Automata Theory and Formal Languages

Sommer Term 2026

15th Lecture

Parsing

Prof. Dr. Matthias Mnich *Institute for Algorithms and Complexity*

Parsing

Deterministic Context Free Grammars

Since PDA and DPDA do not describe the same language class, the question arises, how the context free grammars need to be restricted to describe the deterministic context free languages.

Deterministic Context Free Grammars

Since PDA and DPDA do not describe the same language class, the question arises, how the context free grammars need to be restricted to describe the deterministic context free languages.

$\Rightarrow$ LL(k) and LR(k) grammars.

Deterministic Context Free Grammars

Since PDA and DPDA do not describe the same language class, the question arises, how the context free grammars need to be restricted to describe the deterministic context free languages.

$\Rightarrow$ LL(k) and LR(k) grammars.

LR(k) grammars were introduced by Donald Knuth in 1965 while LL(k) grammars have been proposed by Stearns and Lewis in 1969.

Leftmost and Rightmost Derivation

During a *leftmost derivation* one always replaces the farthermost left non-terminal. Within a *rightmost derivation* one always replaces the farthermost right non-terminal.

Leftmost and Rightmost Derivation

Example

Grammar G with

$$S \to T + S \mid T$$
$$T \to a \mid b \mid c$$

Consider the string $a + b + c$

Leftmost and Rightmost Derivation

Example

Grammar G with

$$S \to T + S \mid T$$
$$T \to a \mid b \mid c$$

Consider the string $a + b + c$

Leftmost Derivation

$$S \to T + S \to a + S \to a + T + S$$
$$\to a + b + S \to a + b + T \to a + b + c$$

Leftmost and Rightmost Derivation

Example

Grammar G with

$$S \to T + S \mid T$$
$$T \to a \mid b \mid c$$

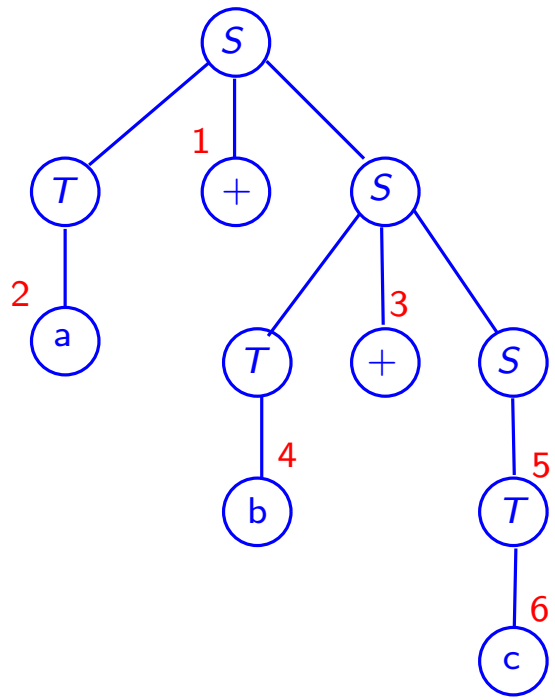
Consider the string $a + b + c$

Rightmost Derivation

$$S \to T + S \to T + T + S \to T + T + T$$
$$\to T + T + c \to T + b + c \to a + b + c$$

