

解析表达文法

在[计算机科学](#)领域，**解析表达文法**，简称**PEG**（英语：**P**arsing **E**xpression **G**rammar），是一种分析型**形式文法**。PEG在2004年由布莱恩·福特（Bryan Ford）推出，^[1]，它与20世纪70年代初引入的**自顶向下的语法分析语言**家族密切相关。

在语法上，PEG很接近**上下文无关文法**（CFG），但是他们采用了不同的解释：例如PEG中的选择操作符总是会选中第一个匹配项，而在CFG中则是不明确的。这更接近于字符串识别在实际中的应用，例如使用**递归下降解析器**的情况。另外不像CFG，PEG不能有**二义性**：在解析一个字符串的时候，只能有单一确定的语法**分析树**。据推测，存在上下文无关语言，不能用PEG处理，但尚未得到证实。^[1]这个特性使得PEG更适合计算机语言的解析，对于一般的CFG算法（如**依尔利算法**）的性能可以与之比拟的**自然语言**就不是很合适。^[2]

定义

语法

形式上，一个解析表达文法由以下部分组成：

- 一个有限的**非终结符**的集合 N
- 一个有限的**终结符**的集合 Σ ，和 N 没有交集
- 一个有限的解析规则的集合 P
- 一个被称作**开始表达式**的解析表达式 e_s

P 中的每一个解析规则以 $A \leftarrow e$ 的形式出现，这里 A 是一个非终结符， e 是一个**解析表达式**。解析表达式是类似**正则表达式**的层次表达式：

1. 原子解析表达式由以下组成:

- 任何的终结符，
- 任何的非终结符，
- 空字符串 ϵ .

2. 给定已经存在的解析表达式 e , e_1 和 e_2 , 一个新的解析表达式可以通过以下操作构成:

- 序列: $e_1 e_2$
- 有序选择: e_1 / e_2
- 零个或更多: e^*
- 一个或更多: e^+
- 可选: $e?$
- 肯定断言: $\&e$
- 否定断言: $!e$

语义

CFG和PEG的关键不同是PEG的选择操作符是**有序的**。如果第一个选项成功匹配，则忽略第二个选项。因此PEG的有序选择是不可**交换**的，这点和**上下文无关文法**或者正则表达式在教科书上的定义不同。有序选择类似于某些**逻辑编程**语言中的**软截断**操作符。

这导致的区别就是如果一个上下文无关文法被直接转换为解析表达文法，所有的不确定性的地方都会被确定下来，方法是从所有可能的解析树中选择一个分支。通过仔细安排文法可能项的顺序，编程的人就可以自由控制那一个解析分支被

选中。

与布尔上下文无关语法一样，解析表达式语法也添加了肯定断言以及否定断言。因为它们可以使用任意复杂的子表达式“向前查看”输入字符串，而不需要实际消耗它，所以它们提供了一个强大的语法向前查找和消歧功能，特别是在重新排序替代方案不能指定所需的准确的解析树时。

解析表达式的解释

解析表达文法里面的每一个非终结符本质上表示递归下降解析器里面的一个解析函数，其对应的解析表达式展示了这个函数包含的代码内容。概念上，每一个解析函数接受一个输入字符串作为参数，返回以下其中一个结果：

- 成功，函数可能向前移动或者“消耗”一个或多个输入字符串的字符
- 失败，不消耗任何字符

一个**非终结符**有可能成功但是不消耗任何输入字符，这也是一种不同于失败的结果。

只由一个**终结符**组成的原子解析表达式：成功，如果输入字符串的第一个字符就是定义中的终结符，这种情况下消耗这个输入字符；否之失败。由空字符串组成的原子解析表达式总是成功并且不消耗任何输入。只由一个非终结符A组成的原子解析表达式表示对非终结符A的解析函数的递归调用。

序列操作符 e_1e_2 首先调用 e_1 ，如果 e_1 成功，接着对 e_1 消耗剩下的输入字符串调用 e_2 ，最后返回结果。如果 e_1 或者 e_2 失败，那么序列表达式 e_1e_2 失败。

选择操作符 e_1 / e_2 首先调用 e_1 ，如果 e_1 成功，立刻返回结果。否则如果 e_1 失败，选择操作符回溯到输入字符串匹配 e_1 的原始位置，调用 e_2 ，最后返回 e_2 结果。

