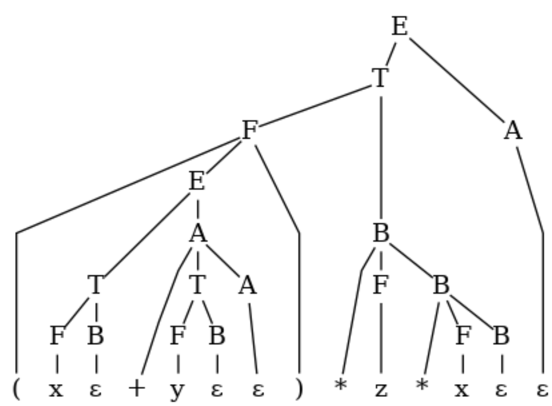
Draw a parse tree for the string $(x + y) * z * x$ according to the grammar below.

$$
\begin{aligned}
E &\longrightarrow TA \\
A &\longrightarrow +TA \\
A &\longrightarrow \epsilon \\
T &\longrightarrow FB \\
B &\longrightarrow *FB \\
B &\longrightarrow \epsilon \\
F &\longrightarrow (E) \\
F &\longrightarrow x \\
F &\longrightarrow y \\
F &\longrightarrow z
\end{aligned}
$$

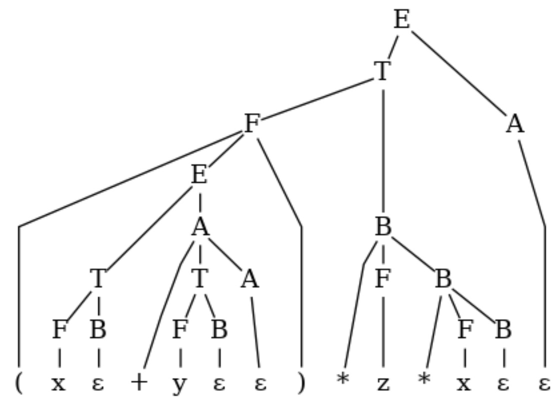
Answer:



Discussion:

The structure of a parse tree reflects the rules in the grammar. Each symbol on the right side of a rule has its own branch. To illustrate this structure, consider a derivation for $(x + y) * z * x$:

$$
\begin{aligned}
E &\Longrightarrow TA \\
&\Longrightarrow FBA \\
&\Longrightarrow (E)BA \\
&\Longrightarrow (TA)BA \\
&\Longrightarrow (FBA)BA \\
&\Longrightarrow (xBA)BA \\
&\Longrightarrow (xA)BA \\
&\Longrightarrow (x+TA)BA \\
&\Longrightarrow (x+FBA)BA \\
&\Longrightarrow (x+yBA)BA \\
&\Longrightarrow (x+yA)BA \\
&\Longrightarrow (x+y)BA \\
&\Longrightarrow (x+y)*FBA \\
&\Longrightarrow (x+y)*zBA \\
&\Longrightarrow (x+y)*z*FBA \\
&\Longrightarrow (x+y)*z*xBA \\
&\Longrightarrow (x+y)*z*xA \\
&\Longrightarrow (x+y)*z*x
\end{aligned}
$$



The parse tree is shown on the right. The root of the parse tree (at the top) is the start symbol in the derivation — E in this case. The first step of the derivation uses the rule $E \longrightarrow TA$, and the two branches below the E in the parse tree are for T and A. Next, the rule $T \longrightarrow FB$ is used, and the two branches below the T in the parse tree are for F and B. Note that (and) are symbols in this grammar, not part of the production rule syntax.

Find a context-free grammar for the language $\{\ a^n b^n c^k \mid n, k \in \mathbb{N}\ \}$.

Answer:

$$
\begin{aligned}
S &\longrightarrow TC \\
T &\longrightarrow aTb \\
T &\longrightarrow \epsilon \\
C &\longrightarrow cC \\
C &\longrightarrow \epsilon
\end{aligned}
$$

Discussion:

When constructing context-free grammar rules, pay attention to both the order of symbols and, when there's a relationship between the numbers of two different elements, generate matched sets of symbols.

For this problem, the as, bs, and cs come in an order — as first, then bs, then cs. Also, there must be the same number of as as bs but there can be any number of cs.

