# The language of languages

[article index] [email me] [@mattmight] [rss]

Languages form the terrain of computing.

Programming languages, protocol specifications, query languages, file formats, pattern languages, memory layouts, formal languages, config files, mark-up languages, formatting languages and meta-languages shape the way we compute.

So, what shapes languages?

Grammars do.

Grammars are the language of languages.

Behind every language, there is a grammar that determines its structure.

This article explains grammars and common notations for grammars, such as Backus-Naur Form (BNF), Extended Backus-Naur Form (EBNF) and regular extensions to BNF.

After reading this article, you will be able to identify and interpret all commonly used notation for grammars.

## Defining a language

A grammar defines a language.

In computer science, the most common type of grammar is the context-free grammar, and these grammars will be the primary focus of this article.

Context-free grammars have sufficient richness to describe the recursive syntactic structure of many (though certainly not all) languages.

I'll discuss grammars beyond context-free at the end.

## Components of a context-free grammar

A set of rules is the core component of a grammar.

Each rule has two parts: (1) a name and (2) an expansion of the name.

For instance, if we were creating a grammar to handle english text, we might add a rule like:

*noun-phrase* may expand into *article noun*.

from which we could ultimately deduce that "the dog" is a *noun-phrase*.

Or, if we were describing a programming language, we could add a rule like:

*expression* may expand into *expression* + *expression*

If we're working with grammars as mathematical objects, then instead of writing "may expand into," we'd simply write $\rightarrow$:

$$\textit{noun-phrase} \rightarrow \textit{article noun}$$
$$\textit{expression} \rightarrow \textit{expression} + \textit{expression}$$

As an example, consider the classic unambiguous expression grammar:

$$\begin{aligned}
\textit{expr} &\rightarrow \textit{term} + \textit{expr} \\
\textit{expr} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow \texttt{(}\ \textit{expr}\ \texttt{)} \\
\textit{factor} &\rightarrow \textit{const} \\
\textit{const} &\rightarrow \textit{integer}
\end{aligned}$$

So, how do we know that `3 * 7` is a valid expression?

Because:

*expr* may expand into *term*;
which may expand into *term* * *factor*;
which may expand into *factor* * *factor*;
which may expand into *const* * *factor*;
which may expand into *const* * *const*;
which may expand into `3` * *const*;
which may expand into `3 * 7`.

# Backus-Naur Form (BNF) notation

When describing languages, Backus-Naur form (BNF) is a formal notation for encoding grammars intended for human consumption.

Many programming languages, protocols or formats have a BNF description in their specification.

Every rule in Backus-Naur form has the following structure:

$$\textit{name}\ ::=\ \textit{expansion}$$

The symbol `::=` means "may expand into" and "may be replaced with."

In some texts, a *name* is also called a *non-terminal symbol*.

Every *name* in Backus-Naur form is surrounded by angle brackets, `< >`, whether it appears on the left- or right-hand side of the rule.

An *expansion* is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and choice.

A terminal symbol is a literal like (`"+"` or `"function"`) or a class of literals (like `integer`).

Simply juxtaposing expressions indicates sequencing.

A vertical bar `|` indicates choice.

For example, in BNF, the classic expression grammar is:

```
<expr> ::= <term> "+" <expr>
        |  <term>

<term> ::= <factor> "*" <term>
        |  <factor>

<factor> ::= "(" <expr> ")"
          |  <const>

<const> ::= integer
```

Naturally, we can define a grammar for rules in BNF:

$$\begin{gathered} \textit{rule} \rightarrow \textit{name}\ \texttt{::=}\ \textit{expansion} \\ \textit{name} \rightarrow \texttt{<}\ \textit{identifier}\ \texttt{>} \\ \textit{expansion} \rightarrow \textit{expansion}\ \textit{expansion} \\ \textit{expansion} \rightarrow \textit{expansion}\ \texttt{|}\ \textit{expansion} \\ \textit{expansion} \rightarrow \textit{name} \\ \textit{expansion} \rightarrow \textit{terminal} \end{gathered}$$

We might define identifiers as using the regular expression `[-A-Za-z_0-9]+`.

A terminal could be a quoted literal (like `"+"`, `"switch"` or `"<<="`) or the name of a class of literals (like `integer`).

The name of a class of literals is usually defined by other means, such as a regular expression or even prose.

# Extended BNF (EBNF) notation

Extended Backus-Naur form (EBNF) is a collection of extensions to Backus-Naur form.

Not all of these are strictly a superset, as some change the rule-definition relation `::=` to `=`, while others remove the angled brackets from non-terminals.

More important than the minor syntactic differences between the forms of EBNF are the additional operations it allows in expansions.

### Option

In EBNF, square brackets around an expansion, `[ expansion ]`, indicates that this expansion is optional.

For example, the rule:

```
<term> ::= [ "-" ] <factor>
```

allows factors to be negated.

### Repetition

In EBNF, curly braces indicate that the expression may be repeated zero or more times.

For example, the rule:

```
<args> ::= <arg> { "," <arg> }
```

defines a conventional comma-separated argument list.

### Grouping

To indicate precedence, EBNF grammars may use parentheses, `()`, to explictly define the order of expansion.