零个或多个，一个或多个，和**可选**操作符分别消耗零个或多个，一个或多个，或者零个或一个连续重复的子表达式e。与上下文无关文法和正则表达式不同的是，尽管如此，在PEG里这些操作符总是执行贪婪的行为，那就是消耗尽可能多的输入，而且绝对不回溯。（正则表达式一开始执行贪婪匹配，但是如果整个正则表达式失败后，会回退并尝试短一些的匹配。）例如，解析表达式 a^* 总是尽可能多的消耗输入字符串中连续出现的a，解析表达式 $(a^* a)$ 则必然会失败因为前半部分 a^* 绝对不会留下一丁点a给后半部分去匹配。

最后，肯定断言和否定断言实现了句法断言。 $\&e$ 表达式调用子表达式e，如果e成功，则返回成功；否则返回失败。无论结果如何都不消耗任何字符。反之，当e失败时 $!e$ 表达式成功，e成功时 $!e$ 表达式失败，同样无论结果如何都不消耗任何字符。因为向前判断的子表达式e可以任意的复杂，所以断言表达式提供了强大的句法向前判断和去除二义性的能力。

示例

这是一个简单的解析表达文法，它识别基本的数学表达式，只使用了基本的四个运算符并且只接受正整数作为操作数。

```
Expr    ← Sum
Sum      ← Product (('+' / '-') Product)*
Product ← Value (('*' / '/') Value)*
Value    ← [0-9]+ / '(' Expr ')'
```

在上面这个例子里面，终结符就是字符文本，用单引号括起来表示。比如 `'('` 和 `')'`。`[0-9]` 这个区间是10个字符的缩写，表示数字0到数字9里面的任意一个。（这里区间的语义和正则表达式里面的一样。）非终结符就是被定义成其他表达式的符号：*Value*, *Product*, *Sum*, 以及 *Expr*。

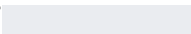
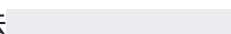
接下来的递归规则匹配了标准C风格的if/then/else语句。因为/操作符的隐式优先级安排，可选的else语句总是会被绑定到最内层的if语句。（在上下文无关文法里，这种结构的文法会导致悬空的else语句这种二义性错误。

```
S ← 'if' C 'then' S 'else' S / 'if' C 'then' S
```

下面的这个递归规则匹配Pascal格式的注释语法，(*和*)括号，其内部可以有嵌套的(*和*)对。注释符号放在双引号内内是为了与其他PEG操作符区分开来。

```
Begin ← '('  
End   ← ')'   
C     ← Begin N* End  
N     ← C / (!Begin !End Z)  
Z     ← any single character
```

解析表达式 **foo &(bar)** 只有当 foo 后面紧跟着字符串 bar 的时候，才会匹配并消耗 foo。而解析表达式 **foo !(bar)** 只有当 foo 后面没有紧跟着字符串 bar 的时候，才会匹配。表达式 **!(a+ b) a** 只有当a 不是一连串a后面连着一个b的情况下出现的时候，才能匹配一个单独的字母a。

解析表达式 **('a'/'b')*** 匹配任意长度的a和b序列。解析规则 **S ← 'a' S? 'b'** 描述了 。这样一个简单的上下文无关匹配语言。而下面的这个解析表达文法则可以描述经典的非上下文无关文法 ：

```
S ← &(A 'c') 'a'+ B !('a'/'b'/'c')  
A ← 'a' A? 'b'  
B ← 'b' B? 'c'
```

根据解析表达文法实现解析器

所有的解析表达文法都能够被直接转化为[递归下降解析器](#)。^[3] 尽管如此，因为PEG公式提供了理论上不受限制的向前检查的能力，所以最终得到的解析器还是可以避免最坏情况下指数级[时间复杂度](#)的。

通过保存增量解析步骤的结果和确保每一个解析函数在同一个输入位置只被调用一次，就可以把任意解析表达文法转化为一个Packrat Parser，可以实现线性的时间复杂度解析，其代价是足够大量的空间占用。

一个Packrat Parser^[3]是一种结构上类似于递归下降解析器的语法解析器，区别是在解析过程中，它会记下所有互相递归调用的函数的中间结果。因为保存了这些信息，一个 Packrat Parser就拥有了以线性时间复杂度解析多数上下文无关文法和所有解析表达文法的能力（包括某些表示的不是上下文无关文法语言的文法）。

