

Parsing Expression Grammars

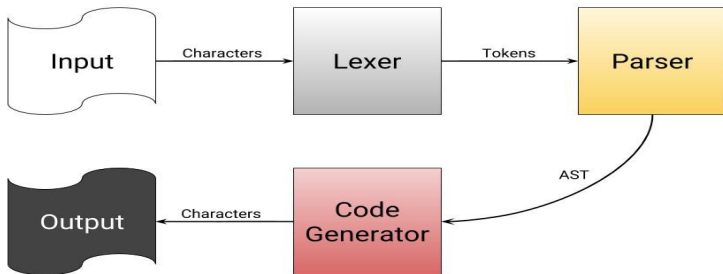
July 9, 2021

Università degli Studi di Pisa
Anno Accademico 2020-21

Parsing

What is a parser?

Part of the compiler. Checks the stream of tokens produced by the *lexer* for syntactical errors and produces an intermediate representation (usually an abstract syntax tree) of the source that is used in later steps to generate machine code.



How do we represent the syntax of a programming language?

Classical problem, solved by using formal language theory.

Regular Expressions and **Context Free Grammars** are the main models.

CFG: definition

Context free grammar defined as:

$$G = (V, \Sigma, P, S)$$

- V = nonterminals
- Σ = terminals
- P = productions (or rules)
- S = start symbol

Productions are in the form $A \rightarrow \alpha$, where $A \in V$, $\alpha \in (V \cup \Sigma)^*$

$S \Rightarrow A \mid bb$

$A \Rightarrow B \mid b$

$B \Rightarrow S \mid a$

Given the rules of grammar G , we want to find a sequence of productions that generates a target expression (**derivation**).

Parsing is the process of discovering such sequence.

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \textit{expression}$$

Leftmost derivation: at each step expand the leftmost non-terminal symbol

Rightmost derivation: at each step expand the rightmost non-terminal symbol

1	<i>Expr</i>	→	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	→	<i>Term</i> × <i>Factor</i>
5			<i>Term</i> ÷ <i>Factor</i>
6			<i>Factor</i>
7	<i>Factor</i>	→	(<i>Expr</i>)
8			num
9			ident

Figure 1: classic grammar for arithmetic expressions

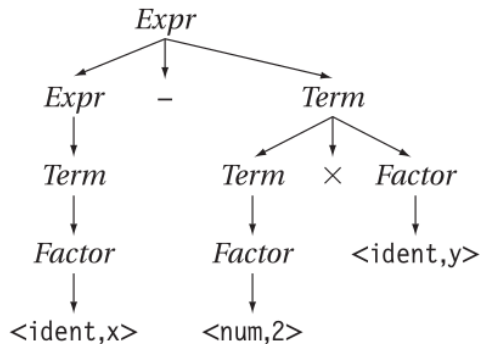


Figure 2: parse tree for the expression $x-2*y$

Parsing techniques

Two main techniques:

- Top Down (recursive descent)
Builds the parse tree from root to leaves. Can be modified to do predictive parsing (LL(1) parser).
- Bottom Up
LR(1) parsers, build the parse tree from leaves to root. Bottom-up parsing handles a larger set of grammars

We will focus on top down parsers.

Top down parsers


Start from root of parse tree

At each level pick a production to match the input

If the derivation doesn't match the input \Rightarrow backtrack and pick a different production

Issues

- Backtracking causes parsing to be exponential in time complexity
- Left recursion

$E \rightarrow E + T$ $ E - T$ $ T$ $T \rightarrow T * F$ $ T / F$ $ F$		$E \rightarrow TE'$ $E' \rightarrow +TE'$ $ -TE'$ $ \varepsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT'$ $ /FT'$ $ \varepsilon$
--	---	--

- Ambiguity

If ex1 then if ex2 then s1 else s2

Parsing Expression Grammars: an introduction

Introduced by Bryan Ford in 2004 [1].

