos++;
// Character range [c1-c2]       return true;
                                  boolean nextIn(char c1, char
                              c2)        }
{ ... }
// String "s"                    // Any character
boolean next(String s)            boolean next()
{ ... }                             { ... }
```

Fig 2: Actions in form of java procedures

III. RECURSIVE-DESCENT PARSING

Parsing expression grammar is essentially a top-down parsing strategy, and as such parsers are closely related to recursive descent parsers. First build a recursive descent parser for a language and then convert it into a parser.

Additive ← Multitive `+' Additive / Multitive
Multitive ← Primary `*' Multitive / Primary
Primary ← `(' Additive `)' / Decimal
Decimal ← `0' / …. / `9'

Figure 3: Grammar for a language

To construct a recursive-descent parser for this grammar, define four functions, one for each of the non terminals in the grammar. Each function takes the string to be parsed, attempts to recognize some prefix of the input string as a derivation of the corresponding non terminal, and returns either a “success" or “failure" result. Each function can recursively call itself and the other functions in order to recognize the non terminals appearing on the right-hand sides of its corresponding grammar rules. To implement this recursive-descent parser, the result of a parsing functions:

```
data Result v = Parsed v String
| NoParse
```

In this parser, each of the four parsing functions takes a String and produces a Result with a semantic value of type Int:

```
pAdditive :: String -> Result Int
pMultitive :: String -> Result Int
pPrimary :: String -> Result Int
pDecimal :: String -> Result Int
```

IV. INTEGRATED LEXICAL ANALYSIS

Bottom up parsing algorithms usually assume that the “raw" input text has already been partially digested by a separate lexical analyzer into a stream of tokens[6]. The parser then treats these tokens as atomic units even though each may represent multiple consecutive input characters. This separation is usually necessary because conventional linear-time

```
-- Additive <- Multitive AdditiveSuffix
pAdditive :: Derivs -> Result Int
pAdditive d = case dvMultitive d of
Parsed vl d' ->
case dvAdditiveSuffix d' of
Parsed f d'' ->
Parsed (f vl) d''
          _ -> NoParse
          _ -> NoParse
          -- AdditiveSuffix <-
          -- '+' Multitive AdditiveSuffix
          -- / '-' Multitive AdditiveSuffix
          -- / ( )
pAdditiveSuffix :: Derivs -> Result (Int -> Int)
pAdditiveSuffix d = alt1 where
-- Alternative 1: '+' Multitive AdditiveSuffix
alt1 = case dvChar d of
Parsed '+' d' ->
case dvMultitive d' of
Parsed vr d'' ->
case dvAdditiveSuffix d'' of
Parsed f d''' ->
Parsed (\vl -> f (vl + vr))
d'''
_ -> alt2
_ -> alt2
_ -> alt2
-- Alternative 2: '-' Multitive AdditiveSuffix
alt1 = case dvChar d of
Parsed '-' d' ->
case dvMultitive d' of
Parsed vr d'' ->
case dvAdditiveSuffix d'' of
Parsed f d''' ->
Parsed (\vl -> f (vl - vr))
d'''
_ -> alt3
_ -> alt3
_ -> alt3
-- Alternative 3: (empty string)
alt3 = Parsed (\v -> v) d
```

Figure 4: Packrat parsing functions for left-associative addition and subtraction

V. EVALUATING THE PARSING PROCESS

A. The grammar

Parsing Expression Grammar is a list of one or more ”rules” of the form:

name = expr ;

where *expr* is a parsing expression, and *name* is a name given to it. The *name* is a string of one or more letters (a-z, A-Z) and/or digits, starting with a letter. White space is allowed everywhere except inside names. Comments starting with a double slash and extending to the end of a line are also allowed. The order of the rules does not matter, except that the expression specified first is the ”top expression”, invoked at the start of the parser.

A specific grammar may look like this:

Example 1:

```
Sum = Number ("+" Number)* !_ ;
Number = [0-9]+ ;
```

It consists of two named expressions: Sum and Number. They define a parser consisting of two procedures named Sum and Number. The parser starts by invoking Sum. The Sum invokes Number, and if this succeeds, repeatedly invokes ("+" Number) as long as it succeeds. Finally, Sum invokes a sub-procedure for the predicate ”!_”, which succeeds only if it does not see any character ahead – that is, only at the end of input. The Number reads digits in the range from 0 through 9 as long as it succeeds, and is expected to find at least one such digit.

B. Parser

Example 1 generated Parser:

```
import madhavi.runtime.Source;
public class myParser extends
madhavi.runtime.ParserBase
{
 final madhavi.runtime.SemanticsBase sem;
  // Constructor
 public myParser()
 {
   sem = new madhavi.runtime.SemanticsBase();
   sem.rule = this;
   super.sem = sem;
 }
   // Run the parser
 public boolean parse(Source src)
 {
   super.init(src);
   sem.init();
   if (Sum()) return true;
   return failure();
 }
   // Get semantics
 public madhavi.runtime.SemanticsBase semantics()
 { return sem; }
 // Parsing procedures
// Sum = Number ("+" Number)* !_ ;
 private boolean Sum()
 {
   begin("Sum");
   if (!Number()) return reject();
   while (Sum_0());
   if (!aheadNot()) return reject();
   return accept();
 }
   // Sum_0 = "+" Number
 private boolean Sum_0()
 {
   begin("");
   if (!next('+')) return rejectInner();
   if (!Number()) return rejectInner();
   return acceptInner();
 }
   // Number = [0-9]+ ;
 private boolean Number()
 {
   begin("Number");
   if (!nextIn('0','9')) return reject();
   while (nextIn('0','9'));
   return accept();
 }
}
```

Result:
2 rules
1 unnamed
3 terminals

In above example class myParser is defined as a subclass of madhavi.runtime.ParserBase. The service methods are inherited from that superclass. The structure of parser is thus as follows:

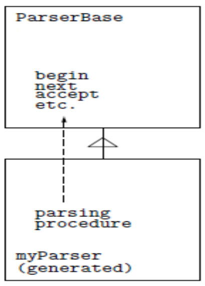


Fig 5: Service methods of ParserBase

C. Backtracking

The parser developed above is a backtracking parser. If alt1 in the pAdditive function fails, for example, then the parser effectively “backtracks" to the initial position, starting over with the original input string s in the second alternative alt2, regardless of whether the first alternative failed to match during its first, second, or third stage. The standard strategy for making top-down parsers practical is to design them so that they can \predict" which of several alternative rules to apply before actually making any recursive calls.

To illustrate backtracking, an alternative number format as:

```
Input = Space Sum !_ ;
Sum = Number (Plus Number)* ;
Number = Digits? "." Digits Space // fraction
/ Digits Space ; // integer
Plus = "+" Space ;
Digits = [0-9]+ ;
Space = " "* ;
```

The alternative format is a decimal fraction with or without digits before the decimal point. The definition of Number does not have the *LL*(1) property: both alternatives may start with Digits. Encountering a sequence of digits followed by a blank or plus,Number() starts with the first alternative, and calls Digits() that consumes all the digits and constructs a Phrase to represent them. Not finding the decimal point,Number() discards the Phrase, backtracks to where it started, and tries the second alternative that again calls Digits() to repeat the same job. This is the price for circumventing the *LL*(1) requirement. The backtracking activity by generating an instrumented version of the parser. This is done by specifying an option to the Generate utility. The instrumented parser uses the same semantics class as the ordinary one.The instrumented parser using another utility, TestParser. It produces this result for input “123 + 4567”:

```
49 calls: 32 ok, 15 failed, 2 backtracked.
11 rescanned.
backtrack length: max 4, average 3.5.
Backtracking, rescan, reuse:
```

```
procedure        ok     fail       back      resc      reuse
totbk    maxbk at
-------------      -----   -----    -----     -----    -----     --
---      ----- --
Digits             4      0          0          2        0         0
0
Number_0          0      0          2          0        0         7
4           After '123+'
[0-9]             14     4          0          9        0         0
0
```

The first three lines tell that to process the input “123 + 4567”, the parser executed 49 calls to parsing procedures, of which 32 succeeded, 15 failed, and two backtracked.
As expected, the parser backtracked 3 characters on the first Number and 4 on the second, so the maximum backtrack length was 4 and the average backtrack length was 3.5. You can also see that 11 of the procedure calls were “re-scans”: the same procedure called again at the same input position.

D. Using memorization

This is done by attaching to each parsing procedure a cache that can hold a small number [1-9] of the most recent Phrases created by the procedure. As each Phrase contains a pointer to the input text, the procedure may find that it already has the result, and directly return the Phrase. This function can be exercised by the instrumented parser by specifying, via an option, the size of the cache. Repeating the test from the preceding section with cache size 1
(that is, one most recent Phrase kept for each procedure) gives this result:

```
40 calls: 23 ok, 13 failed, 2 backtracked.
0 rescanned, 2 reused.
backtrack length: max 4, average 3.5.
Backtracking, rescan, reuse:

procedure        ok     fail    back    resc    reuse    totbk
maxbk at
-------------      -----    -----   -----   -----    -----      -----
----- --
Digits              2        0       0       0         2         0
0
Number_0          0        0      2       0         0         7
4         After '123+'
```

It shows that the parser reused the cached result of Digits on two occasions, thus
eliminating the unnecessary rescanning by [7-9].

VI. CONCLUSIONS

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*. 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. When deciding whether memoization or not, it introduces some overhead. It may cost more in performance than some moderate rescanning.

VII. REFERENCES

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

[2] 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.

[3] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Speci_cation, The 3rd Edi-tion*. Addison-Wesley, 2005. http://java.sun.com/docs/books/jls/thirdedition/html/j3TOC.html.

[4] 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.

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

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

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

[8] Redziejowski, R. R. Applying classical concepts to Parsing Expression Grammar. *Fundamenta Informaticae 93*, 1–3 (2009), 325–336.

[9] Mizushima, K., Maeda, A., and Yamaguchi, Y. Packrat parsers can handle practical grammars in mostly constant space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010* (2010), S. Lerner and A. Rountev, Eds., ACM, pp. 29–36.

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