从解析表达文法创建[LL分析器](#)和[LR分析器](#)也是可行的，但是在这两种情况下，不受限制的向前检查的能力就不能用了。

优势

因为PEG更加严格更加强大，PEG可以成为很好的[正则表达式](#)的替代品。例如，一个正则表达式本身是无法匹配嵌套的括号对，因为正则表达式不是递归的，但是PEG却能做到这点。

所有的PEG都能通过使用Packrat Parser达到线性时间解析，如同上文所述。

CFG表达的解析器，比如LR解析器，需要首先进行一个单独的断词步骤。这个步骤根据空白的位置或者发音等等因素把输入分成词。分词是必要的，因为 这类解析器使用向前检查来判断上下文无关文法是否匹配要求。PEG不需要单独的断

词步骤，断词的规则和其他语法规则可以同样的方式写在一起。

许多CFG固有的存在二义性，即使它们原本要描述的东西并不具有二义性。C, C++, Java里面著名的[悬空else问题](#)就是一个例子。这个问题通常都是应用文法之外的一个规则解决。而在PEG里面，因为使用了优先权，所以根本不存在这种问题。

劣势

PEG是新事物，还没有被广泛的应用。相比之下，正则表达式和CFG已经产生了数十年了，用来解析的代码也已经优化的很好，并且很多开发者都熟悉怎么使用他们。

PEG不能表达[左递归](#)的解析规则。例如，上面的数学运算文法，通过引入更多的规则，来使得乘法和加法的优先级能够在一行里面表达出来这可是非常的有诱惑力的，结果可能得到下面的文法：

```
Value ← [0-9.]+ / '(' Expr ')'  
Product ← Expr (('*' / '/') Expr)*  
Sum ← Expr (('+' / '-') Expr)*  
Expr ← Product / Sum / Value
```

这个文法的问题就是，为了匹配Expr，你需要首先判断是否某处匹配Product，而为了匹配Product，你又必须判断是不是此处匹配Expr。而这个[循环定义](#)是不可分析的。

尽管如此，我们总是可以通过重写左递归规则把左递归消除掉。[\[2\]\[4\]](#) 例如，一个左递归可能无限的重复一个规则，就像在CFG里面的：

```
string-of-a ← string-of-a 'a' | 'a'
```

在PEG里面，可以用加号操作符重写为：

```
string-of-a ← 'a'+
```

PEG还涉及到Packrat Parsing，一种通过占用更多的存储空间来消除不必要解析步骤的方法。Packrat Parsing需要的存储空间与总的输入文件的大小成正比，而不是LR解析器的与解析树的深度成正比。如果稍作修改，传统的Packrat Parser也可以支持左递归。

参见

- [形式文法](#)
- [正则表达式](#)
- [自顶向下解析的语言](#)
- [解析器-生成器比较](#)
- [解析器组合子](#)

参考文献

1. Ford, Bryan p. [Parsing Expression Grammars: A Recognition Based Syntactic Foundation](#)^[?]. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. [ACM](#). 2004 [2010-07-30].

ISBN 1-58113-729-X. doi:10.1145/964001.964011 [↗](#).

2. Bryan Ford. [Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time](#) [↗](#) (PDF). 2002.
3. Ford, Bryan. [Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking](#) [↗](#). Massachusetts Institute of Technology. September 2002 [2007-07-27].
4. Aho, A.V., Sethi, R. and Ullman, J.D. (1986) "Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA*.
- Medeiros, Sérgio; [Ierusalimsky, Roberto](#). A parsing machine for PEGs. Proc. of the 2008 symposium on Dynamic languages. *ACM*: article #2. 2008. ISBN 978-1-60558-270-2. doi:10.1145/1408681.1408683 [↗](#).

外部链接

- [Parsing Expression Grammars: A Recognition-Based Syntactic Foundation](#) [↗](#) (PDF slides)
- [Converting a string expression into a lambda expression using an expression parser](#) [↗](#)
- [The Packrat Parsing and Parsing Expression Grammars Page](#) [↗](#)
- [Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking](#) [↗](#)
- The [constructed language](#) [Lojban](#) has a [fairly large PEG grammar](#) [↗](#) allowing unambiguous parsing of Lojban text.