An alternative, recognition based, formal foundation for language syntax.

Similar approach to EBNF notation, where a CFG is enriched with RE-like features.

A key difference

The nondeterministic choice operator '|' is substituted by a *prioritized* choice operator '/'.

$$\begin{array}{c} A \rightarrow ab \mid a \\ \Updownarrow \\ A \rightarrow a \mid ab \end{array}$$

$$\begin{array}{c} A \leftarrow ab / a \\ \Updownarrow \\ A \leftarrow a / ab \end{array}$$

Operators

A parsing expression grammar consists of a set of definition of the form ' $A \leftarrow e$ ', where A is a nonterminal and e is a *parsing expression*.

A parsing expression can be constructed using the operators defined in Table 1.

Operator	Type	Precedence	Description
' '	Primary	5	Literal String
" "	Primary	5	Literal String
[]	Primary	5	Character class
.	Primary	5	Any character
(e)	Primary	5	Grouping
e?	Unary suffix	4	Optional
e*	Unary suffix	4	Zero or more
e+	Unary suffix	4	One or more
&e	Unary prefix	3	And-predicate
!e	Unary prefix	3	Not-predicate
e1 e2	Binary	2	Sequence
e1 / e2	Binary	1	Prioritized choice

Table 1: Operators for constructing parsing expression

' ' and " " delimit string literals, while [] indicate characters classes, which can also be specified using ranges such as 'a-z' . The ' . ' constant matches any character.

?, * and + work as in normal regular expression, but they are greedy instead of non deterministic.

& and ! are *syntactic predicates*¹. The expression '&e' tries to match the given pattern and then unconditionally backtracks to the starting point, maintaining the knowledge of whether e succeeded or failed without consuming any input. '!e' succeed if e fails and viceversa. These two operators are fundamental for the expressive power of Parsing Expression Grammars, allowing them to describe languages that are not even parsed using CFGs (e.g $a^n b^n c^n$).

¹See [2]

Peg describing its own ASCII syntax

Hierarchical syntax

Grammar \leftarrow Space Def⁺ EOF
Def \leftarrow LArrow Expr
Expr \leftarrow Seq (Slash Seq)*
Seq \leftarrow Prefix*

Prefix \leftarrow (And / Not)? Suffix
Suffix \leftarrow Primary (Question / Star / Plus)?
Primary \leftarrow Identifier !LArrow
/ Open Expr Close
/ Literal
/ Class / Dot

Lexical syntax

Identifier \leftarrow Istart Icont* Space
Istart \leftarrow [a-zA-Z]
Icont \leftarrow Istart / [0-9]
Literal \leftarrow ['](!['] Char)* ['] Space
/ ["](!["] Char)* ["] Space
Class \leftarrow '[' (![' Range)* ']' Space
Range \leftarrow Char '-' Char
/ Char
Char \leftarrow '\\\' [nrt"' \ [\] \\]
/ '\\\' [0-2][0-7][0-7]
/ '\\\' [0-2][0-7]?
/ '!\\'

Larrow \leftarrow '←' Space
Slash \leftarrow '/' Space
And \leftarrow '&' Space
Not \leftarrow '!' Space
Question \leftarrow '?' Space
Star \leftarrow '*' Space
Plus \leftarrow '+' Space
Open \leftarrow '(' Space
Close \leftarrow ')' Space
Dot \leftarrow '.' Space
Space \leftarrow (Ws / Comment)*
Comment \leftarrow '#' (!EOL .)* EOL
Space \leftarrow ' ' / '\t' /
EOL \leftarrow '\r\n' / '\n' / '\r'
EOF \leftarrow ! .

Unified language definition

The large majority of syntax description uses a context free grammar to specify the hierarchical structure and a set of regular expressions that specify the individual lexical elements.

Context Free Grammars cannot express all idioms, such as the greedy rule for identifiers and numbers, or "negative" syntax to described quoted string literals.

