Chapter Twelve: Context-Free Languages

We defined the right-linear grammars by giving a simple restriction on the form of each production. By relaxing that restriction a bit, we get a broader class of grammars: the context-free grammars. These grammars generate the context-free languages, which include all the regular languages along with many that are not regular.

Outline

Examples

- We've proved that these languages are not regular, yet they have grammars
  - $\{a^n b^n\}$
  - $\{xx^R \mid x \in \{a,b\}^*\}$
  - $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$

$S \rightarrow aSb \mid \varepsilon$

$S \rightarrow aSa \mid bSb \mid \varepsilon$

$S \rightarrow aSa \mid R$
$R \rightarrow bR \mid b$

- Although not right-linear, these grammars still follow a rather restricted form…

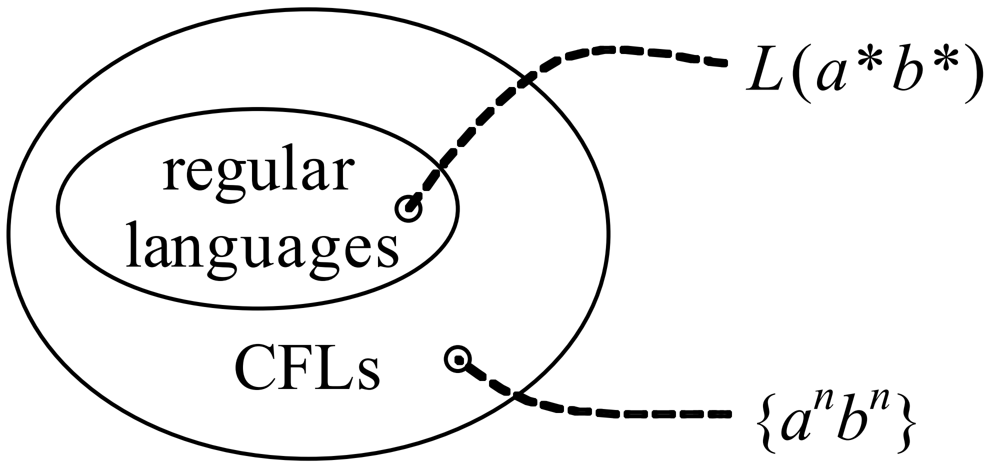
Context-Free Grammars

- A context-free grammar (CFG) is one in which every production has a single nonterminal symbol on the left-hand side
- A production like $R \to y$ is permitted
  - It says that R can be replaced with y, regardless of the context of symbols around R in the string
- One like $uRz \to uyz$ is not permitted
  - That would be context-sensitive: it says that R can be replaced with y only in a specific context

Context-Free Languages

- A context-free language (CFL) is one that is $L(G)$ for some CFG G
- Every regular language is a CFL
  - Every regular language has a right-linear grammar
  - Every right-linear grammar is a CFG
- But not every CFL is regular
  - $\{a^n b^n\}$
  - $\{xx^R \mid x \in \{a,b\}^*\}$
  - $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$

Language Classes So Far



Outline

Writing CFGs

- Programming:
  - A program is a finite, structured, mechanical thing that specifies a potentially infinite collection of runtime behaviors
  - You have to imagine how the code you are crafting will unfold when it executes
- Writing grammars:
  - A grammar is a finite, structured, mechanical thing that specifies a potentially infinite language
  - You have to imagine how the productions you are crafting will unfold in the derivations of terminal strings
- Programming and grammar-writing use some of the same mental muscles
- Here follow some techniques and examples…

Regular Languages

- If the language is regular, we already have a technique for constructing a CFG
  - Start with an NFA
  - Convert to a right-linear grammar using the construction from chapter 10

Example

$L = \{x \in \{0,1\}^* \mid$ the number of 0s in x is divisible by 3$\}$

$$\begin{tikzcd}
{} \arrow[r] & S \arrow[r, "0"] \arrow[loop above, "1"] & T \arrow[r, "0"] \arrow[loop above, "1"] & U \arrow[ll, bend left, "0"] \arrow[loop above, "1"]
\end{tikzcd}$$

$$\boxed{\begin{array}{l} S \rightarrow 1S \mid 0T \mid \varepsilon \\ T \rightarrow 1T \mid 0U \\ U \rightarrow 1U \mid 0S \end{array}}$$

