

CS_275 Automata and Formal Language Theory

Course Notes

Part II: The Recognition Problem (II)

Chapter II.5.: Properties of Context Free Grammars (14)

Anton Setzer

(Based on a book draft by J. V. Tucker and K. Stephenson)

Dept. of Computer Science, Swansea University

[http://www.cs.swan.ac.uk/~csetzer/lectures/
automataFormalLanguage/current/index.html](http://www.cs.swan.ac.uk/~csetzer/lectures/automataFormalLanguage/current/index.html)

April 19, 2018

II.5.1. Derivation Trees for Context-Free Grammars (14.1)

II.5.2. Uniqueness of Derivation Trees (14.1)

II.5.4. The Pumping Lemma for CFG (14.4)

II.5.5. Floyd's Theorem

II.5.1. Derivation Trees for Context-Free Grammars (14.1)

II.5.2. Uniqueness of Derivation Trees (14.1)

II.5.4. The Pumping Lemma for CFG (14.4)

II.5.5. Floyd's Theorem

Derivation Trees or Parse Trees

- ▶ Context free Grammars (abbreviated as CFG in the following) allow to apply to a non-terminal at position without needing the context.
- ▶ Therefore we can expand the non-terminals independently of each other.
- ▶ This allows us to define derivation trees (also called parse trees).

Example

Consider the grammar

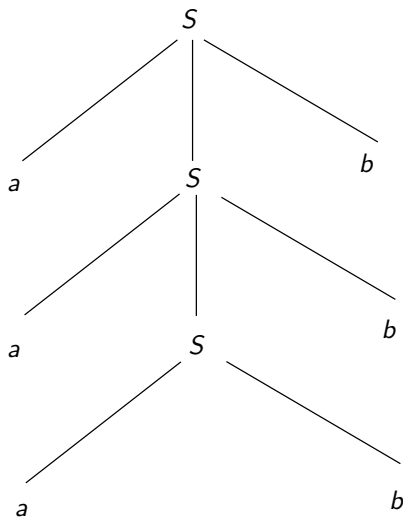
grammar	G
terminals	a, b
nonterminals	S
start symbol	S
productions	$S \rightarrow aSb$ $S \rightarrow ab$

Example Derivation

We derive *aaabbbb* in it:

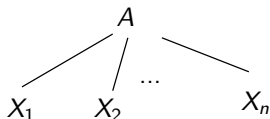
$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaabbbb \end{aligned}$$

Derivation Tree

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$


Form of the Derivation Tree

- ▶ Nodes are labelled with elements of $N \cup T \cup \{\epsilon\}$.
- ▶ A node with label A has a subtree



only if A is a non-terminal and there is a production

$$A \longrightarrow X_1 X_2 \cdots X_n$$

where $X_i \in T \cup N$.

- ▶ All leaves of the tree together read from left to right form the string derived, namely $aaabbb$.

This is called the **frontier** of the derivation tree.

- ▶ We will as well consider derivation trees not ending in a string of terminals, so the frontier is an element of $(T \cup N)^*$.

Definition Derivation Tree

Definition

Let $G = (T, N, S, P)$ be a CFG. A derivation tree or parse tree for G is a finite tree with

- ▶ nodes labelled by elements of $N \cup T \cup \{\epsilon\}$,
- ▶ s.t. a node A has children with labels X_1, \dots, X_n only if $A \in N$ and there is a production

$$A \longrightarrow X_1 X_2 \cdots X_n$$

- ▶ If the node of one of the children of A is ϵ , then this node is the only child of this tree.

The frontier of the tree is the set of leaves read from left to right in sequence, which is an element $(T \cup N)^*$.

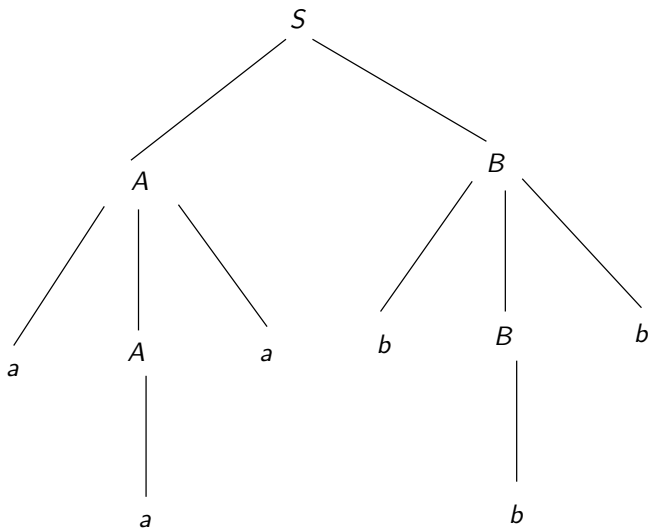
The root of the tree is the node at the top of the derivation tree.

Left-Most and Right-Most Derivations

From a derivation tree we can obtain a derivation in various orders.
Consider the grammar

grammar	G
terminals	a, b
nonterminals	S, A, B
start symbol	S
productions	$S \longrightarrow AB,$ $A \longrightarrow aAa, A \longrightarrow a$ $B \longrightarrow bBb, B \longrightarrow b$

Example Derivation Tree



Different Derivations of $aaabbb$

We can derive $aaabbb$ in different ways:

$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaabBb \Rightarrow aaabbb$$

A left most derivation

$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow Abbb \Rightarrow aAabbb \Rightarrow aaabbb$$

A right most derivation

$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAabBb \Rightarrow aaabBb \Rightarrow aaabbb$$

$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAabBb \Rightarrow aAabbb \Rightarrow aaabbb$$

$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow aAabBb \Rightarrow aaabBb \Rightarrow aaabbb$$

$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow aAabBb \Rightarrow aAabbb \Rightarrow aaabbb$$

Left-Most Derivation

Definition

Let $G = (T, N, S, P)$ be a CFG.

A single-step derivation $w \Rightarrow w'$ is left-most if a rule was applied to the left-most non-terminal in w , i.e.

- ▶ $w = sAt$ for some $A \in N$, $s \in T^*$ (consisting only of terminals), $t \in (S \cup T)^*$,
- ▶ and there exist a production $A \rightarrow v$
- ▶ s.t. $w' = svv$.

Left-Most Derivation

Definition

Let $G = (T, N, S, P)$ be a CFG.

A single-step derivation $w \Rightarrow w'$ is right-most if a rule was applied to the right-most non-terminal in w , i.e.

- ▶ $w = sAt$ for some $A \in N$, $s \in (S \cup T)^*$, $t \in T^*$ (consisting only of terminals),
- ▶ there exist a production $A \rightarrow v$
- ▶ s.t. $w' = svv$.