Regular Expressions cannot describe recursive syntax.

Using both allows us to surpass the limitations.

These issues are non existent in PEGs, thanks to its operators.

The greedy nature of the repetition operator ensures that a sequence of letters is always interpreted as a single identifier. Negative syntax can be described thanks to the ! operators, as seen in the Char or Class definitions.

Lexical elements definition can refer to the hierarchical portion of the syntax

$$\text{Comment} \leftarrow ' (* ' (\text{Comment} / ! ' *) ' .) * ' *) '$$

This allows us to use a single, unified model to concisely express a machine-oriented language.

Handling ambiguity

Ambiguity is usually resolved in CFGs by using informal meta-rules (e.g, for the dangling else problems, we assume that an if without else is prioritized)

Thanks to prioritized choice, repetition operators and syntactic predicates, many ambiguities can be entirely avoided in PEGs

Statement \leftarrow IF Cond THEN Statement ELSE Statement
/ IF Cond THEN Statement / ...

Left recursion is still unavailable in PEGs. Because of the enforced priority, a rule such as

$$A \leftarrow A a \mid a$$

causes an infinite loop (can be avoided using repetition operators).

A parsing expression grammar is still a purely syntactic formalism. As such, it cannot fully express languages whose syntax depends on semantics predicates (e.g typedef identifiers in C) .

Formal analysis of PEGs

The definition of PEGs that we saw is very concrete and implementation oriented.

To reason about grammar and languages properties, we need to abstract away from the unessential details and retain only the basic structure.

Parsing Expression Grammar

$$G = (V_n, V_t, R, e_S)$$

- V_n set of nonterminal symbols
- V_t set of terminal symbols
- R finite set of rules
- e_S the *start expression*

$$V_n \cap V_t = \emptyset$$

$r \in R$ is a pair (A, e) , written $A \leftarrow e$, where $A \in V_n$ and e is a parsing expression

R is a function, meaning that for any nonterminal $A \in V_n$ there is one and only one e such that $A \leftarrow e \in R$. We identify with $R(A)$ the expression e associated with A .

R being functional prevents expression from containing undefined references or subroutine failures.

Parsing expression are defined inductively.

It is a parsing expression:

- The empty string ϵ .
- Any terminal $a \in V_t$.
- Any non terminal $A \in V_n$.
- The sequence e_1e_2 , if e_1 and e_2 are parsing expression.
- The choice e_1/e_2 , if e_1 and e_2 are parsing expression.
- The repetition e^* , if e is a parsing expression.
- The not-predicate $!e$, if e is a parsing expression.

$E(G)$: a set containing e_S , the expressions used in R and all their subexpressions.

Repetition free grammar: grammar where no expression contains the * operator.

Predicate free grammar: grammar where no expression contains the ! operator.

Desugaring operators

The abstract syntax does not define ' . ', ' + ', ' ? ', ' & ' and character classes. These constructs are desugared with the following rules:

- $\cdot \rightarrow [a] \ \forall a \in V_T.$
- $[a_1, a_2, \dots, a_n] \rightarrow a_1/a_2/\dots/a_n.$
- $e? \rightarrow e_d/\epsilon$ where e_d is the desugaring of e .
- $e^+ \rightarrow e_d e_d^*.$
- $\&e \rightarrow !(e).$

We formalize the meaning of a grammar G defining the relation \Rightarrow_G .

\Rightarrow_G goes from pairs (e, x) to pairs (n, o) , where e is a parsing expression, $x \in V_t^*$ is an input string, $n \geq 0$ is a "step counter" and $o \in V_T^* \cup f$ is the result of a recognition attempt (f represents a failure).

\Rightarrow_G defined inductively.

