Parsing expression grammar

From Wikipedia, the free encyclopedia
(Redirected from Packrat parser)

In computer science, a **parsing expression grammar**, or **PEG**, is a type of analytic formal grammar, i.e. it describes a formal language in terms of a set of rules for recognizing strings in the language. The formalism was introduced by Bryan Ford in 2004[1] and is closely related to the family of top-down parsing languages introduced in the early 1970s. Syntactically, PEGs also look similar to context-free grammars (CFGs), but they have a different interpretation: the choice operator selects the first match in PEG, while it is ambiguous in CFG. This is closer to how string recognition tends to be done in practice, e.g. by a recursive descent parser.

Unlike CFGs, PEGs cannot be ambiguous; if a string parses, it has exactly one valid parse tree. It is conjectured that there exist context-free languages that cannot be parsed by a PEG, but this is not yet proven.[1] PEGs are well-suited to parsing computer languages, but not natural languages where their performance is comparable to general CFG algorithms such as the Earley algorithm.[2]

Contents [hide]

Definition [edit]

Syntax [edit]

Formally, a parsing expression grammar consists of:

- A finite set N of *nonterminal symbols*.
- A finite set Σ of *terminal symbols* that is disjoint from N.
- A finite set P of *parsing rules*.
- An expression e_S termed the *starting expression*.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a *parsing expression*. A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An *atomic parsing expression* consists of:
   - any terminal symbol,
   - any nonterminal symbol, or
   - the empty string ε.
2. Given any existing parsing expressions e, e_1, and e_2, a new parsing expression can be constructed using the following operators:
   - *Sequence*: $e_1\ e_2$
   - *Ordered choice*: e_1 / e_2
   - *Zero-or-more*: e*
   - *One-or-more*: e+

- *Optional*: *e*?
- *And-predicate*: &*e*
- *Not-predicate*: !*e*

Semantics [edit]

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is *ordered*. If the first alternative succeeds, the second alternative is ignored. Thus ordered choice is not commutative, unlike unordered choice as in context-free grammars. Ordered choice is analogous to soft cut operators available in some logic programming languages.

The consequence is that if a CFG is transliterated directly to a PEG, any ambiguity in the former is resolved by deterministically picking one parse tree from the possible parses. By carefully choosing the order in which the grammar alternatives are specified, a programmer has a great deal of control over which parse tree is selected.

Like boolean context-free grammars, parsing expression grammars also add the and- and not- syntactic predicates. Because they can use an arbitrarily complex sub-expression to "look ahead" into the input string without actually consuming it, they provide a powerful syntactic lookahead and disambiguation facility, in particular when reordering the alternatives cannot specify the exact parse tree desired.

Operational interpretation of parsing expressions [edit]

Each nonterminal in a parsing expression grammar essentially represents a parsing function in a recursive descent parser, and the corresponding parsing expression represents the "code" comprising the function. Each parsing function conceptually takes an input string as its argument, and yields one of the following results:

- *success*, in which the function may optionally move forward or *consume* one or more characters of the input string supplied to it, or
- *failure*, in which case no input is consumed.

An atomic parsing expression consisting of a single **terminal** (i.e. literal) succeeds if the first character of the input string matches that terminal, and in that case consumes the input character; otherwise the expression yields a failure result. An atomic parsing expression consisting of the empty string always trivially succeeds without consuming any input.

An atomic parsing expression consisting of a **nonterminal** *A* represents a recursive call to the nonterminal-function *A*. A nonterminal may succeed without actually consuming any input, and this is considered an outcome distinct from failure.

The **sequence** operator e_1 e_2 first invokes e_1, and if e_1 succeeds, subsequently invokes e_2 on the remainder of the input string left unconsumed by e_1, and returns the result. If either e_1 or e_2 fails, then the sequence expression e_1 e_2 fails.

The **choice** operator e_1 / e_2 first invokes e_1, and if e_1 succeeds, returns its result immediately. Otherwise, if e_1 fails, then the choice operator backtracks to the original input position at which it invoked e_1, but then calls e_2 instead, returning e_2's result.

The **zero-or-more**, **one-or-more**, and **optional** operators consume zero or more, one or more, or zero or one consecutive repetitions of their sub-expression *e*, respectively. Unlike in context-free grammars and regular expressions, however, these operators *always* behave greedily, consuming as much input as possible and never backtracking. (Regular expression matchers may start by matching greedily, but will then backtrack and try shorter matches if they fail to match.) For example, the expression a* will always consume as many a's as are consecutively available in the input string, and the expression (a* a) will always fail because the first part (a*) will never leave any a's for the second part to match.