For the as and bs, the way to generate a matched number is with a rule

$$T \longrightarrow aTb$$

This preserves the order (as before bs) and keeps the same number of as as bs. Note that a rule like

$$T \longrightarrow abT$$

also preserves the same number of as as bs, but it doesn't maintain the order:

$$T \Longrightarrow abT \Longrightarrow ababT$$

To generate any number of a symbol, use a rule like one of the following:

$$C \longrightarrow cC$$

$$C \longrightarrow Cc$$

Which is better, if it matters, depends on what happens to the C on the right side — if it just goes away ($C \longrightarrow \epsilon$), then either rule is fine. But it C is eventually replaced by something else, whether that something else should come before the cs generated or after dictates which rule is correct.

Finally, put these rules together with a rule that establishes the start symbol and sets the ordering of the elements:

$$S \longrightarrow TC$$

Check that this works with a few steps of a derivation:

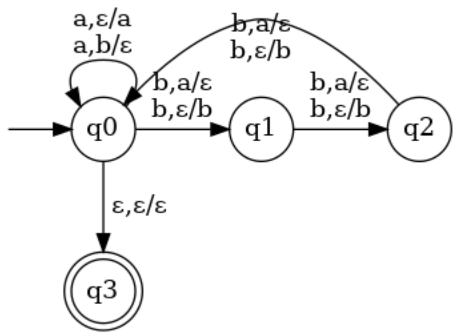
$$S \Longrightarrow TC \Longrightarrow aTbC \Longrightarrow aaTbbC \Longrightarrow aaTbbcC \Longrightarrow aaTbbccC$$

What remains is a few rules to clean up the Ts and Cs. Since $n, k \in \mathbb{N}$, there doesn't need to be at least one of any symbol so $T \longrightarrow \epsilon$ and $C \longrightarrow \epsilon$ can do that cleanup.

Create a pushdown automaton that accepts the language

$$L = \{\ w \in \{a,b\}^* \mid n_a(w) = n_b(w) \text{ and consecutive } b\text{'s only occur in multiples of 3 } \}$$

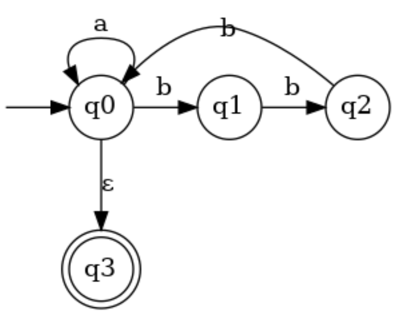
Answer:



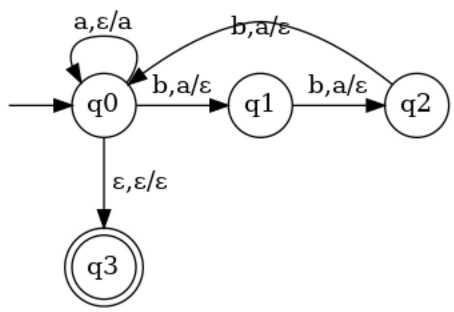
Discussion: Keep in mind the two key elements of a pushdown automaton and the role each plays:

- The state transitions consume the input string, and states are used for tracking specific numbers and sequences of symbols.
- The stack is used for matching — one symbol with another, or a count of symbols with another count.

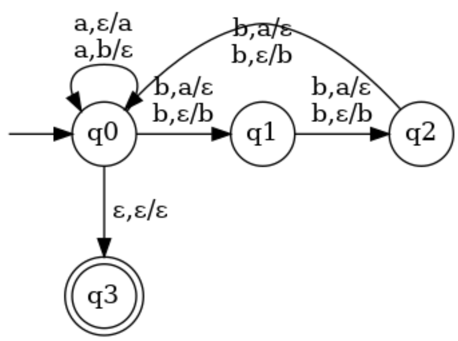
Start with the states. "Consecutive b's only occur in multiples of 3" is a specific numbers and sequences of symbols thing, so first build an NFA for that:



Then address the stack. $n_a(w) = n_b(w)$ is a matching thing, so that is the stack's job. Push when there's an a and pop when there's a b to match numbers:



However, there's a problem — for a string like $abbbaa$, some of the as are after bs so there isn't yet enough on the stack to pop three times for all the bs. The stack needs to be used not just to count as, but to count whatever there is an excess of.



Now reading a a means popping a b if there is one on the stack (an excess of bs) or pushing an a (an excess of as), and reading a b means popping an a if there is one on the stack (an excess of as) or pushing a b (an excess of bs). This is not a deterministic automaton, but that's OK. (The push option is always an option, but remember that the stack has to be empty in order to accept — if things pushed aren't popped when possible, the stack won't be empty in the end.)