Left/Right-Most Derivation Sequence

Definition

Let $G = (T, N, S, P)$ be a CFG

1. A derivation sequence $w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots w_n$ is left-most, if each derivation step $w_i \Rightarrow w_{i+1}$ is left-most.
2. Right-most derivation sequences are defined analogously.

Theorem II.5.1.1. (Derivation Trees and Language Generation)

Theorem

Let $G = (T, N, S, P)$ be a CFG, $A \in T$, $w, w' \in (T \cup N)^*$, Then the following are equivalent

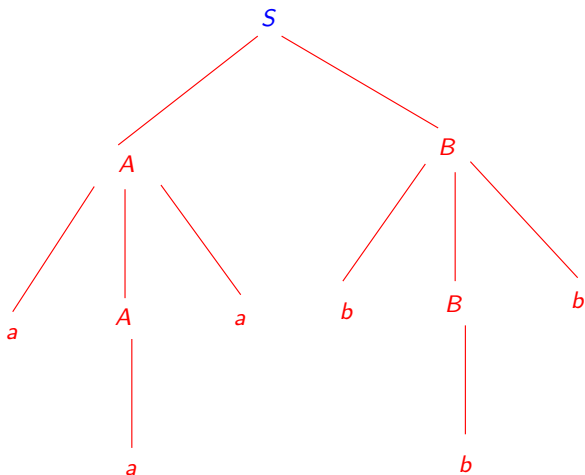
- (1) There exist a derivation tree with root labelled by A and frontier w' .
- (2) $A \Rightarrow^* w'$.

In case $w' \in T^*$, the derivation sequence $w \Rightarrow^* w'$ can both be chosen as a left-most and as a right-most derivation sequence

Proof of Theorem II.5.1.1.

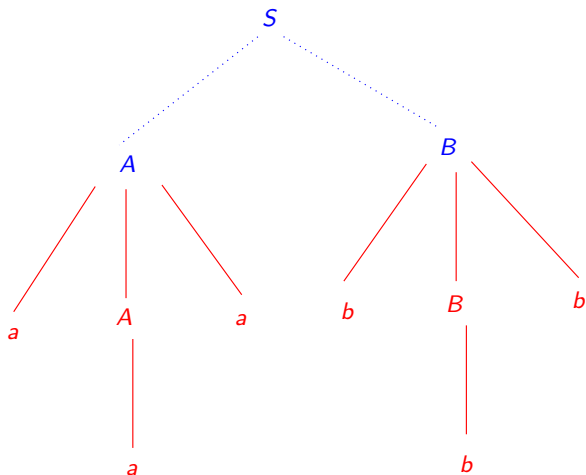
- ▶ A proof of this theorem can be found in the additional material.
- ▶ We illustrate this theorem by an example.
 - ▶ We will first present a left-most derivation.
 - ▶ Then we will present a right most derivation.

Example Left-Most Derivation (Step 1)

 S 

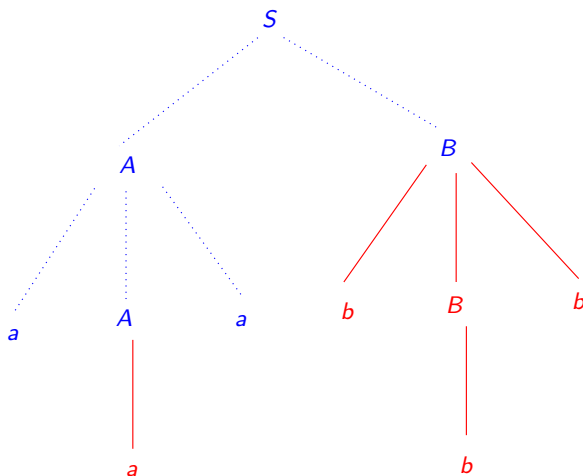
Example Left-Most Derivation (Step 2)

$$S \Rightarrow AB$$



Example Left-Most Derivation (Step 3)

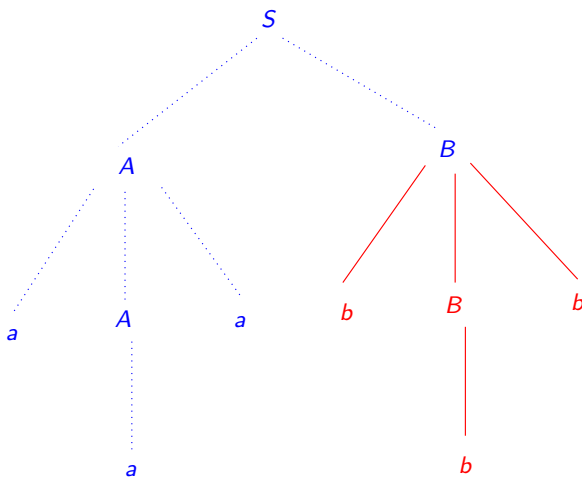
$$S \Rightarrow AB \Rightarrow aAaB$$



Derivation tree for a is trivial.

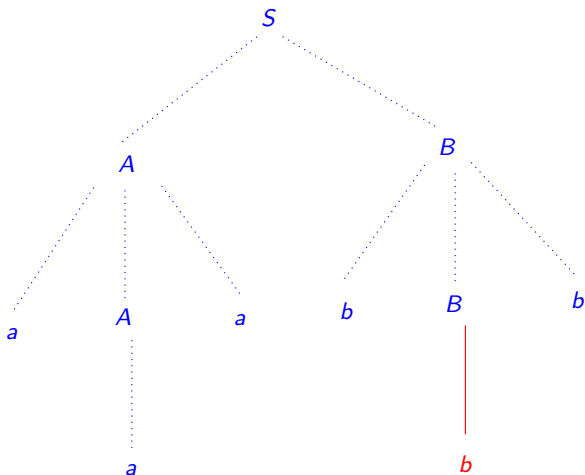
Example Left-Most Derivation (Step 4)

$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB$$

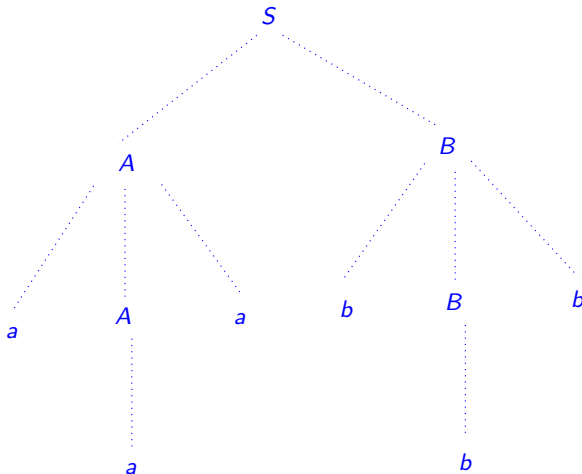


Example Left-Most Derivation (Step 5)