For example, the rule:

```
<expr> ::= <term> ("+" | "-") <expr>
```

defines an expression form that allows both addition and subtraction.

### Concatenation

In some forms of EBNF, the `,` operator explicitly denotes concatenation, rather than relying on juxtaposition.

## Augmented BNF (ABNF) notation

Protocol specifications often use Augmented Backus-Naur Form (ABNF).

For example, RFC 5322 (email), uses ABNF.

RFC 5234 defines ABNF.

ABNF is similar to EBNF in principle, except that its notations for choice, option and repetition differs.

ABNF also provides the ability to specify specific byte values exactly -- detail which matters in protocols.

In ABNF:

- choice is `/`; and
- option uses square brackets: `[ ]`; and
- repetition is *prefix* `*`; and
- repetition `n` or more times is *prefix* `n*`; and
- repetition `n` to `m` times is *prefix* `n*m`.

EBNF's `{ expansion }` becomes `*(expansion)` in ABNF.

Here's a definition of a date and time format taken from RFC 5322.

```
date-time       =   [ day-of-week "," ] date time [CFWS]

day-of-week     =   ([FWS] day-name) / obs-day-of-week

day-name        =   "Mon" / "Tue" / "Wed" / "Thu" /
                    "Fri" / "Sat" / "Sun"

date            =   day month year

day             =   ([FWS] 1*2DIGIT FWS) / obs-day

month           =   "Jan" / "Feb" / "Mar" / "Apr" /
                    "May" / "Jun" / "Jul" / "Aug" /
                    "Sep" / "Oct" / "Nov" / "Dec"

year            =   (FWS 4*DIGIT FWS) / obs-year

time            =   time-of-day zone

time-of-day     =   hour ":" minute [ ":" second ]

hour            =   2DIGIT / obs-hour

minute          =   2DIGIT / obs-minute

second          =   2DIGIT / obs-second

zone            =   (FWS ( "+" / "-" ) 4DIGIT) / obs-zone
```

# Regular extensions to BNF

It's common to find regular-expression-like operations inside grammars.

For instance, the Python lexical specification uses them.

In these grammars:

- postfix `*` means "repeated 0 or more times"
- postfix `+` means "repeated 1 or more times"
- postfix `?` means "0 or 1 times"

The definition of floating point literals in Python is a good example of combining several notations:

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
```

```
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

It does not use angle brackets around names (like many EBNF notations and ABNF), yet does use `::=` (like BNF). It mixes regular operations like `+` for non-empty repetition with EBNF conventions like `[ ]` for option.

The grammar for the entire Python language uses a slightly different (but still regular) notation.

## Grammars in mathematics

Even when grammars are not an object of mathematical study themselves, in texts that deal with discrete mathematical structures, grammars appear to define new notations and new structures.

For more on this, see my article on translating math into code.

## Beyond context-free grammars

Regular expressions sit just beneath context-free grammars in descriptive power: you could rewrite any regular expression into a grammar that represents the srings matched by the expression. But, the reverse is not true: not every grammar can be converted into an equivalent regular expression.

To go beyond the expressive power of context-free grammars, one needs to allow a degree of context-sensitivity in the grammar.

Context-sensitivity means that terminal symbols may also appear in the left-hand sides of rules.

Consider the following contrived grammar:

```
<top> ::= <a> ")"
<a>   ::= "(" <exp>

"(" <exp> ")" ::= 7
```

`<top>` may expand into `<a> ")"`;
which may expand into `"(" <exp> ")"`;
which may expand into `7`.

While this change appears small, it makes grammars equivalent to Turing machines in terms of the languages they can describe.

By restricting the rules so that the the left-hand side has strictly fewer symbols than all expansions on the right, context-sensitive grammars are equivalent to (decidable) linear-bounded automata.

Even though some languages are context-sensitive, context-sensitive grammars are rarely used for describing computer languages.

For instance, C is slightly context-sensitive because of the way it handles identifiers and type, but this context-sensitivity is resolved by a special

convention, rather than by introducing context-sensitivity into the grammar.

## Parsing

This article covered the process of interpreting grammars and common notations.

A closely related topic is parsing.

Parsing takes a grammar and a string and answers two questions:

1. Is that string in the language of the grammar?
2. What is the *structure* of that string relative to the grammar?

For an comprehensive treatment of parsing techniques, I recommend Grune and Jacobs, Parsing Techniques: A Practical Guide:

As an aside, if you think you've invented a new parsing technique, you need to check this book first. Your peer reviewers will check it.

My own articles on parsing may also serve as a useful reference:

- Desugaring regular operations in context-free grammars
- Parsing regular expressions with recursive descent
- Standalone lexers with lex: synopsis, examples, and pitfalls
- Parsing with derivatives (Yacc is dead: An update)
- A non-blocking lexing toolkit for Scala from regex derivatives
- Lexical analysis and syntax-highlighting in JavaScript
- Matching regular expressions with derivatives
- Implementing regular expressions and NFAs in Java
- Parsing M-Expressions in Scala with combinators

---

[article index] [email me] [@mattmight] [rss]