The **and-predicate** expression &*e* invokes the sub-expression *e*, and then succeeds if *e* succeeds and fails if *e* fails, but in either case *never consumes any input*.

The **not-predicate** expression !*e* succeeds if *e* fails and fails if *e* succeeds, again consuming no input in either case.

Examples [edit]

This is a PEG that recognizes mathematical formulas that apply the basic four operations to non-negative integers.

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product)*
```

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

In the above example, the terminal symbols are characters of text, represented by characters in single quotes, such as `'('` and `')'`. The range `[0-9]` is also a shortcut for ten characters, indicating any one of the digits 0 through 9. (This range syntax is the same as the syntax used by regular expressions.) The nonterminal symbols are the ones that expand to other rules: *Value*, *Product*, *Sum*, and *Expr*.

The parsing expression **('a'/'b')*** matches and consumes an arbitrary-length sequence of a's and b's. The production rule ***S*** **← 'a'** ***S*****? 'b'** describes the simple context-free "matching language" $\{a^n b^n : n \geq 1\}$. The following parsing expression grammar describes the classic non-context-free language $\{a^n b^n c^n : n \geq 1\}$:

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

The following recursive rule matches standard C-style if/then/else statements in such a way that the optional "else" clause always binds to the innermost "if", because of the implicit prioritization of the '/' operator. (In a context-free grammar, this construct yields the classic dangling else ambiguity.)

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

The parsing expression **foo &(bar)** matches and consumes the text "foo" but only if it is followed by the text "bar". The parsing expression **foo !(bar)** matches the text "foo" but only if it is *not* followed by the text "bar". The expression **!(a+ b) a** matches a single "a" but only if it is not part of an arbitrarily long sequence of a's followed by a b.

The following recursive rule matches Pascal-style nested comment syntax, (* which can (* nest *) like this *). The comment symbols appear in single quotes to distinguish them from PEG operators.

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

Implementing parsers from parsing expression grammars [edit]

Any parsing expression grammar can be converted directly into a recursive descent parser.[3] Due to the unlimited lookahead capability that the grammar formalism provides, however, the resulting parser could exhibit exponential time performance in the worst case.

It is possible to obtain better performance for any parsing expression grammar by converting its recursive descent parser into a *packrat parser*, which always runs in linear time, at the cost of substantially greater storage space requirements. A packrat parser[3] is a form of parser similar to a recursive descent parser in construction, except that during the parsing process it memoizes the intermediate results of all invocations of the mutually recursive parsing functions, ensuring that each parsing function is only invoked at most once at a given input position. Because of this memoization, a packrat parser has the ability to parse many context-free grammars and *any* parsing expression grammar (including some that do not represent context-free languages) in linear time. Examples of memoized recursive descent parsers are known from at least as early as 1993.[4] Note that this analysis of the performance of a packrat parser assumes that enough memory is available to hold all of the memoized results; in practice, if there were not enough memory, some parsing functions might have to be invoked more than once at the same input position, and consequently the parser could take more than linear time.

It is also possible to build LL parsers and LR parsers from parsing expression grammars, with better worst-case performance than a recursive descent parser, but the unlimited lookahead capability of the grammar formalism is then lost. Therefore, not all languages that can be expressed using parsing expression grammars can be parsed by LL or LR parsers.

Advantages [edit]

Compared to pure regular expressions (i.e. without back-references), PEGs are strictly more powerful, but require significantly more memory. For example, a regular expression inherently cannot find an arbitrary number of matched pairs of parentheses, because it is not recursive, but a PEG can. However, a PEG will require an amount of memory proportional to the length of the input, while a regular expression matcher will require only a constant amount of memory.

Any PEG can be parsed in linear time by using a packrat parser, as described above.

Parsers for languages expressed as a CFG, such as LR parsers, require a separate tokenization step to be done first, which breaks up the input based on the location of spaces, punctuation, etc. The tokenization is necessary because of the way these parsers use *lookahead* to parse CFGs that meet certain requirements in linear time.[*citation needed*] PEGs do not require tokenization to be a separate step, and tokenization rules can be written in the same way as any other grammar rule.

Many CFGs contain ambiguities, even when they're intended to describe unambiguous languages. The "dangling else" problem in C, C++, and Java is one example. These problems are often resolved by applying a rule outside of the grammar. In a PEG, these ambiguities never arise, because of prioritization.

Disadvantages [edit]

Memory consumption [edit]

PEG parsing is typically carried out via *packrat parsing*, which uses memoization[5][6] to eliminate redundant parsing steps. Packrat parsing requires storage proportional to the total input size, rather than the depth of the parse tree as with LR parsers. This is a significant difference in many domains: for example, hand-written source code has an effectively constant expression nesting depth independent of the length of the program—expressions nested beyond a certain depth tend to get refactored.

For some grammars and some inputs, the depth of the parse tree can be proportional to the input size,[7] so both an LR parser and a packrat parser will appear to have the same worst-case asymptotic performance. A more accurate analysis would take the depth of the parse tree into account separately from the input size. This is similar to a situation which arises in graph algorithms: the Bellman–Ford algorithm and Floyd–Warshall algorithm appear to have the same running time ($O(|V|^3)$) if only the number of vertices is considered. However, a more precise analysis which accounts for the number of edges as a separate parameter assigns the Bellman–Ford algorithm a time of $O(|V| * |E|)$, which is only quadratic[*dubious – discuss*] in the size of the input (rather than cubic).

Indirect left recursion [edit]

PEGs cannot express *left-recursive* rules where a rule refers to itself without moving forward in the string. For example, in the arithmetic grammar above, it would be tempting to move some rules around so that the precedence order of products and sums could be expressed in one line:

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

In this new grammar matching an `Expr` requires testing if a `Product` matches while matching a `Product` requires testing if an `Expr` matches. This circular definition cannot be resolved. However, left-recursive rules can always be rewritten to eliminate left-recursion.[2][8] For example, the following left-recursive CFG rule:

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

can be rewritten in a PEG using the plus operator:

```
string-of-a ← 'a'+
```

The process of rewriting *indirectly* left-recursive rules is complex in some packrat parsers, especially when semantic actions are involved.

With some modification, traditional packrat parsing can support direct left recursion,[3][9][10] but doing so results in a loss of the linear-time parsing property[9] which is generally the justification for using PEGs and packrat parsing in the first place. Only the OMeta parsing algorithm[11] supports full direct and indirect left recursion without additional attendant complexity (but again, at a loss of the linear time complexity), whereas all GLR parsers support left recursion.

Expressive power [edit]

PEG packrat parsers cannot recognize some unambiguous nondeterministic CFG rules, such as the following:[2]

```
S ← 'x' S 'x' | 'x'
```

Neither LL(k) nor LR(k) parsing algorithms are capable of recognizing this example. However, this language is parsable by a general CFG parser like the CYK algorithm.

See also [edit]

- Formal grammar
- Regular expressions
- Top-down parsing language
- Comparison of parser generators
- Parser combinators

References [edit]

1. ^ *a* *b* Ford, Bryan p (2004). "Parsing Expression Grammars: A Recognition Based Syntactic Foundation". *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. doi:10.1145/964001.964011. ISBN 1-58113-729-X. Retrieved 2010-07-30.
2. ^ *a* *b* *c* Bryan Ford (2002). "Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time" (PDF).
3. ^ *a* *b* *c* Ford, Bryan (September 2002). "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking". Massachusetts Institute of Technology. Retrieved 2007-07-27.
4. ^ Merritt, Doug (November 1993). "Transparent Recursive Descent". Usenet group comp.compilers. Retrieved 2009-09-04.
5. ^ Bryan Ford. "The Packrat Parsing and Parsing Expression Grammars Page". Retrieved 23 Nov 2010.
6. ^ Rick Jelliffe. "What is a Packrat Parser? What are Brzozowski Derivatives?". Retrieved 23 Nov 2010.
7. ^ for example, the LISP expression (x (x (x (x ....))))
8. ^ Aho, A.V., Sethi, R. and Ullman, J.D. (1986) "Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.*
9. ^ *a* *b* Alessandro Warth, James R. Douglass, Todd Millstein (January 2008). "Packrat Parsers Can Support Left Recursion" (PDF). Viewpoints Research Institute. Retrieved 2008-10-02.
10. ^ Ruedi Steinmann (March 2009). "Handling Left Recursion in Packrat Parsers" (PDF).
11. ^ "Packrat Parsers Can Support Left Recursion" (PDF). PEPM '08. January 2008. Retrieved 2009-08-04.

- Medeiros, Sérgio; Ierusalimschy, Roberto (2008). "A parsing machine for PEGs". *Proc. of the 2008 symposium on Dynamic languages*. ACM. pp. article #2. doi:10.1145/1408681.1408683. ISBN 978-1-60558-270-2.

External links [edit]

This article's **use of external links may not follow Wikipedia's policies or guidelines**. Please improve this article by removing excessive or inappropriate external links, and converting useful links where appropriate into footnote references. *(September 2011)*

- 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.

Categories: Formal languages | Parsing algorithms