$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaabBb$$

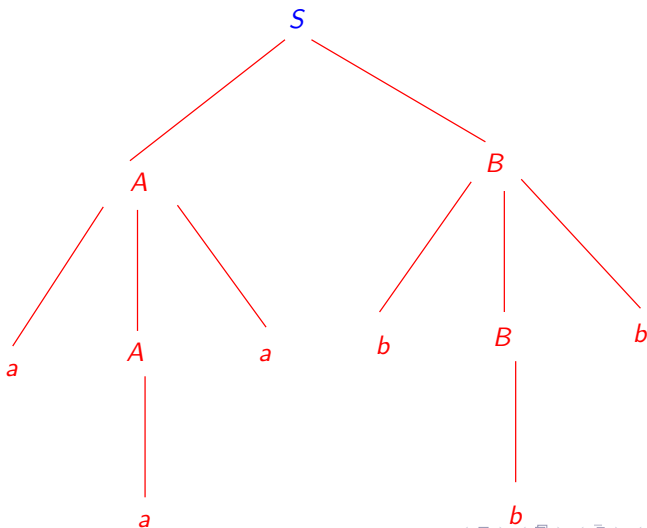


Example Left-Most Derivation (Step 6)

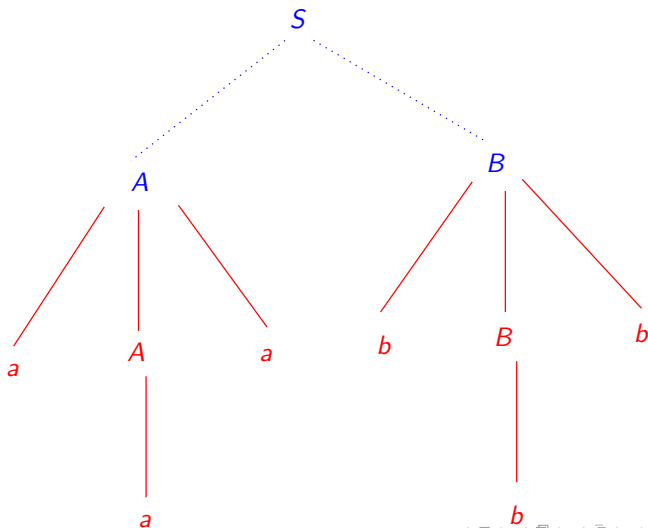
$$S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaabBb \Rightarrow aaabbb$$


Final derivation.

Example Right-Most Derivation (Step 1)

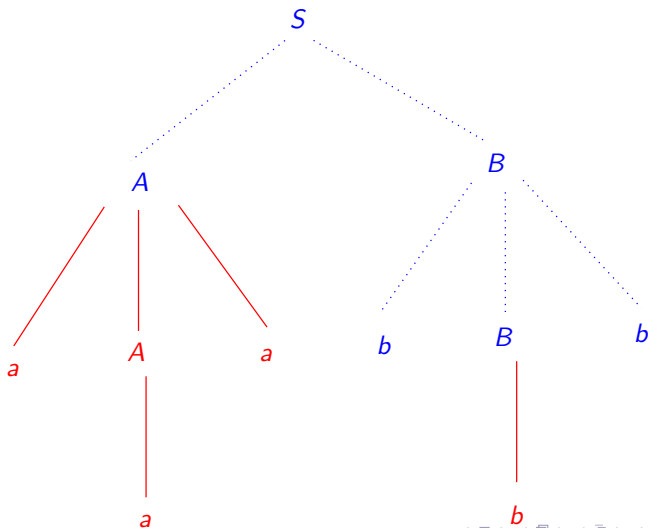
 S 

Example Right-Most Derivation (Step 2)

 $S \Rightarrow AB$ 

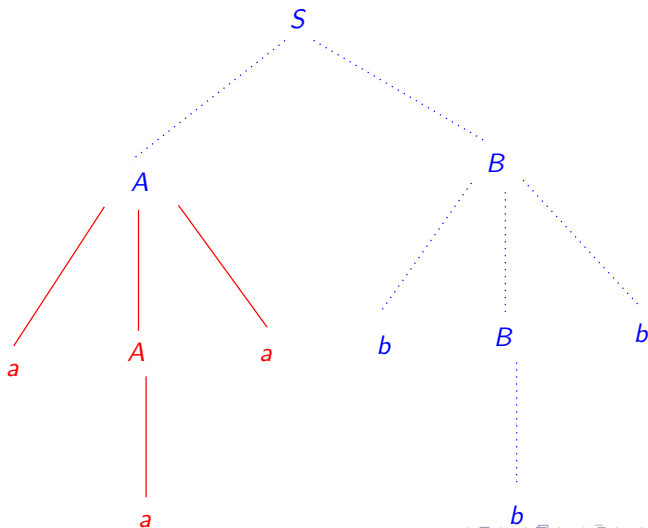
Example Right-Most Derivation (Step 3)

$$S \Rightarrow AB \Rightarrow AbBb$$

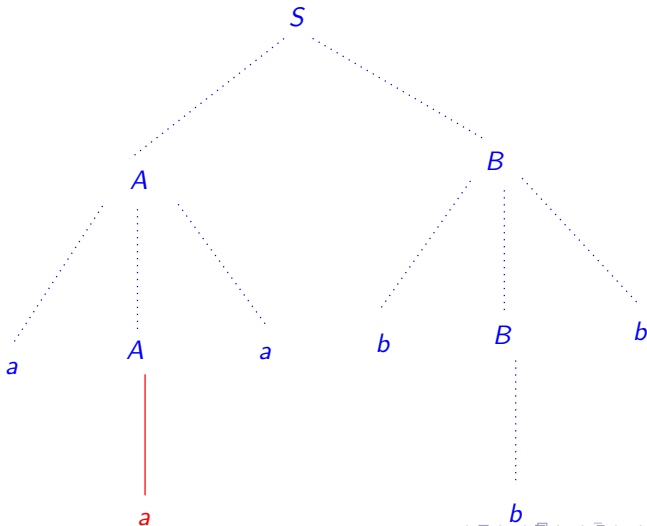


Example Right-Most Derivation (Step 4)