- $(\epsilon, x) \Rightarrow (1, \epsilon) \forall x \in V_t^*$.
- $(a, ax) \Rightarrow (1, a)$ if $A \in V_t, x \in V_t^*$.
- $(a, bx) \Rightarrow (1, f)$ if $a \neq b$ and $(a, \epsilon) \Rightarrow (1, f)$.
- $(A, x) \Rightarrow (n+1, o)$ if $A \Rightarrow e \in R$ and $(e, x) \Rightarrow (n, o)$.
- If $(e_1, x_1x_2y) \Rightarrow (n_1, x_1)$ and $(e_2, x_2y) \Rightarrow (n_2, x_2)$, then $(e_1e_2, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2)$.
- If $(e_1, x) \Rightarrow (n_1, f)$, then $(e_1e_2, x) \Rightarrow (n_1 + 1, f)$.
- If $(e_1, x_1y) \Rightarrow (n_1, x_1)$ and $(e_2, y) \Rightarrow (n_2, f)$, then $(e_1e_2, x_1y) \Rightarrow (n_1 + n_2 + 1, f)$.
- If $(e_1, xy) \Rightarrow (n_1, x)$, then $(e_1/e_2, xy) \Rightarrow (n_1, x)$.
- If $(e_1, x) \Rightarrow (n_1, f)$ and $(e_2, x) \Rightarrow (n_2, o)$, then $(e_1/e_2, x) \Rightarrow (n_1 + n_2 + 1, o)$.
- If $(e, x_1x_2y) \Rightarrow (n_1, x_1)$ and $(e^*, x_2y) \Rightarrow (n_2, x_2)$, then $(e^*, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2)$.
- If $(e, x) \Rightarrow (n_1, f)$, then $(e^*, x) \Rightarrow (n_1 + 1, \epsilon)$.
- If $(e, xy) \Rightarrow (n, x)$, then $(!e, xy) \Rightarrow (n + 1, f)$.
- If $(e, x) \Rightarrow (n, f)$, then $(!e, x) \Rightarrow (n + 1, f)$.

Languages and grammars properties

A language L is a parsing expression language (PEL) iff there exists a parsing expression grammar G whose language is L .

Theorem

The class of parsing expression languages is closed under intersection, union, and complement.

Proof

Suppose that we have $G_1 = (V_N^1, V_T, R^1, e_S^1)$, $G_2 = (V_N^2, V_T, R^2, e_S^2)$ that recognizes $L(G_1)$ and $L(G_2)$. Assuming that $V_N^1 \cap V_N^2 = \emptyset$, we define the grammar $G = (V_N^1 \cup V_N^2, V_T, R^1 \cup R^2, e'_S)$ where e'_S is one of the following:

- If $e'_S = e_S^1 / e_S^2$, then $L(G) = L(G_1) \cup L(G_2)$
- If $e'_S = \&e_S^1 e_S^2$, then $L(G) = L(G_1) \cap L(G_2)$
- If $e'_S = !e_S^1$, then $L(G) = V_T^* - L(G_1)$

The class of parsing expression includes non context-free languages. As an example, the language $a^n b^n c^n$ can be recognized using a PEG $G = (\{A, B, D\}, \{a, b, c\}, R, D)$ where R is composed by the following rules:

$$A \leftarrow aAb/\epsilon$$

$$B \leftarrow bBc/\epsilon$$

$$D \leftarrow \&(A!b)a^*B!.$$

It is undecidable whether the language $L(G)$ of an arbitrary grammar G is empty. Because of this, it is also undecidable the problem of knowing if a grammar G is complete.

We call a grammar *well formed* if it does not have left recursive rules.

Grammar identities

$$e_1(e_2 e_3) \asymp (e_1 e_2) e_3$$

$$e_1(e_2/e_3) \asymp (e_1/e_2) e_3$$

$$e_1(e_2/e_3) \asymp (e_1/e_2)/(e_1 e_3)$$

$$e_1(e_2/e_3) \not\asymp (e_1/e_3)/(e_2 e_3)$$

$$e_1 ! e_2 \asymp ! (e_1 e_2) e_1$$

e_1 and e_2 are disjoint if they succeed on disjoint sets of input strings.