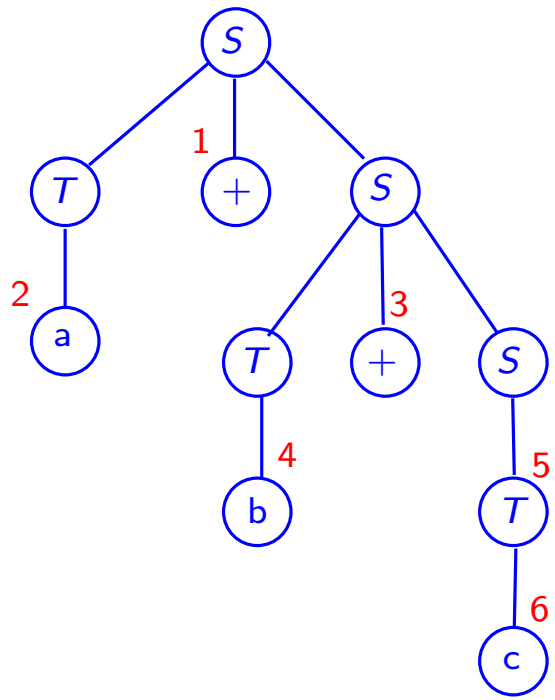
Leftmost and Rightmost Derivation

Leftmost Derivation

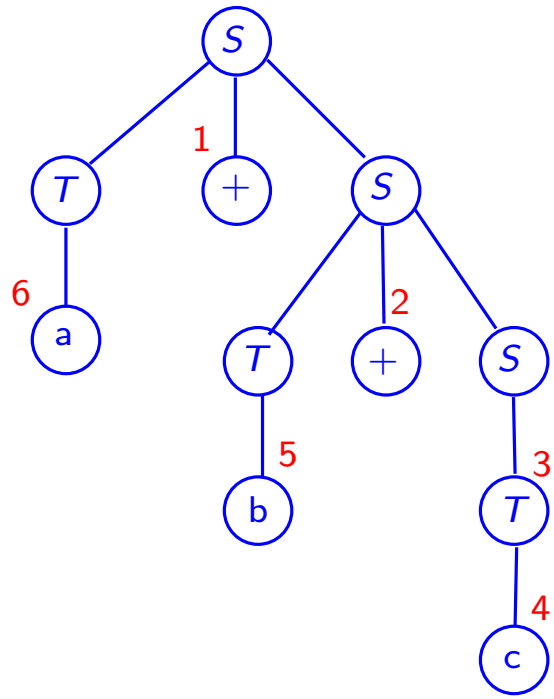


Leftmost and Rightmost Derivation

Leftmost Derivation



Rightmost Derivation



LL(k) and LR(k) Languages/Grammars

- LL(k) and LR(k) grammar allow to parse programs in linear time and they are highly relevant when building a compiler.
- They are both related to the deterministic context-free languages

$$L(NFA) \subset L(LL(k)) \subset L(LR(k)) = L(DPDA) \subset L(PDA)$$

- LL means that the input is read from left to right (first L) and grammar generates leftmost derivations (second L)
- LR means that the input is read from left to right and grammar generates rightmost derivations
- k is the so-called LOOKAHEAD and allows to look k characters ahead when parsing string using the grammar.
- Defining LL(k) and LR(k) grammar require a more advanced definition ($\rightarrow$ compiler theory)

Parsing

In the case of context-free grammars, the production process takes the form of a *parse tree*. What we know about the parse tree is its *root node* and the *leaf nodes*. Parsing is the process of determining what is in between.

Parsing

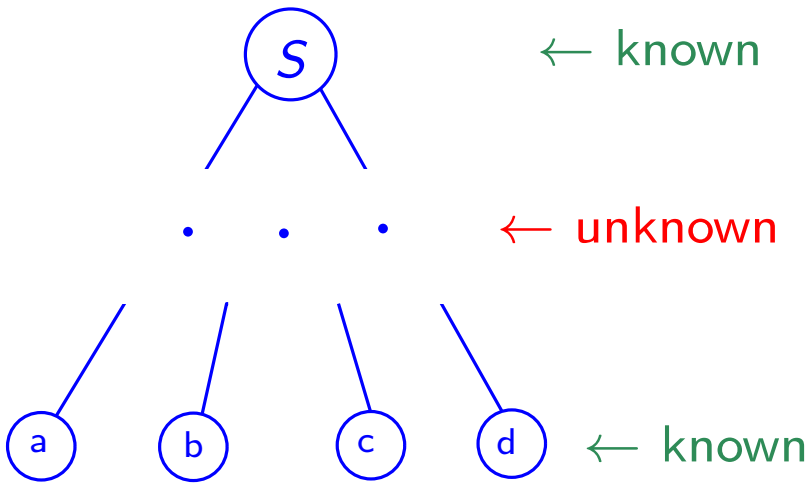
In the case of context-free grammars, the production process takes the form of a *parse tree*. What we know about the parse tree is its *root node* and the *leaf nodes*. Parsing is the process of determining what is in between.

For example if the string is $abcd$ we know already that S is the root node and that a,b,c,d are the leaf nodes.

Parsing

In the case of context-free grammars, the production process takes the form of a *parse tree*. What we know about the parse tree is its *root node* and the *leaf nodes*. Parsing is the process of determining what is in between.