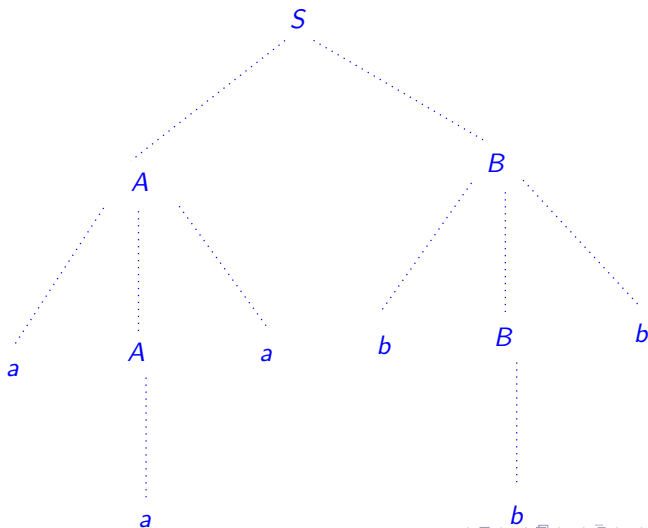
$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow Abbb$$



Example Right-Most Derivation (Step 5)

$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow Abbb \Rightarrow aAabbb$$


Example Right-Most Derivation (Step 6)

$$S \Rightarrow AB \Rightarrow AbBb \Rightarrow Abbb \Rightarrow aAabbb \Rightarrow aaabbb$$


II.5.1. Derivation Trees for Context-Free Grammars (14.1)

II.5.2. Uniqueness of Derivation Trees (14.1)

II.5.4. The Pumping Lemma for CFG (14.4)

II.5.5. Floyd's Theorem

Theorem II.5.2.4 Uniqueness of Derivation (Trees)

Theorem (II.5.2.4)

Let $G = (T, N, S, P)$ be a CFG, $w \in T^*$. The following are equivalent:

- (1) There exist exactly one derivation tree with label S and frontier w .
- (2) There exist exactly one left-most derivation sequence $S \Rightarrow^* w$.
- (3) There exist exactly one right-most derivation sequence $S \Rightarrow^* w$.

Proof: See Additional Material.

Ambiguous Grammars

Definition

A CFG $G = (T, N, S, P)$ is ambiguous, if there is a string $w \in L(G)$ having more than one derivation tree (or, equivalently, having more than one left-most or more than one right-most derivation).

Example 1

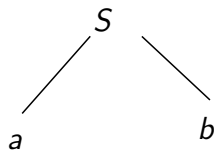
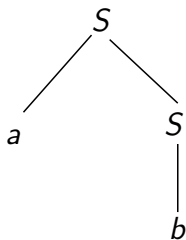
grammar	G
terminals	S
nonterminals	a, b
start symbol	S
productions	$S \rightarrow aS$ $S \rightarrow b$ $S \rightarrow ab$

Example 1

There are two left-most derivations of ab :

$$S \Rightarrow aS \Rightarrow ab \text{ and } S \Rightarrow ab$$

And two derivation trees:



Example 2: Dangling Else

Assume the following grammar which is a cut down version of the grammar G^{while} introduced in I.2.4 with **if_then_else_fi** replaced by **if_then** and a **if_then_else_**):

grammar	$G^{Dangling_else}$
import	$G^{Identifier}, G^{Arithmetic_Expression}, G^{Boolean_Expression}$
terminals	if, then, else, :=
nonterminals	<i>Program</i>
start symbol	<i>Program</i>
productions	$Program \rightarrow Id := AExp$ $Program \rightarrow \mathbf{if} BExp \mathbf{then} Program \mathbf{else} Program$ $Program \rightarrow \mathbf{if} BExp \mathbf{then} Program$

“import” means that we add all the ingredients of those grammars, including the terminals. The grammars $G^{Identifier}$, $G^{Arithmetic_Expression}$, $G^{Boolean_Expression}$ have start symbols Id , $AExp$, $BExp$, respectively.

Example 2: Dangling Else

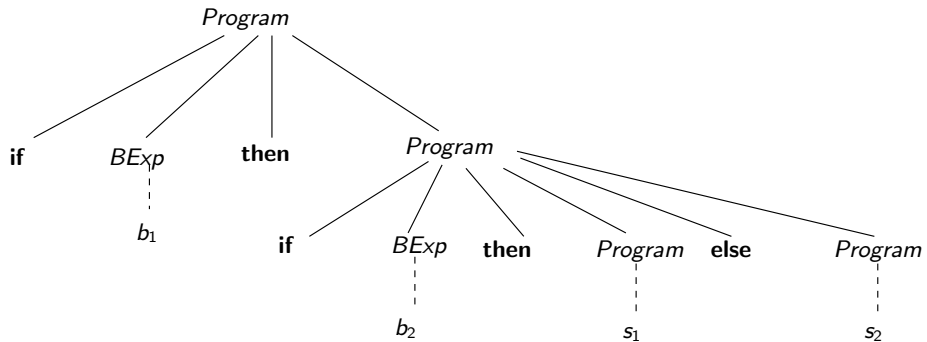
Assume strings b_1, b_2 deriving from $BExp$ and string s_1, s_2 deriving from $Program$.

The string

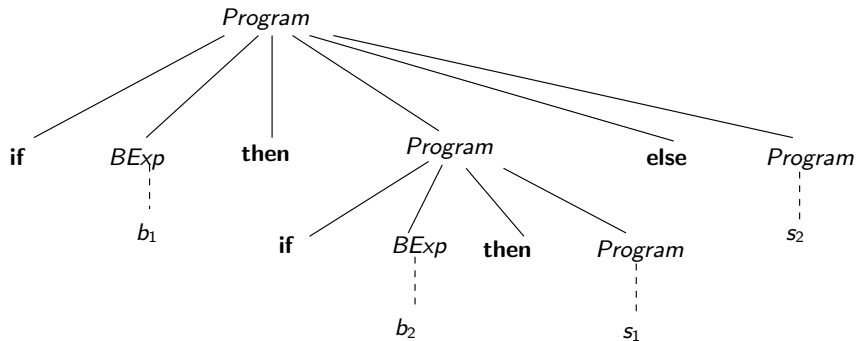
if b_1 then if b_2 then s_1 else s_2

has two derivation trees (we omit the derivation trees for b_i, s_j .)

First Derivation Tree



Second Derivation Tree



Different Interpretations of the Program

The two different derivation trees of the program

if b_1 then if b_2 then s_1 else s_2

correspond to two different ways of executing the program:

Execution following the Derivation Tree 1

- ▶ In the first the else case belongs to the second if. It is executed if b_1 is true and b_2 is false.

The program can be using suggestive indentation be written as follows:

```
if  $b_1$  then
    if  $b_2$  then
         $S_1$ 
    else
         $S_2$ 
```


Execution following the Derivation Tree 2

- ▶ In the second derivation tree, the else case belongs to the first if. It is executed if b_1 is false.