Example

$L = \{x \in \{0,1\}^* \mid \text{the number of 0s in } x \text{ is divisible by 3}\}$

- The conversion from NFA to grammar always works
- But it does not always produce a pretty grammar
- It may be possible to design a smaller or otherwise more readable CFG manually:

$$\boxed{\begin{aligned} S &\to 1S \mid 0T \mid \varepsilon \\ T &\to 1T \mid 0U \\ U &\to 1U \mid 0S \end{aligned}}$$

$$\boxed{\begin{aligned} S &\to T0T0T0S \mid T \\ T &\to 1T \mid \varepsilon \end{aligned}}$$

Balanced Pairs

- CFLs often seem to involve balanced pairs
  - $\{a^n b^n\}$: every *a* paired with *b* on the other side
  - $\{xx^R \mid x \in \{a,b\}^*\}$: each symbol in *x* paired with its mirror image in x^R
  - $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$: each *a* on the left paired with one on the right
- To get matching pairs, use a recursive production of the form $R \to xRy$
- This generates any number of *x*s, each of which is matched with a *y* on the other side

Examples

- We've seen these before:
  - $\{a^n b^n\}$ $\boxed{S \rightarrow aSb \mid \varepsilon}$
  - $\{xx^R \mid x \in \{a,b\}^*\}$ $\boxed{S \rightarrow aSa \mid bSb \mid \varepsilon}$
  - $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$ $\boxed{\begin{array}{l} S \rightarrow aSa \mid R \\ R \rightarrow bR \mid b \end{array}}$
- Notice that they all use the $R \rightarrow xRy$ trick

Examples

- $\{a^n b^{3n}\}$
  - Each a on the left can be paired with three bs on the right
  - That gives

$$\boxed{S \rightarrow aSbbb \mid \varepsilon}$$

- $\{xy \mid x \in \{a,b\}^*, y \in \{c,d\}^*, \text{ and } |x| = |y|\}$
  - Each symbol on the left (either a or b) can be paired with one on the right (either c or d)
  - That gives

$$\boxed{\begin{array}{l} S \rightarrow XSY \mid \varepsilon \\ X \rightarrow a \mid b \\ Y \rightarrow c \mid d \end{array}}$$

Concatenations

- A divide-and-conquer approach is often helpful
- For example, $L = \{a^n b^n c^m d^m\}$
  - We can make grammars for $\{a^n b^n\}$ and $\{c^m d^m\}$:

$$\boxed{S_1 \to aS_1b \mid \varepsilon} \qquad \boxed{S_2 \to cS_2d \mid \varepsilon}$$

  - Now every string in L consists of a string from the first followed by a string from the second
  - So combine the two grammars and add a new start symbol:

$$\boxed{\begin{array}{l} S \to S_1S_2 \\ S_1 \to aS_1b \mid \\ \varepsilon S_2 \to cS_2d \mid \varepsilon \end{array}}$$

Concatenations, In General

- Sometimes a CFL L can be thought of as the concatenation of two languages L_1 and L_2
  - That is, $L = L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- Then you can write a CFG for L by combining separate CFGs for L_1 and L_2
  - Be careful to keep the two sets of nonterminals separate, so no nonterminal is used in both
  - In particular, use two separate start symbols S_1 and S_2
- The grammar for L consists of all the productions from the two sub-grammars, plus a new start symbol S with the production $S \to S_1S_2$

Unions, In General

- Sometimes a CFL L can be thought of as the union of two languages $L = L_1 \cup L_2$
- Then you can write a CFG for L by combining separate CFGs for L_1 and L_2
  - Be careful to keep the two sets of nonterminals separate, so no nonterminal is used in both
  - In particular, use two separate start symbols S_1 and S_2
- The grammar for L consists of all the productions from the two sub-grammars, plus a new start symbol S with the production $S \rightarrow S_1 \mid S_2$

Example

$L = \{z \in \{a,b\}^* \mid z = xx^R$ for some x, or $|z|$ is odd$\}$

- This can be thought of as a union: $L = L_1 \cup L_2$
  - $L_1 = \{xx^R \mid x \in \{a,b\}^*\}$