$e_1/e_2 \asymp e_2/e_1$ if e_1 and e_2 are disjoint.

PEGs can be reduced to simpler forms, which may be more useful to implement and easier to reasonate about.

Eliminating repetition operators

Any repetition expression e^* can be replaced with a new nonterminal A with the definition $A \leftarrow eA/\epsilon$. This new rule must be right recursive.

Eliminating predicates

It is possible to remove all predicate operators from a well formed PEG. The procedure goes beyond the scope of this seminar so we will not see it, but it is shown by Ford in the original paper [1].

Reduction to TDPL

Any predicate free PEG can be reduced to a restricted system introduced by Birman in 1970 [3] and renamed "Top Down Parsing Language (TPDL)" by Aho and Ullman [4].

TPDL can be seen as the PEG equivalent of CNF for context free grammars. A TDPL grammar is a PEG $G = (V_N, V_T, R, S)$ where $S \in V_N$ and the rules in R can have only the following form:

1. $A \leftarrow \epsilon$
2. $A \leftarrow a$, where $a \in V_T$.
3. $A \leftarrow f$, where $f \equiv !\epsilon$.
4. $A \leftarrow BC/D$, where $B, C, D \in V_N$.

Any predicate-free PEG $G = (V_N, V_T, R, e_S)$ can be reduced to an equivalent TDPL $G' = (V'_N, V_T, R', S)$.

We first add the nonterminal S and the rule $S \leftarrow e_S$. Then we add two nonterminals E and F and their rules $E \leftarrow \epsilon$ and $F \leftarrow f$. Finally we rewrite the non-TDPL rules in the following way:

- $A \leftarrow B \mapsto A \leftarrow BE/F$
- $A \leftarrow e_1 e_2 \mapsto A \leftarrow BC/F, B \leftarrow e_1, C \leftarrow e_2.$
- $A \leftarrow e_1/e_2 \mapsto A \leftarrow BE/C, B \leftarrow e_1, C \leftarrow e_2.$
- $A \leftarrow e^* \mapsto A \leftarrow BA/E, B \leftarrow e.$

Reduction to GTDPL

Similar reasoning can be applied to reduce a PEG to a "Generalized Top Down Parsing Language (GTDPL), which utilizes slightly different rules that essentially provides the functionality of predicates in PEGs.

1. $A \leftarrow \epsilon$
2. $A \leftarrow a$, where $a \in V_T$.
3. $A \leftarrow f$, where $f \equiv !\epsilon$.
4. $A \leftarrow B[C, D]$, where $B[C, D] \equiv BC/!BD$, and $B, C, D \in V_N$.

Any predicate-free PEG $G = (V_N, V_T, R, e_S)$ can be reduced to an equivalent TDPL $G' = (V'_N, V_T, R', S)$.

We first add the definitions $S \leftarrow e_S$, $E \leftarrow \epsilon$ and $F \leftarrow f$. We rewrite the non-GTDPL rules in the following way:

- $A \leftarrow B \mapsto A \leftarrow B[E, F]$
- $A \leftarrow e_1 e_2 \mapsto A \leftarrow B[C, F], B \leftarrow e_1, C \leftarrow e_2.$
- $A \leftarrow e_1 / e_2 \mapsto A \leftarrow B[E, C], B \leftarrow e_1, C \leftarrow e_2.$
- $A \leftarrow e^* \mapsto A \leftarrow B[A, E], B \leftarrow e.$
- $A \leftarrow !e \mapsto A \leftarrow B[F, E], B \leftarrow e.$

Any well formed GTDPL grammar that does not accept the empty string can be reduced to an equivalent TDPL grammar.

These reductions allows use the tabular parsing technique described by Aho and Ullman [4], meaning that we can construct a linear time parser for any given PEG.