The program can be using suggestive indentation be written as follows:

```
if  $b_1$  then
    if  $b_2$  then
         $s_1$ 
    else
         $s_2$ 
```

2 Solutions for Solving the Problem

There are 2 solutions for solving this problem.

The first solution is to add to **if_then** and **if_then_else** a symbol **fi** (or some other keyword such as **endif**) labelling the end of the statement.

grammar	$G^{Unambiguous_if}$
import	$G^{Identifier}, G^{Arithmetic_Expression}, G^{Boolean_Expression}$
terminals	if, then, else, :=
nonterminals	<i>Program</i>
start symbol	<i>Program</i>
productions	$Program \rightarrow Id := AExp$ $Program \rightarrow \mathbf{if} BExp \mathbf{then} Program \mathbf{else} Program \mathbf{fi}$ $Program \rightarrow \mathbf{if} BExp \mathbf{then} Program \mathbf{fi}$

Solution 1

Now the two interpretations of the original string would be written in as two different strings:

- ▶ “Else” belonging to the second “if” is written as

$$\textit{if } b_1 \textbf{ then } \textit{if } b_2 \textbf{ then } s_1 \textit{else } s_2 \textit{ fi fi}$$

- ▶ “Else” belong to the first “if” is written as

$$\textit{if } b_1 \textbf{ then } \textit{if } b_2 \textbf{ then } s_1 \textit{ fi else } s_2 \textit{ fi}$$

- ▶ This solution has been taken for instance in Algol, in the bash shell (Linux), and in Ada (where *fi* is replaced by “end if”).
- ▶ A similar solution was taken in Java and some other languages: they require that the subprograms of an **if_then_** or **if_then_else_** are enclosed by brackets $\{\dots\}$.

Solution 2

- ▶ The 2nd solution is to modify the grammar so that the derivation tree will be possible only for one of the two choices.
- ▶ For this we modify the grammar so that the statement s_1 in

if b_1 then s_1 else s_2

is not matched by

if b'_1 then s'_1

but only by

if b'_1 then s'_1 else s'_2

- ▶ This solution has been taken in most other programming languages.

Solution 2

- ▶ For this we split Programs into two categories:
 - ▶ Those derived from MatchedIf. In a program deriving from MatchedIf, each **if** is matched by an **else** clause.
 - ▶ Those derived from UnmatchedIf. These have at least one **if** with no matching **else** clause.

Solution 2

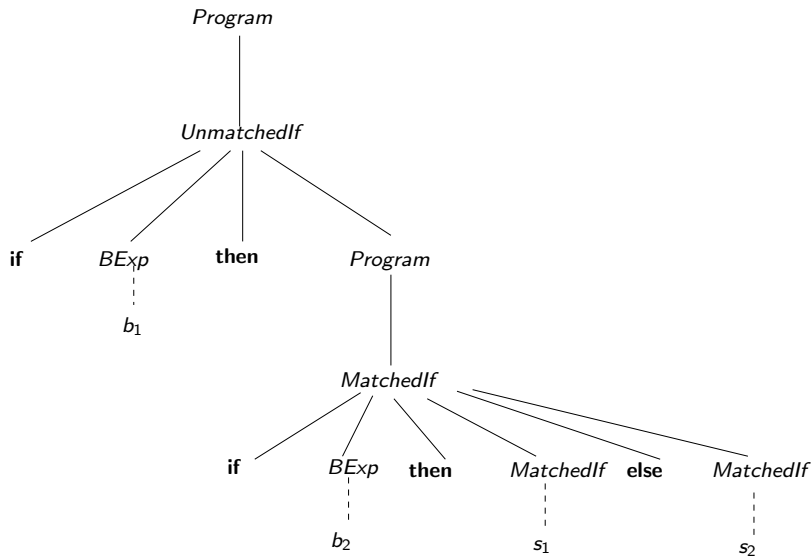
- ▶ The grammar will make sure that a **else** will always be associated with the first **if** to the left, which has no unmatched **else** yet.
- ▶ So **if_then_***mathbf{else}* expression will be parsed as in the first derivation tree.

Solution 2

Here is the grammar:

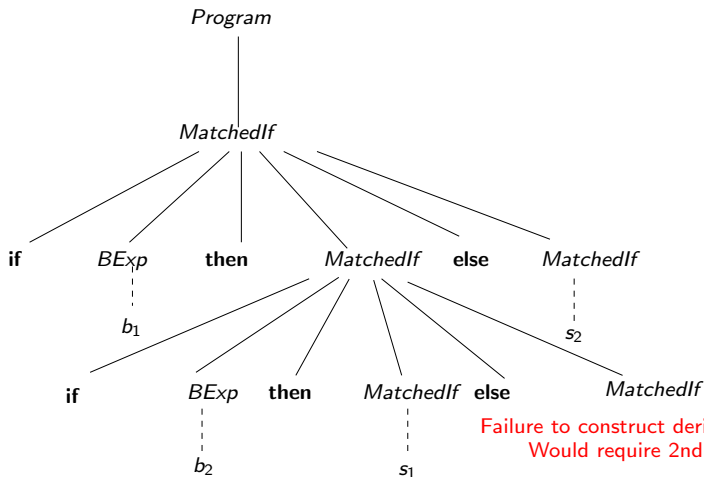
grammar	$G^{Dangling_Else}$
import	$G^{Identifier}, G^{Arithmetic_Expression}, G^{Boolean_Expression}$
terminals	if, then, else, :=
nonterminals	<i>Program</i>
start symbol	<i>Program</i>
productions	<i>Program</i> \longrightarrow <i>UnmatchedIf</i>
	<i>Program</i> \longrightarrow <i>MatchedIf</i>
	<i>MatchedIf</i> \longrightarrow <i>Id := AExp</i>
	<i>MatchedIf</i> \longrightarrow if <i>BExp</i> then <i>MatchedIf</i> else <i>MatchedIf</i>
	<i>UnmatchedIf</i> \longrightarrow if <i>BExp</i> then <i>Program</i>
	<i>UnmatchedIf</i> \longrightarrow if <i>BExp</i> then <i>MatchedIf</i> else <i>UnmatchedIf</i>

