Context-Free Grammar (CFG)

A *context-free grammar* looks like this bunch of rules:

$$
\begin{array}{ll}
E \to E + E & E \to M \\
M \to M \times M & M \to A \\
A \to 0 & A \to 1 \\
A \to (E) &
\end{array}
$$

Main idea:

- E, M, A are *non-terminal symbols* aka *variables*. When you see them, you apply rules to expand.
  One of them is designated as the *start symbol*. You always start from it. I will designate E as the start symbol.
- $+$, $\times$, 0, 1, $($, $)$ are *terminal symbols*. They are the characters you want in your language.

Derivation (aka Generation)

Derivation is a finite sequence of applying the rules until all non-terminal symbols are gone. Often aim for a specific final string.

$$\begin{aligned}
E &\to M & &\to 1 \times (M + E) \\
&\to M \times M & &\to 1 \times (A + E) \\
&\to A \times M & &\to 1 \times (0 + E) \\
&\to 1 \times M & &\to 1 \times (0 + M) \\
&\to 1 \times A & &\to 1 \times (0 + M \times M) \\
&\to 1 \times (E) & &\to 1 \times (0 + A \times M) \\
&\to 1 \times (E + E) & &\to 1 \times (0 + 1 \times A) \\
& & &\to 1 \times (0 + 1 \times 1)
\end{aligned}$$

Context-free grammars can support: matching parentheses, unlimited nesting.

Backus-Naur Form (BNF)

Backus-Naur Form is a computerized, practical notation for CFGs.

- Surround non-terminal symbols by <>; allow multi-letter names.
- Merge rules with the same LHS.
- (Some versions.) Surround terminal strings by single or double quotes.
- Use `::=` for →.

Our example grammar in BNF:

```
<expr> ::= <expr> "+" <expr> | <mul>
<mul> ::= <mul> "*" <mul> | <atom>
<atom> ::= "0" | "1" | "(" <expr> ")"
```

Extended Backus-Naur Form (EBNF)

- `{...}` for 0 or more occurrences.
- `[...]` for 0 or 1 occurrences.
- Some versions: No <> needed around non-terminal symbols.

Example: Lisp/Scheme S-expression[1] grammar (basic):

In BNF:

```
<s-expr> ::= <identifier> | "(" <s-exprs> ")"
<s-exprs> ::= <s-expr> | <s-expr> <s-exprs>
```

In EBNF:

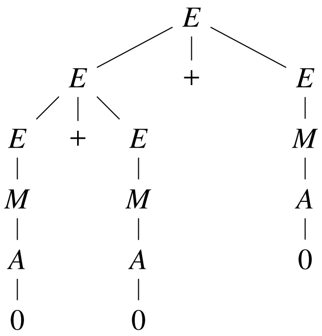
```
<s-expr> ::= <identifier>
           | "(" <s-expr> { <s-expr> } ")"
```

So you need fewer artificial non-terminals and rules that merely mean “at least 0 of this”, “at least 1 of that”, etc.

[1]“symbolic expression”

Parse Tree aka Derivation Tree

A *parse tree* aka *derivation tree* presents a derivation with more structure (tree), less repetition.



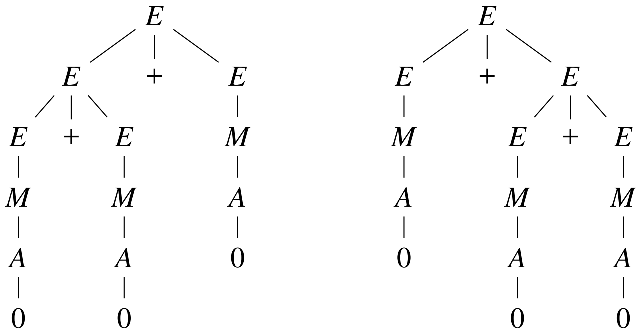
This example generates $0 + 0 + 0$.