The tabular parsing technique has been extended and improved by Ford to produce what is called *packrat parsing*[5]. It essentially can be seen as a lazy version of the tabular algorithm. Intermediate results are stored in a table, so we can lazily reuse them when backtracking. It makes top down parsing linear in time at the cost of using more space.

A use case: pgen

While being a relatively new subject, parsing expression grammars have recently proved their power in concrete parsers implementation.

In PEP 617 [6], CPython creator and maintainer Guido Van Rossum announced the proposal of replacing the old python parser generator, named pgen, with a new PEG parser generator.

The proposal has been accepted, and the new module has been introduced in python 3.9. The old parser will still be the main Python parser until version 3.10, where it will be completely replaced.

Python parser: an overview

Despite technically being LL(1), the Python grammar has several rules that are not LL(1), requiring several workarounds to be used in the grammar and other parts of CPython.

```
namedexpr_test: test [':= ' test]
```

Sometimes the problem is not solvable with any workaround. As an example, no LL(1) rule can be made to support writing multiline parenthesized context managers:

```
with(  
    open("a") as a,  
    open("b") as b,  
    open("c") as c,  
)
```

Since open parenthesis are in the first sets of grammar items that can appear as context managers, the rule would be ambiguous.

- Huge coupling between AST generation routines and the shape of the resulting tree. Many actions are directly tied to the implicit structure of the ast, making the code more complex and implementation dependent.
- No left recursion allowed, since the grammar is LL(1).
- The current parser does not directly generate the AST. It instead creates a Concrete Syntax Tree, which is then transformed to an abstract one. This structure is not used by anything else and it requires to be kept entirely in memory, heavily increasing space consumption.

The proposal

The new proposed PEG parser is divided in three components:

1. A parser generator. It reads a grammar file and produces a PEG parser written in Python or C.
2. A PEG meta-grammar that auto generates a Python parser used for the parser generator itself.
3. A generated parser that produces C and Python AST objects.

Left recursion

While PEG parsers normally do not support it, the proposed parser can handle both direct and indirect left recursion, thanks to a memoization cache.

Syntax

The syntax is quite similar to the one that we saw. For simplicity the left arrow \leftarrow is replaced by ' : ' and the choice operator reuses the symbol ' | '.

Grammar actions

The proposed PEG parser is able to directly generate AST nodes for a rule via grammar actions, which are language specific expressions evaluated when a rule is successfully parsed. This removes the need for an intermediate CST, freeing up space.

During testing, it has been shown that the new parser comes within the performance of the old one up to 10%, both in time and space consumption.

While the packrat parsing algorithm requires more space of the normal LL(1) parser to store intermediate results, this is balanced by not having to build the intermediate CST.

It has been found that the new parser is slightly faster, but uses around 10% more memory.

- [PegJS](#)
- [PEGTL](#)
- [Rust-peg](#)

Thanks for your attention!

```
>>> __peg_parser__
```

```
File "<stdin>", line 1
```

```
__peg_parser__
```

```
^
```

```
SyntaxError: You found it!
```

- [1] Bryan Ford.
Parsing expression grammars: A recognition-based syntactic foundation.
SIGPLAN Not., 39(1):111–122, January 2004.
- [2] Terence J Parr and Russell W Quong.
Adding semantic and syntactic predicates to ll (k): pred-ll (k).
In *International Conference on Compiler Construction*, pages 263–277. Springer, 1994.
- [3] Alexander Birman.
The TMG Recognition in Schema.
PhD thesis, Citeseer, 1970.

- [4] Alfred V. Aho and Jeffrey D. Ullman.
The Theory of Parsing, Translation, and Compiling.
Prentice-Hall, Inc., USA, 1972.
- [5] Bryan Ford.
Packrat parsing: simple, powerful, lazy, linear time, functional pearl.
ACM SIGPLAN Notices, 37(9):36–47, 2002.
- [6] Guido van Rossum.
Pep 617 – new peg parser for cpython.
2020.