Unique Derivation Tree 2nd Solution



Failure of Starting Derivation Tree with MatchedIf

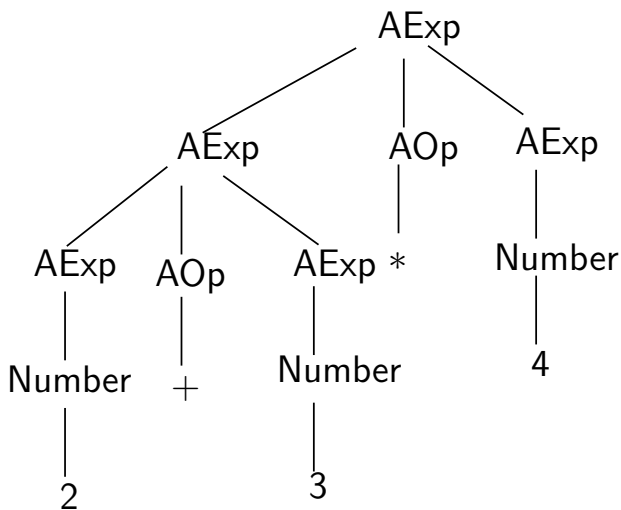
Trying to construct derivation tree for expression which matches else to first if fails:

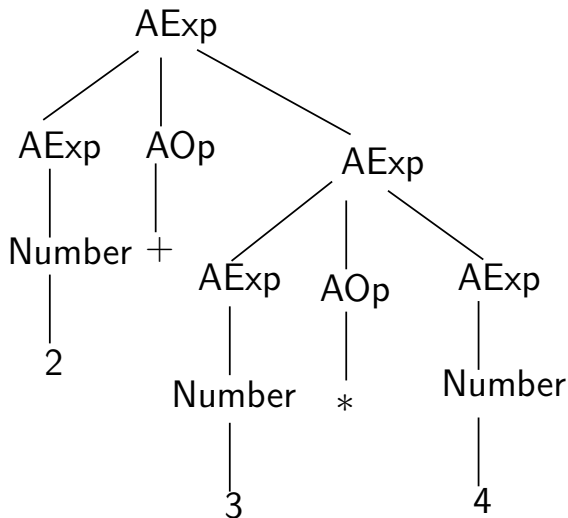


Example: Grammar for Arithmetic Expressions

Remember the grammar for arithmetic expressions
(using elements of BNF notation)

grammar	$G^{Arithmetic_Expression}$
import	$G^{Identifier}, G^{Number}$
terminals	$+, -, *, /, (,)$
nonterminals	$AExp, AOp$
start symbol	$AExp$
productions	$AExp \longrightarrow Id \mid Number$ $AExp \longrightarrow (AExp)$ $AExp \longrightarrow AExp AOp AExp$ $AOp \longrightarrow + \mid - \mid * \mid /$

First Parse tree for $2 + 3 * 4$ 

Second Parse tree for $2 + 3 * 4$ 

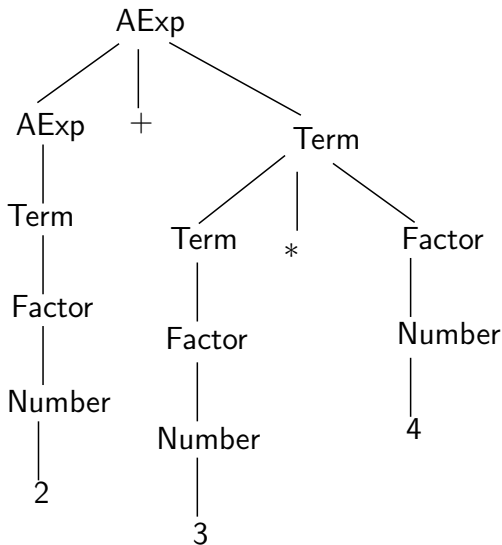
Difference in Evaluation

- ▶ The first parse tree corresponds to parsing it as if it were $(2 + 3) * 4$
Evaluation will return 20.
- ▶ The second parse tree corresponds to parsing it as if it were $2 + (3 * 4)$
Evaluation will return 14.

Unambiguous Version

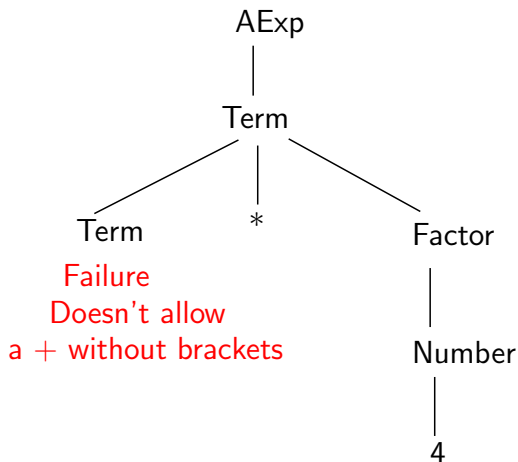
grammar	$G_{\text{unambiguous}}^{\text{Arithmetic_Expression}}$
import	$G^{\text{Identifier}}, G^{\text{Number}}$
terminals	$+, -, *, /, (,)$
nonterminals	$AExp, Term, Factor$
start symbol	$AExp$
productions	$AExp \longrightarrow AExp + Term \mid AExp - Term \mid Term$ $Term \longrightarrow Term * Factor \mid Term / Factor \mid Factor$ $Factor \longrightarrow Id \mid Number \mid (AExp)$

Unique Parse Tree for $2 + 3 * 4$ in $G_{\text{Arithmetic_Expression}}$ *unambiguous*



Failure to Parse $2 + 3 * 4$ with $*$ Binding Stronger Than $+$

Trying to construct derivation tree for expression which parses $2 + 3 * 4$ as $2 + 3 * 4$ fails:



Making Context Free Grammars Unambiguous

The following is known about Context Free Grammars:

- ▶ There are languages defined by context free grammars which **cannot be defined by an unambiguous grammar**.
 - ▶ Context free grammars, for which there exist no equivalent unambiguous grammar, are called inherently ambiguous grammars.
See Hopcroft/Motwani/Ullman [HMU07], 5.4.4, p. 213.
- ▶ It is **undecidable** whether a grammar is **ambiguous**. (Same book, 7.4.5, p. 307.)
- ▶ It is **undecidable** whether a grammar is **inherently ambiguous**. (Same book, 7.4.5 and 9.5.2, p. 413).

II.5.2. Normal Forms for Context-Free Grammars (14.2)

This section has been moved to “Additional Material”.