For example if the string is $abcd$ we know already that S is the root node and that a,b,c,d are the leaf nodes.



Top-down vs Bottom-up Parsing

There are several different forms how to parse a deterministic context-free language. We will discuss

- bottom-up parsing
- top-down parsing

Top-down vs Bottom-up Parsing

There are several different forms how to parse a deterministic context-free language. We will discuss

- bottom-up parsing
- top-down parsing

Bottom-up parsing requires an LR(k) grammar, i.e. the grammar should generate rightmost derivations. Top-down parsing requires LL(k) grammars, i.e. the grammar should generate leftmost derivations.

Top-down vs Bottom-up Parsing

There are several different forms how to parse a deterministic context-free language. We will discuss

- bottom-up parsing
- top-down parsing

Bottom-up parsing requires an LR(k) grammar, i.e. the grammar should generate rightmost derivations. Top-down parsing requires LL(k) grammars, i.e. the grammar should generate leftmost derivations.

We will in the following assume that these assumptions hold.

Example Grammar

We will consider the following example grammar:

$$S \to xyz \mid aBC$$
$$B \to c \mid cd$$
$$C \to eg \mid df$$

Bottom-up Parsing

With bottom-up parsing we start at the bottom of the parse tree with the individual characters and try to apply the rules in reverse, with the goal ending up at the start variable S. If it is not possible to apply further rules in reverse and the reduced string is **not** S then we need to *backtrack* and try combining tokens in a different way.

Bottom-up Parsing

With bottom-up parsing we start at the bottom of the parse tree with the individual characters and try to apply the rules in reverse, with the goal ending up at the start variable S. If it is not possible to apply further rules in reverse and the reduced string is **not** S then we need to *backtrack* and try combining tokens in a different way.

With bottom-up parsing one typically maintains a *stack* which keeps track of the characters and variables seen so far. At each step, we *shift* a new character on the stack and try to *reduce* by applying a production in reverse (shift-reduce principle).