    $\boxed{S_1 \to aS_1a \mid bS_1b \mid \varepsilon}$

  - $L_2 = \{z \in \{a,b\}^* \mid \ |z| \text{ is odd}\}$

    $$\boxed{\begin{array}{l} S_2 \to XXS_2 \mid X \\ X \to a \mid b \end{array}}$$

- So a grammar for L is

$$\boxed{\begin{array}{l} S \to S_1 \mid S_2 \\ S_1 \to aS_1a \mid bS_1b \mid \varepsilon \\ S_2 \to XXS_2 \mid X \\ X \to a \mid b \end{array}}$$

Example

$$L = \{a^n b^m \mid n \neq m\}$$

- This can be thought of as a union:
  - $L = \{a^n b^m \mid n < m\} \cup \{a^n b^m \mid n > m\}$
- Each of those two parts can be thought of as a concatenation:
  - $L_1 = \{a^n b^n\}$
  - $L_2 = \{b^i \mid i > 0\}$
  - $L_3 = \{a^i \mid i > 0\}$
  - $L = L_1 L_2 \cup L_3 L_1$
- The resulting grammar:

$$\boxed{\begin{array}{l} S \to S_1 S_2 \mid S_3 S_1 \\ S_1 \to a S_1 b \mid \varepsilon \\ S_2 \to b S_2 \mid b \\ S_3 \to a S_3 \mid a \end{array}}$$

Outline

BNF

- John Backus and Peter Naur
- A way to use grammars to define the syntax of programming languages (Algol), 1959-1963
- BNF: Backus-Naur Form
- A BNF grammar is a CFG, with notational changes:
  - Nonterminals are written as words enclosed in angle brackets: <*exp*> instead of *E*
  - Productions use ::= instead of $\rightarrow$
  - The empty string is <*empty*> instead of ε
- CFGs (due to Chomsky) came a few years earlier, but BNF was developed independently

Example

$$\begin{array}{l} \langle exp \rangle ::= \langle exp \rangle \texttt{ - } \langle exp \rangle \mid \langle exp \rangle \texttt{ * } \langle exp \rangle \mid \langle exp \rangle = \langle exp \rangle \\ \qquad\qquad \mid \langle exp \rangle < \langle exp \rangle \mid \texttt{(}\langle exp \rangle\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \end{array}$$

- This BNF generates a little language of expressions:
  - `a<b`
  - `(a-(b*c))`

Example

<stmt> ::= *<exp-stmt>* | *<while-stmt>* | *<compound-stmt>* |...
<exp-stmt> ::= *<exp>* `;`
<while-stmt> ::= `while ( `*<exp>*` ) `*<stmt>*
<compound-stmt> ::= `{` *<stmt-list>* `}`
<stmt-list> ::= *<stmt>* *<stmt-list>* | *<empty>*