II.5.1. Derivation Trees for Context-Free Grammars (14.1)

II.5.2. Uniqueness of Derivation Trees (14.1)

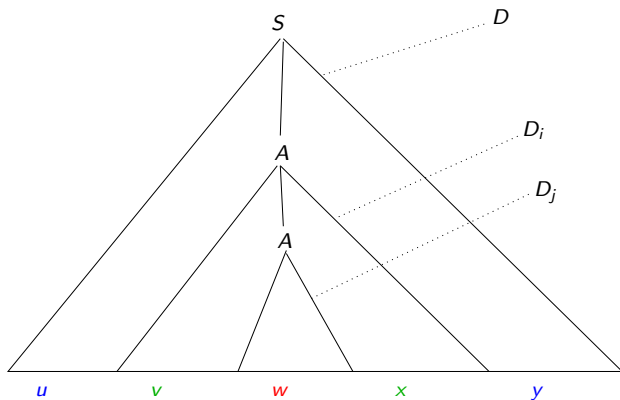
II.5.4. The Pumping Lemma for CFG (14.4)

II.5.5. Floyd's Theorem

Idea of the Pumping Lemma for CFG

- ▶ The **Pumping Lemma for Regular Languages** is based on the fact that if a string derived by a finite state automaton is sufficiently big, we pass **through at least one state twice**.
- ▶ We could equivalently have used the fact that in a **left- or right-linear grammar**, if a string derived is sufficiently big, **we pass through one non-terminal at least twice**.
- ▶ For **CFG**, we need that if a string derived is sufficiently big, we **pass through one non-terminal** at least **twice**.
- ▶ However, if this **occurrence** is **in two disjoint subtrees** of the derivation trees, there is **no relation between them**.
- ▶ What we need that there is a **path in the subtree** which **passes through one non-terminal at least twice**.

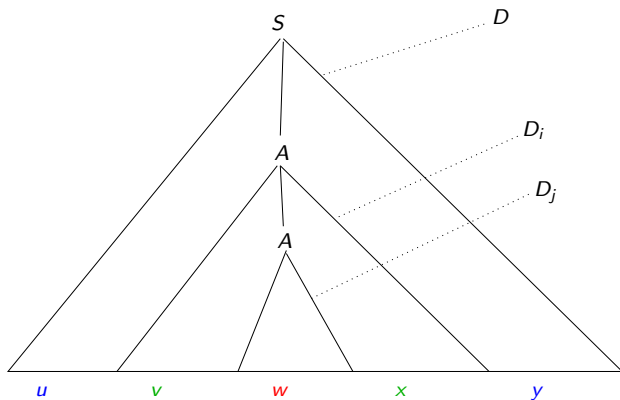
Picture



Idea of the Pumping Lemma for CFG

- ▶ **If** in a derivation tree of a CFG with l non-terminals there is **no repetition of a non-terminal in any path, then** the **tree can have height at most l** .
 - ▶ Here the height of a tree consisting of the root only is defined as 0.
- ▶ Since at any node of a derivation tree there are only finitely many rules to apply to, one can easily see that there are only **finitely many derivation trees in a CFG with height at most $l + 1$** and arbitrary non-terminal as root.
- ▶ Let k' be the **maximum length of any string derived from any non-terminal with height at most $l + 1$** . Let $k := k' + 1$.
- ▶ Assume a derivation of a string z with $|z| \geq k$.
- ▶ We can omit in the derivation any subderivations where $A \Rightarrow^* A$ and this derivation takes more than one step.
- ▶ The derivation must have **height $\geq l + 1$** , and therefore contain a **subderivation of height exactly $l + 1$ of a string from some non-terminal**.

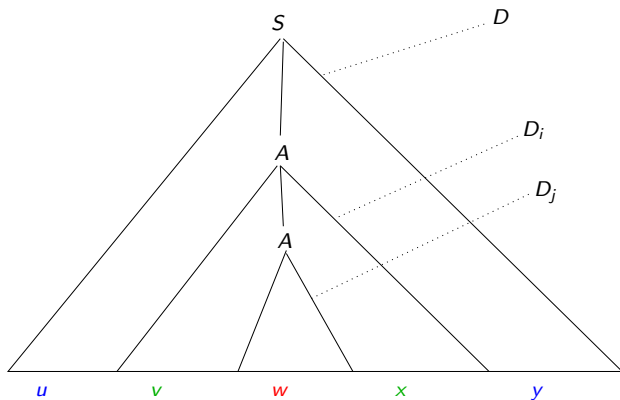
Picture



Idea of the Pumping Lemma for CFG

- ▶ In this subderivation there is a path from the root to a terminal, in which at least one non-terminal A occurs twice.
- ▶ And we can have that subtree starting with upper occurrence of A has height $\leq l + 1$.
- ▶ Therefore the string derived from that A has length at most $k' < k$.
- ▶ Let
 - ▶ w be the string deriving from the lower A ,
 - ▶ vwx be the string deriving from the upper A , with v and x deriving to the left and right of the lower A ,
 - ▶ $z = uvwxy$, with u , y , deriving to the left and right of the upper A ,
- ▶ Then $|vwx| \leq k' < k$.

Picture

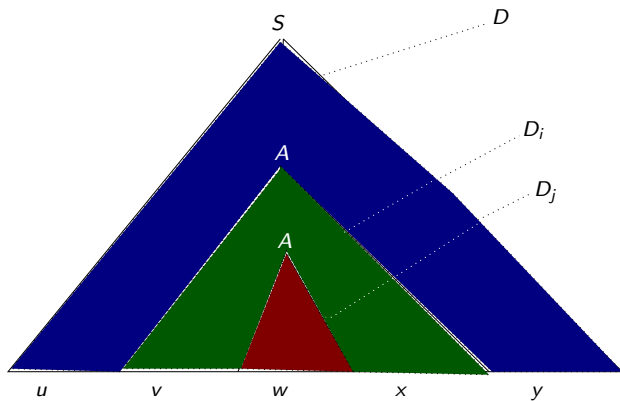


Idea of the Pumping Lemma for CFG

So for any CFG G we can find a constant k s.t.

- ▶ in any derivation tree of a word $z \in L(G)$ s.t. $|z| \geq k$,
- ▶ we can
 - ▶ decompose $z = uvwxy$
 - ▶ find a nonterminal A ,
 - ▶ and derivations
 - ▶ $S \Rightarrow^* uAy$ (written blue on the next slide),
 - ▶ $A \Rightarrow^* vAx$ (written green on the next slide),
 - ▶ $A \Rightarrow^* w$ (written red on the next slide)
 - ▶ The subderivation $A \Rightarrow^* vAx$ plays the role of the loop we had in the pumping lemma for regular languages.
 - ▶ Furthermore the middle part vwx can be chosen to be of length $\leq k$.
 - ▶ $vx \neq \epsilon$ since we omitted subderivations $A \Rightarrow^* A$ taking more than one step.
- ▶ The following pictures don't **come out well on the black and white** handouts. Please look at them on the **online version**.

