Context-Free Grammars

Recall: Regular languages had regexes as a representation

We saw that for some non-regular languages, we could write grammars to represent them.

A grammar is a 4-tuple of the form (NT, T, R, S)

Set of non-terminals

Set of terminals

set of production rules

$\in NT$ Start symbol

Each production rule in a context-free grammar has a non-terminal on the left, followed by ::= or $\longrightarrow$, and some sequence of terminals & non-terminals on the right

$S ::= \varepsilon$

$S \to \varepsilon$

The main use of CFGs is in parsing.

For example, I might want to recognize the language of all strings with balanced parentheses

But the same CFG from above does not work for this!

$$\mathcal{L}_{()} = \left\{ \omega \;\middle|\; \begin{array}{l} \omega \text{ contains an equal number of `(' and `)', and} \\ \text{every prefix } s \text{ of } \omega \text{ contains at least} \\ \text{as many `('s as `)'s} \end{array} \right\}$$

This is a very verbose description.

What is a CFG for $\mathcal{L}$?

$S ::= \varepsilon \mid (S) \mid SS$

$G = \left(\{S\}, \{(,)\}, \{S ::= \varepsilon, S ::= (S), S ::= SS\}, S \right)$

How do we prove that $\mathcal{L}(G) = \mathcal{L}_{()}$?

① Show that every string $w \in \mathcal{L}(G)$ is s.t. $w \in \mathcal{L}_{()}$.

② Show that every string $w \in \mathcal{L}_{()}$ is s.t. $w \in \mathcal{L}(G)$.

Possible cases for w:

$w = \varepsilon$: $S ::= \varepsilon$

$w = ?$

Any language $\mathcal{L}$ s.t. $\mathcal{L} = \mathcal{L}(G)$ for some CFG G is called a context-free language.

$\mathcal{L}_{ab} = \{ \omega \mid \omega$ has an equal number of 'a's and 'b's$\}$ is context-free as is the language of balanced parentheses.

We said that the main application of CFGs is in parsing. We decide whether or not a string is well-formed by checking if there is some sequence of rules which generates it.

Consider $\mathcal{L}_{ab} = \{ \omega \mid \omega$ has an equal number of 'a's and 'b's$\}$

$\mathcal{L}_{ab}$ is generated by the grammar

$$S ::= \varepsilon \mid aSb \mid bSa \mid SS$$

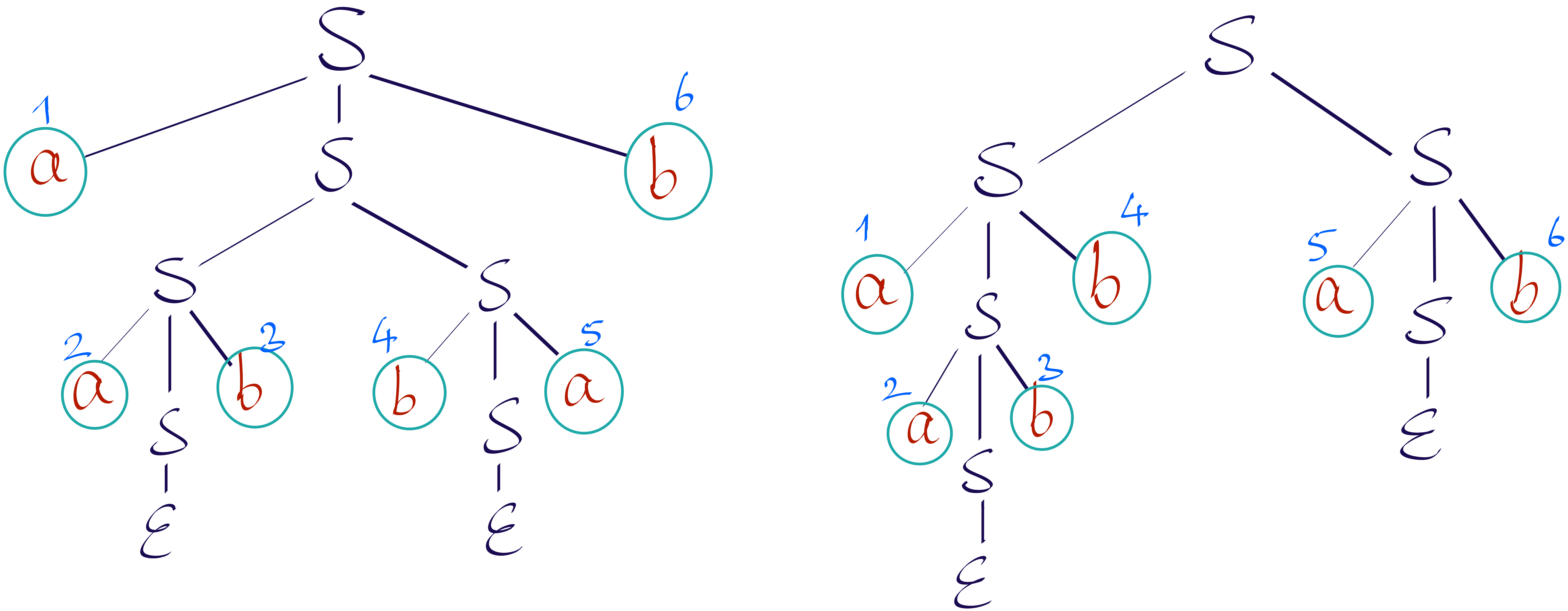
Consider aabbab $\in \mathcal{L}_{ab}$. How might this grammar generate it?

Have to guess the rule which was applied to get this string, and continue recursively!

So suppose we got aabbab by using $S ::= aSb$.

Now I need to check if the grammar can generate abba etc.

The easiest way to keep track of this is via parse trees.



A grammar which can generate multiple parse trees for a string is called ambiguous (otherwise, unambiguous)

The grammar $S ::= \varepsilon \mid aSbS \mid bSaS$ is also ambiguous

To remove ambiguity, one must somehow ensure that matches are unique.

$$S ::= \varepsilon \mid aBS \mid bAS$$

match the first 'b' against this 'a', then the rest

$$B ::= b \mid aBB$$

needs to match two 'b's against two already-read 'a's

$$A ::= a \mid bAA$$

needs to match two 'a's against two already-read 'b's

Proving that a grammar is unambiguous can be difficult!

Can sometimes do induction on strings in the language, but not always!

Much like we provided a machine model for regexes via DFAs/NFAs, we would like a machine model for CFGs as well.

We said that DFAs cannot count, so there was no DFA for $\mathcal{L}_{ab}$, because recognizing $\mathcal{L}_{ab}$, intuitively, required the machine to

- count #'a's
- count #'b's
- check that these numbers were equal.

What is a small extension one can do to a DFA/NFA so it can recognize $\mathcal{L}_{ab}$?

Suppose we add a counter ctr which can increment by one, decrement by one, and check whether it is zero.

Initially: ctr = 0

If you see an 'a', increment ctr

If you see a 'b', decrement ctr (cannot do this if ctr = 0)

If this machine has an accepting run on a string ω, ω has as many 'a's as 'b's.

Exercise: Try to formalize this as a new kind of finite automaton.

Each state needs to track the value of the counter

Cannot decrement at a state if counter is zero.

For a language like $L_{pal} = \{ \omega \cdot rev(\omega) \mid \omega \in \Sigma^* \}$,
it is not clear how to use a single counter to help recognize it.

But what we want is that if we somehow guess the end of ω,
the letter we read last in ω should also be
the letter we read first in whatever follows,
and that this inside-out matching continues till the end.

A stack could help us keep track of this!