- This BNF generates C-like statements, like

  -
    ```
    while (a<b) {
      c = c * a;
      a = a + a;
    }
    ```
- This is just a toy example; the BNF grammar for a full language may include hundreds of productions

Outline

Formal vs. Programming Languages

- A formal language is just a set of strings:
  - DFAs, NFAs, grammars, and regular expressions define these sets in a purely syntactic way
  - They do not ascribe meaning to the strings
- Programming languages are more than that:
  - *Syntax*, as with formal languages
  - Plus *semantics*: what the program means, what it is supposed to do
- The BNF grammar specifies not only syntax, but a bit of semantics as well

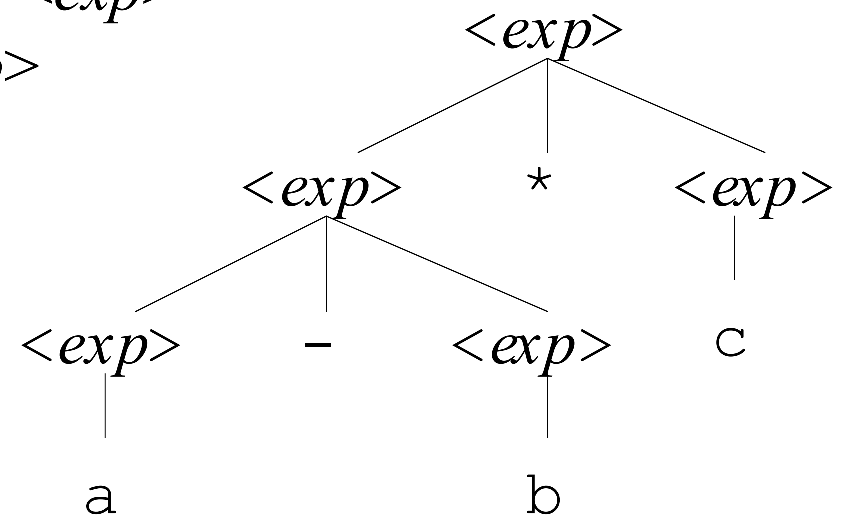
Parse Trees

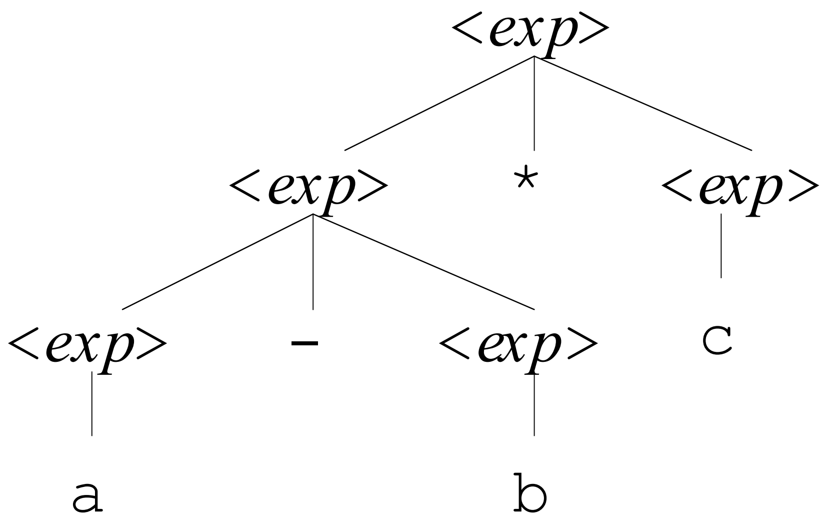
- We've treated productions as rules for building strings
- Now think of them as rules for building *trees:*
  - Start with S at the root
  - Add children to the nodes, always following the rules of the grammar: $R \rightarrow x$ says that the symbols in x may be added as children of the nonterminal symbol R