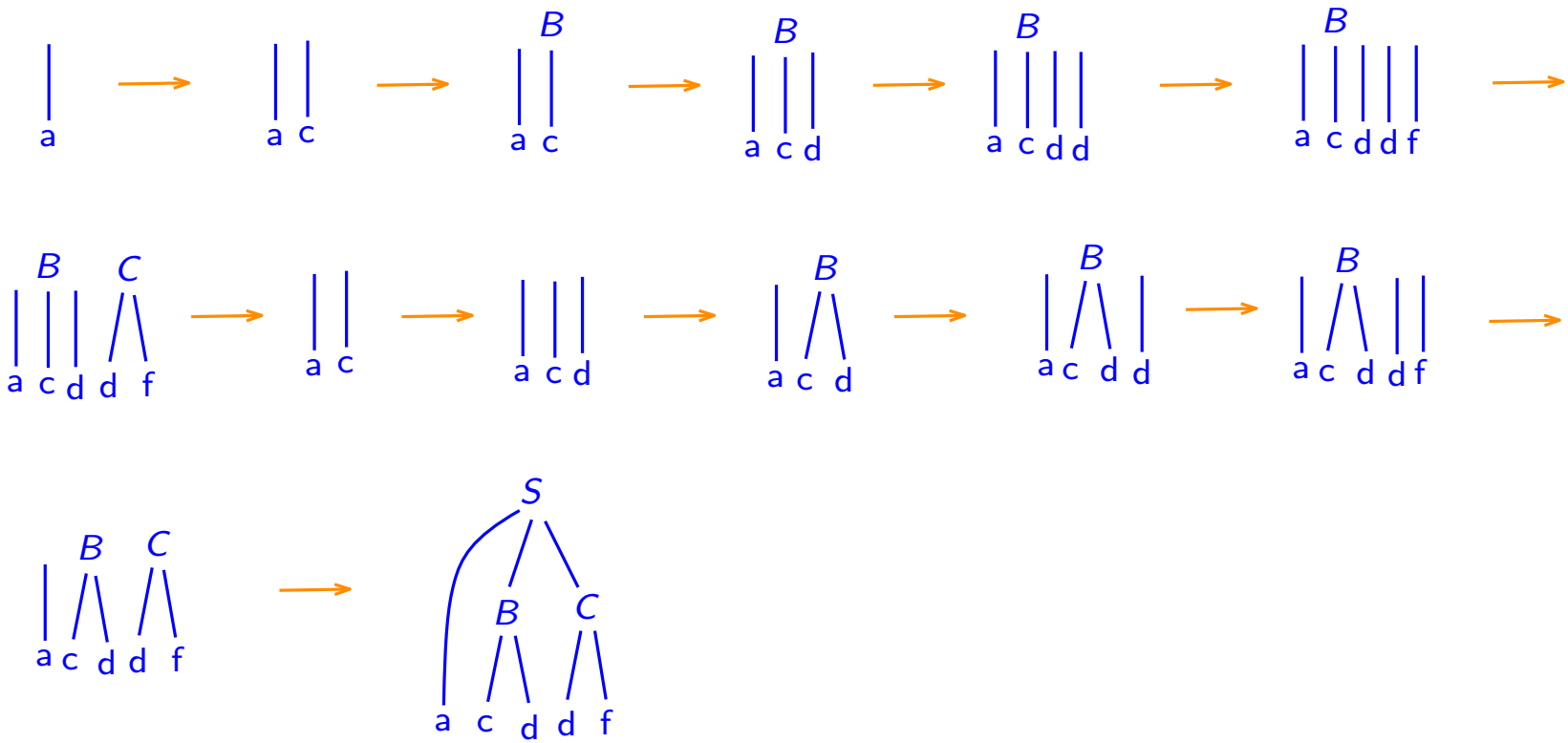
Example - Bottom-up Parsing of acddf

- ε can't be reduced
- a can't be reduced
- ac **can be reduced as follows:**
- reduce ac to aB
  - aB can't be reduced
  - aBd can't be reduced
  - $aBdd$ can't be reduced
  - $aBddf$ **can be reduced as follows:**
  - reduce $aBddf$ to $aBdC$
    - $aBdC$ can't be reduced
    - End of string. Stack is $aBdC \neq S$. **Failure, Must backtrack**
  - $aBddf$ can't be reduced
- ac can't be reduced
- acd **can be reduced as follows:**
- reduce acd to aB
  - aB can't be reduced
  - aBd can't be reduced
  - $aBdf$ **can be reduced as follows:**
  - reduce $aBdf$ to aBC
    - aBC **can be reduced as follows:**
    - reduce aBC to S
    - **End of string. Stack is $S = S$, Success!**

Grammar:

$$S \rightarrow xyz \mid aBC$$
$$B \rightarrow c \mid cd$$
$$C \rightarrow eg \mid df$$

Example - Bottom-up Parsing of acddf



Top-down Parsing

With top-down parsing we start at the top and assume that the string matches S. Then we look at the logical implications of this assumptions.

Top-down Parsing

With top-down parsing we start at the top and assume that the string matches S. Then we look at the logical implications of this assumptions.

For example, the fact that the string $acddf$ matches S implies that either (1) the string matches xyz or (2) the string matches aBC. Since we know that $acddf \neq xyz$ we know that (2) needs to be true (if the string is parsable).

Top-down Parsing

With top-down parsing we start at the top and assume that the string matches S. Then we look at the logical implications of this assumptions.

For example, the fact that the string $acddf$ matches S implies that either (1) the string matches xyz or (2) the string matches aBC. Since we know that $acddf \neq xyz$ we know that (2) needs to be true (if the string is parsable).

(2) has its own logical implication, so that we can proceed in a recursive way. If we end up at the leaf nodes with a true logical statement, we know that the base assumption is true and in turn the string is parseable.

Example - Top-down Parsing of acddf

Grammar:

$S \to xyz \mid aBC$

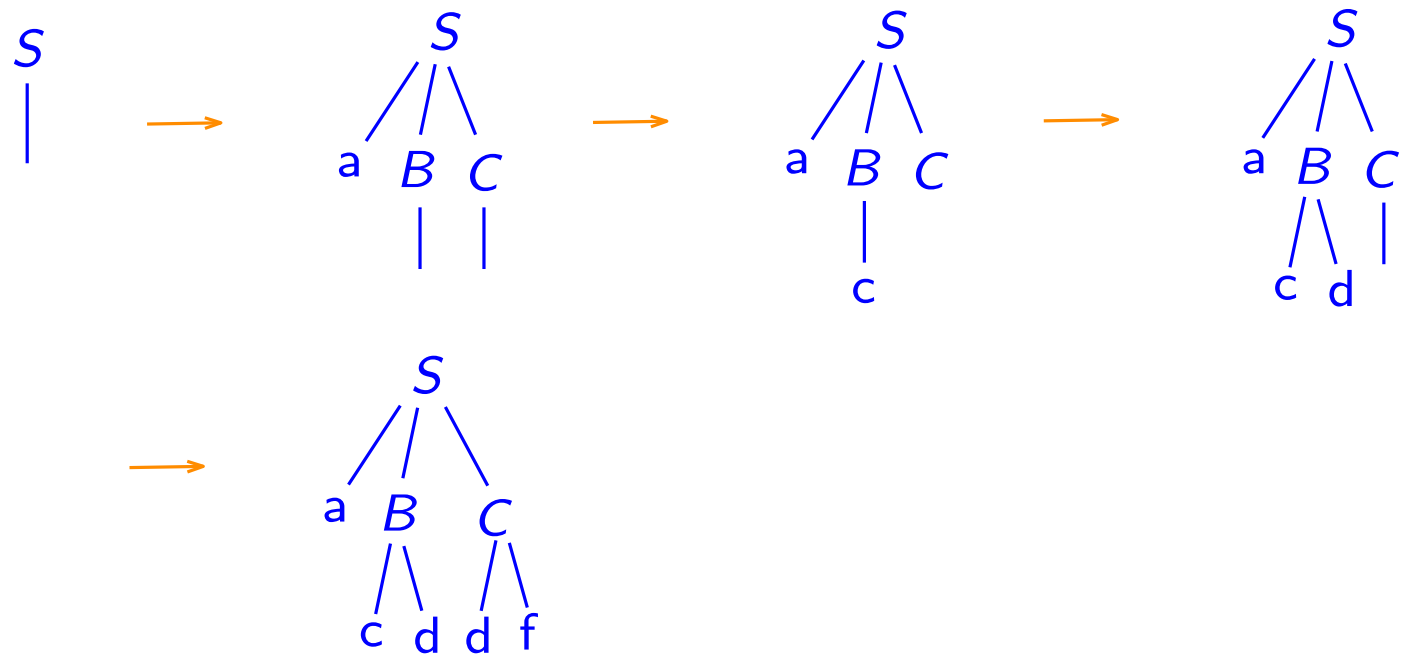
$B \to c \mid cd$

$C \to eg \mid df$

Assertion 1: $acddf$ matches S
- Assertion 2: $acddf$ matches xyz
- **Assertion is false. Try another.**
- Assertion 2: $acddf$ matches aBC, i.e. $cddf$ matches BC
  - Assertion 3: $cddf$ matches cC ($B = c$), i.e. ddf matches C
    - Assertion 4: ddf matches eg
    - **Assertion is false. Try another.**
    - Assertion 4: ddf matches df
    - **Assertion is false.**
  - **Assertion 3 is false. Try another.**
  - Assertion 3: $cddf$ matches cdC ($B = cd$), i.e. df matches C
    - Assertion 4: df matches eg
    - **Assertion is false. Try another.**
    - Assertion 4: df matches df
    - **Assertion is true.**
  - **Assertion 3 is true.**
- **Assertion 2 is true.**

Assertion 1 is true.

Example - Top-down Parsing of acddf



Top-down parsers - Left recursion

Top-down parsers have a problem with grammars having a left-recursion, since it leads to infinite recursion.

Top-down parsers - Left recursion

Top-down parsers have a problem with grammars having a left-recursion, since it leads to infinite recursion.

Lets assume the previous example grammar would have a rule

$$S \rightarrow Sb$$

Top-down parsers - Left recursion

Top-down parsers have a problem with grammars having a left-recursion, since it leads to infinite recursion.

Lets assume the previous example grammar would have a rule

$$S \rightarrow Sb$$

Assertion 1: $acddf$ matches S
- Assertion 2: $acddf$ matches Sb
  - Assertion 3: $acddf$ matches Sbb
    - Assertion 4: $acddf$ matches $Sbbb$
      - ... and so on forever