Parse Trees: General Points

- Internal nodes are non-terminal symbols.
- Both operators and operands are terminal symbols at leaves.
- Whole string recorded, just scattered.
- Purpose: Present derivation, help visualize derivation and grammar.

Ambiguous Grammar

Two different trees generate the same $0 + 0 + 0$:



If this happens, the grammar is *ambiguous*. Not very desirable: parsing is more costly (return all results) or more surprising (return one result but not what you expect).

(Bad news: CFG ambiguity is undecidable.)

Unambiguous Grammar Example

An unambiguous grammar that generates the same language as our ambiguous grammar example:

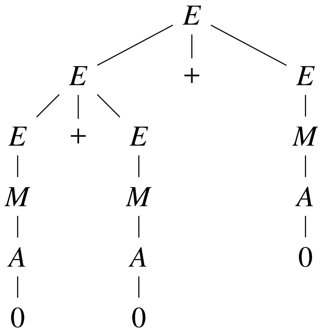
```
<expr> ::= <expr> "+" <mul> | <mul>
<mul> ::= <mul> "*" <atom> | <atom>
<atom> ::= "0" | "1" | "(" <expr> ")"
```

Exercise: Find the parse trees for $0 + 0 + 0$ and $0 \times 0 \times 0$. Observe that you are forced only one answer, and it's left-leaning.

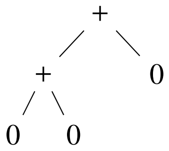
(Bad news: Equivalence of two CFGs is also undecidable.)

Abstract Syntax Tree (AST) (vs Parse Tree)

Parse tree:



Abstract syntax tree:



Abstract Syntax Tree: General Points

- Internal nodes are operators/constructs.
  Example construct: if-then-else.
- Non-terminal symbols gone or replaced by constructs.
- Many terminal symbols gone too if they play no role other than nice syntax (e.g., spaces, parentheses, punctuations).
  Those bearing content, replaced by appropriate representations, not stay as characters.
  E.g., Character `'+'` replaced by data constructor or object, character `'0'` replaced by number 0.
- Purpose: Present essential structure and content, ready for interpreting, compiling, analyses.
- Parsers usually output abstract syntax trees when successful.

Left Recursive vs Right Recursive

```
<expr> ::= <expr> "+" <mul>
```

That is a *left recursive* rule. The recursion is at the beginning (left).

```
<expr> ::= <mul> "+" <expr>
```

That is a *right recursive* rule. The recursion is at the end (right).

Sometimes they convey intentions of left association or right association. But not always.

They affect some parsing algorithms.

Recursive Descent Parsing (by example)

A simple strategy for writing a parser, not very efficient but easy to understand and code up.

A rule like `S ::= "x" R` becomes

```
parserForS:
  read a char, if not 'x', fail
  parserForR
```

A rule like `R ::= "y" | S` becomes

```
parserForR:
  save input position  # for backtracking below
  try:
    read a char, if not 'y', fail
  catch:
    # here be backtracking!
    restore input position
    parserForS
```

Recursive Descent Parsing (general points)

- Use mutually recursive procedures for mutually recursive rules. (In practice can inline, refactor, add helpers.)

  No good for left recursion.

- Use backtracking to handle choice.

  Major cause of the reputation of high inefficiency.

- Execution is like walking down the parse tree. Hence "descent", "top-down".

Some options for handling left recursion:

- Re-design grammar to not have left recursion.
- Many left recursive rules just express left-associating operators. Can be done without left recursive code.

Lexical Analysis aka Tokenization

In principle: Grammar and parser can work on characters directly. But usually messy.

In practice, two stages:

1. *Lexical Analysis*, *tokenization*: Chop character stream into chunks, classify into *lexemes* aka *tokens*, discard spaces.
   ```
   "(xa * xb)**25"
   ↦
   [OpenParen, Var "xa", Op Mul, Var "xb",
   CloseParen, Op Exp, NumLiteral 25]
   ```
   Needs only regular expressions.
2. CFG parsing, but terminal symbols are tokens, not strings.