  - Stop only when all the leaves are terminal symbols
- The result is a *parse tree*

Example

<exp> ::= *<exp>* `-` *<exp>* | *<exp>* `*` *<exp>* | *<exp>* = *<exp>*
| *<exp>* < *<exp>* | `(`*<exp>*`)` | `a` | `b` | `c`

<exp> ⇒ *<exp>* `*` *<exp>*
⇒ *<exp>* `-` *<exp>* `*` *<exp>*
⇒ `a-` *<exp>* `*` *<exp>*
⇒ `a-b*` *<exp>*
⇒ `a-b*c`





- The parse tree specifies:
  - Syntax: it demonstrates that `a-b*c` is in the language
  - Also, the beginnings of semantics: it is a plan for evaluating the expression when the program is run
  - First evaluate `a-b`, then multiply that result by `c`
- It specifies how the parts of the program fit together
- And that says something about what happens when the program runs

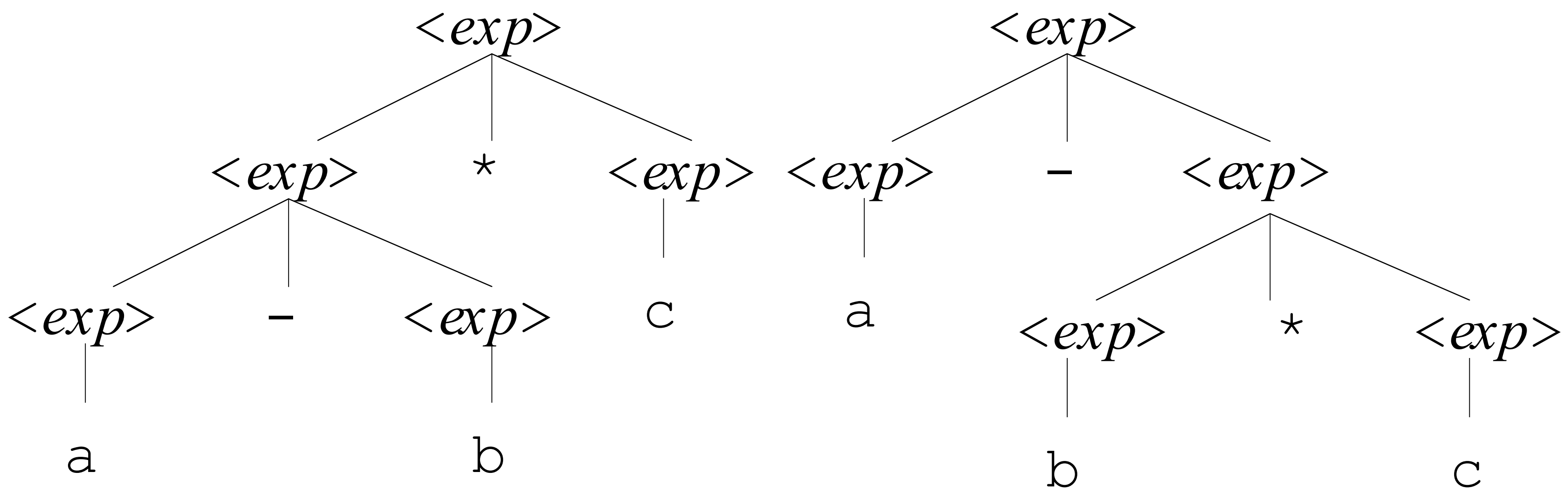
Parsing

- To parse a program is to find a parse tree for it, with respect to a grammar for the language
- Every time you compile a program, the compiler must first parse it
- The parse tree (or a simplified version called the *abstract syntax tree*) is one of the central data structures of almost every compiler
- More about algorithms for parsing in chapter 15

Outline