Picture



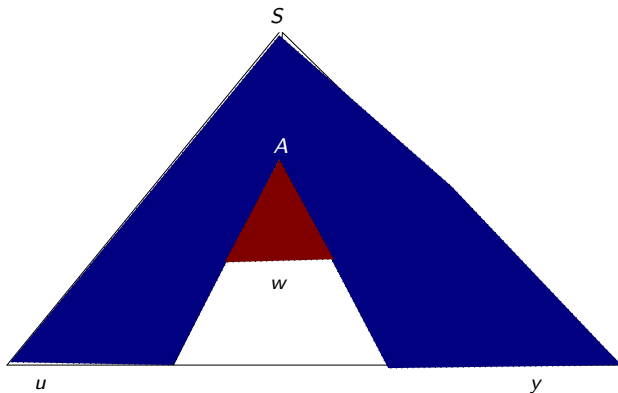
Idea of the Pumping Lemma for CFG

- ▶ Now we can repeat the subderivation $A \Rightarrow^* vAx$ several times and obtain

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxxy \Rightarrow^* \dots \Rightarrow^* uv^iAx^i y \Rightarrow^* uv^i vx^i y$$

- ▶ And therefore we obtain that for all $i \geq 0$ we have $uv^i vx^i y \in L(G)$

Pumping it down to uv^0vx^0yy



Pumping Lemma for CFG

Theorem

Let L be a context free language. Then there exists a constant k s.t. for all strings z of L s.t. $|z| \geq k$ there exist u, v, w, x, y s.t.

- ▶ $z = uvwxy$,
- ▶ $|vwx| \leq k$, i.e. the middle portion is not too long,
- ▶ $|vx| \geq 1$, i.e. v or x are not ϵ ,
- ▶ $\forall i \geq 0. uv^iwx^iy \in L$.

Proof of the Pumping Lemma for CFG

A formal proof can be found in the Additional Material

Example 1

Lemma

The language $L = \{a^i b^j c^i \mid i \geq 0\}$ is not context free.

Proof (Example 1)

- ▶ Assume L is context free.
- ▶ Let k be the constant from the pumping lemma.
- ▶ Let $z := a^k b^k c^k \in L$.
- ▶ By the pumping lemma, $z = uvwxy$ s.t. $|vwx| \leq k$, $|vx| \geq 1$ and $\forall i \geq 0. uv^i wx^i y \in L$.
- ▶ If v contains a 's and b 's or b 's and c 's, uv^2wx^2y is not an element of $a^*b^*a^*b^*$ (i.e. the language defined by this regular expression), since there is an a after a b or a b after a c .
- ▶ Therefore v is part of a^k , b^k or c^k , similarly for x .
- ▶ But now $uv^2wx^2y = a^{k+i}b^{k+j}c^{k+l}$ where at most 2 of (i, j, l) can be $\neq 0$, and at least one is $\neq 0$.
But then $a^{k+i}b^{k+j}c^{k+l} \notin L$, a **contradiction**.

Example 2

Lemma

The language $L = \{a^n b^m a^n b^m \mid n, m \geq 0\}$ is not context free.

Proof (Example 2)

- ▶ Assume L is context free.
- ▶ Let k be the constant from the pumping lemma.
- ▶ Let $z := a^k b^k a^k b^k \in L$.
- ▶ By the pumping lemma, $z = uvwxy$ s.t. $|vwx| \leq k$, $|vx| \geq 1$ and $\forall i \geq 0. uv^i wx^i y \in L$.
- ▶ If v contains both a 's and b 's $uv^2 wx^2 y$ is not an element of $a^* b^* a^* b^*$, since there are more than 3 switches between a s and b s.
- ▶ Therefore v is part of one of the subwords a^k , b^k , similarly for x .
- ▶ But now $uv^2 wx^2 y = a^{k+i} b^{k+j} a^{k+l} b^{k+m}$ where
 - ▶ at most 2 of (i, j, l, m) can be $\neq 0$,
 - ▶ if there are two they are consecutive,
 - ▶ at least one is $\neq 0$.

But then $a^{k+i} b^{k+j} a^{k+l} b^{k+m} \notin L$, a **contradiction**.

Example 3

Lemma

The language $L = \{ww \mid w \in \{a, b\}^\}$ is not context free.*

Proof (Example 3)

- ▶ We use the fact that the intersection of a context free and a regular grammar is context free.
 - ▶ This fact is not shown in this module.
 - ▶ It can be shown using the equivalence of context free languages and languages definable by Push Down Automata.
- ▶ If L were context free, so were $L' := L \cap (a^*b^*a^*b^*)$.
- ▶ But L' is just the language of Example 2, which is not context free, a **contradiction**.

Intersection of CFG

- ▶ In Example 3 we used the fact that the intersection of a context free and a regular language is context free.
- ▶ The intersection of two context free languages is in general not context free:

Consider

$$L_1 := \{a^n b^n c^m \mid n, m \in \mathbb{N}\}$$

$$L_2 := \{a^n b^m c^m \mid n, m \in \mathbb{N}\}$$

Both L_1, L_2 are context free.

However

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

which is the language of Example 1 which is **not context free**.

II.5.1. Derivation Trees for Context-Free Grammars (14.1)

II.5.2. Uniqueness of Derivation Trees (14.1)

II.5.4. The Pumping Lemma for CFG (14.4)

II.5.5. Floyd's Theorem

Repetition of words is not context free

- ▶ We have seen in Example 3 of 11.5.4 (using Pumping Lemma for Context Free Grammars), that

$$L := \{ww \mid w \in \{a, b\}^*\}$$

is **not** context free.

- ▶ Note that

$$\{ww^R \mid w \in \{a, b\}^*\}$$

is context free.

- ▶ A program language which expresses that a variable needs to be declared before contains as a sublanguage L .
 - ▶ More precisely, if we had a context free grammar for such a language, we could derive from it a context free grammar for L .
- ▶ This can be generalised to Floyd's theorem.

II.5.5. Floyd's Theorem

Theorem

Under weak assumptions a programming language, which requires that variables need to be declared before used, cannot be defined by a context free grammar.

A precise formulation and proof of Floyd's theorem can be found in "Additional Material".

II.5.5. Floyd's Theorem

- ▶ Therefore most programming languages cannot be defined by a context free grammar.
- ▶ However, one can define in most cases a context free grammar defining the basic syntax of a language.
 - ▶ Grammar allows to define a parse tree.
 - ▶ Languages which are defined by this grammar are those which can be parsed in such a way.
- ▶ Then one adds a program, which afterwards checks semantic properties of the program,
 - ▶ E.g. that a variable is declared before being used.
 - ▶ Or even more complicated features such as correctness of type checking.
- ▶ Full details can be found in the “Additional Material”.

Chapter II.6.: Push Down Automata

This Chapter will not be taught this year.