Introduction to EBNF

What a Grammar Is

A grammar is a formal description of the structure of a language.

- Defines what strings are valid in the language
- Does not execute code or compute values
- Answers: "What does a well-formed program look like?"

Examples:

- Arithmetic expressions
- A subset of JavaScript
- Configuration file format
- Command language

The Idea Behind BNF and EBNF

BNF (Backus-Naur Form) was created to describe programming languages.

EBNF is a small extension that adds conveniences:

- Repetition
- Optional parts
- Grouping

EBNF is not one standard; many dialects exist.

Basic Notation

An EBNF rule has the form:

```
name ::= definition
```

- `name` is a nonterminal (a grammatical category)
- `definition` is built from nonterminals and terminals

Example:

```
number ::= digit { digit }
digit  ::= "0" | "1" | "2" | ... | "9"
```

A `number` is a `digit` followed by zero or more `digit`s.

Core Operators

- `|` — choice (alternation)
- `{ }` — repetition (zero or more)
- `[ ]` — optional (zero or one)
- `( )` — grouping

Example:

```
sign    ::= "+" | "-"
integer ::= [ sign ] digit { digit }
```

Valid: `7`, `-3`, `+42`

Invalid: `--3`, `+`

Describing Expressions

A simple arithmetic grammar:

```
expression ::= term { ("+" | "-") term }
term       ::= factor { ("*" | "/") factor }
factor     ::= number | "(" expression ")"
number     ::= digit { digit }
digit      ::= "0" | "1" | ... | "9"
```

This encodes precedence: `*` and `/` bind tighter than `+` and `-`.

Valid: `3+4`, `2*(5+1)`, `10/2-3`

Invalid: `+3`, `3+`, `2*(4+`

Tokens vs Grammar

In practice, grammars operate on tokens, not characters.

Instead of:

```
number ::= digit { digit }
```

We assume:

```
number     ::= <NUMBER>
identifier ::= <IDENTIFIER>
```

`<NUMBER>` and `<IDENTIFIER>` are produced by a tokenizer.

This keeps grammar focused on structure, not spelling.

Statements and Blocks

A tiny statement language:

```
program   ::= { statement }
statement ::= assignment ";"
            | "if" "(" expression ")" block
            | "while" "(" expression ")" block
assignment ::= identifier "=" expression
block ::= "{" { statement } "}"
```

Valid program:

```
x = 3;
while (x) {
    x = x - 1;
}
```

What Grammars Do Not Do

A grammar does not:

- Check types
- Evaluate expressions
- Enforce variable declaration rules
- Decide whether a variable exists

Grammar allows:

```
x = y + 3;
```

Even if `y` is undefined.

Grammar answers only:

- Is this structurally valid?

Ambiguity

A grammar is ambiguous if a string can be parsed in more than one way.

Classic example:

```
expression ::= expression "+" expression | number
```

The string `1 + 2 + 3` can be grouped as:

- `(1 + 2) + 3`
- `1 + (2 + 3)`

Solution: Introduce structure (term, factor, etc.).

Why EBNF Matters

EBNF is the bridge between:

- Informal language descriptions
- Working parsers

It lets us:

- Be precise
- Communicate structure
- Reason about edge cases
- Generate or implement parsers