```
<exp> ::= <exp> - <exp> | <exp> * <exp> | <exp> = <exp>
              | <exp> < <exp> | (<exp>) | a | b | c
```

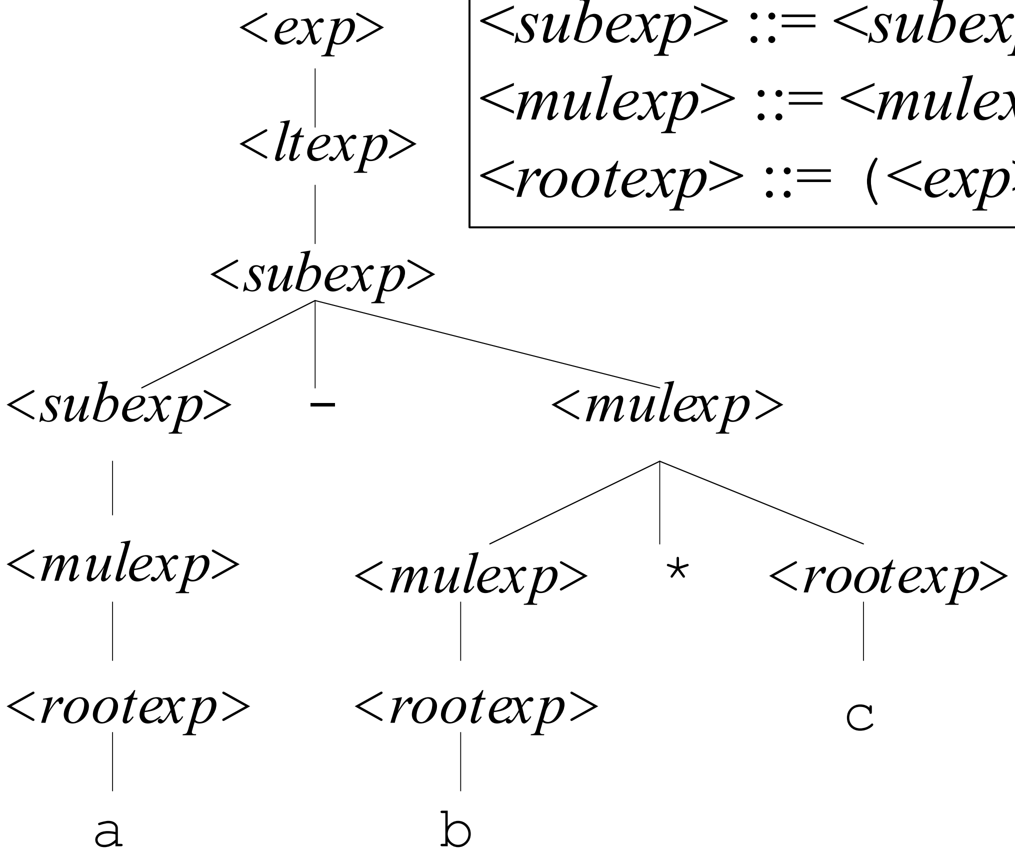
- A grammar is *ambiguous* if there is a string in the language with more than one parse tree
- The grammar above is ambiguous:



Ambiguity

- That kind of ambiguity is unacceptable
- Part of the definition of the language must be a clear decision on whether `a-b*c` means $(a\text{-}b)\times c$ or $a\text{-}(b\times c)$
- To resolve this problem, BNF grammars are usually crafted to be unambiguous
- They not only specify the syntax, but do so with a unique parse tree for each program, one that agrees with the intended semantics
- Not usually difficult, but it generally means making the grammar more complicated

<exp> ::= *<ltexp>* = *<exp>* | *<ltexp>*
<ltexp> ::= *<ltexp>* < *<subexp>* | *<subexp>*
<subexp> ::= *<subexp>* `-` *<mulexp>* | *<mulexp>*
<mulexp> ::= *<mulexp>* `*` *<rootexp>* | *<rootexp>*
<rootexp> ::= `(`*<exp>*`)` | `a` | `b` | `c`



Trade-Off

- The new grammar is unambiguous
  - Strict precedence: `*`, then `-`, then `<`, then `=`
  - Strict associativity: left, so `a-b-c` is computed as (*a*-*b*)-*c*
- On the other hand, it is longer and less readable
- Many BNFs are meant to be used both by people and directly by computer programs
  - The code for the parser part of a compiler can be generated automatically from the grammar by a parser-generator
  - Such programs really want unambiguous grammars

Inherent Ambiguity

- There are CFLs for which it is not possible to give an unambiguous grammar
- They are *inherently ambiguous*
- This is not usually a problem for programming languages

Outline

Extending BNF

- More metasymbols to help with common patterns of language definition:
  - [*something*] means that the *something* inside is optional
  - { *something* } means that the *something* inside can be repeated any number of times (zero or more), like the Kleene star in regular expressions
  - Plain parentheses are used to group things, so that |, [], and {} can be combined unambiguously

Examples

- An if-then statement with optional else

  <if-stmt> ::= `if` *<expr>* `then` *<stmt>* [`else` *<stmt>*]

- A list of zero or more statements, each ending with a semicolon

  <stmt-list> ::= {*<stmt>* `;`}

- A list of zero or more things, each of which can be either a statement or a declaration and each ending with a semicolon:

  <thing-list> ::= { (*<stmt>* | *<declaration>*) `;`}

EBNF

- Plain BNF can handle all those examples, but they're easier with our extensions
- Any grammar syntax that extends BNF in this way is called an *extended BNF* (EBNF)
- Many variations have been used
- There is no widely accepted standard

EBNF and Parse Trees

- The use of {} metasymbols obscures the form of the parse tree
  - BNF: *<mulexp>* ::= *<mulexp>* * *<rootexp>* | *<rootexp>*
  - EBNF: *<mulexp>* ::= *<rootexp>* {* *<rootexp>*}
- The BNF allows only a left-associative parse tree for something like `a*b*c`
- The EBNF is unclear
- With some EBNFs the form above implies left associativity, but there is no widely accepted standard for such conventions