

CS_236 Language and Computation

Course Notes

Chapt. 2: Grammars for Defining Syntax (II)

Sect 2.4: Context Free Grammars and Programming Languages (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/languageComputation/09/index.html>

December 12, 2009

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

CS_236

Sect. 2.4

1/ 204

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

Derivation 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 others.
- ▶ This allows us to define derivation trees (also called parse trees).

CS_236

Sect. 2.4

2/ 204

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

Example

Consider the grammar

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

CS_236

Sect. 2.4.1

3/ 204

CS_236

Sect. 2.4.1

4/ 204

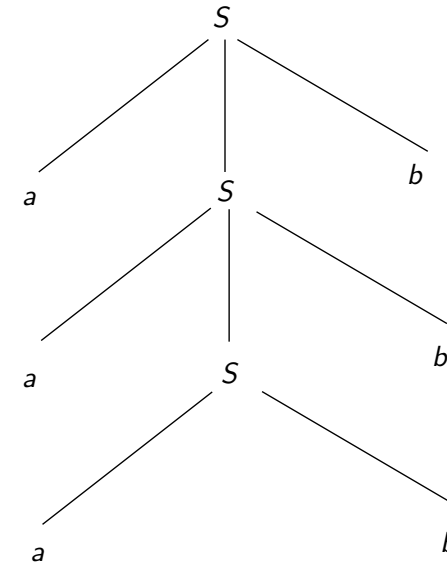
Example Derivation

We derive $aaabbbb$ in it:

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

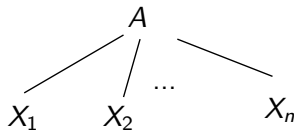
Derivation Tree

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbbb$$



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 $aaabbbb$. 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 $(T \cup N)^*$.

Notations for Derivation Trees

We introduce the following notations:

- ▶ If D is a derivation tree, then
 - ▶ $\text{label}(D)$ is the label of the root of D .
 - ▶ $\text{children}(D) = [D_1, \dots, D_n]$ means that the children (immediate subtrees of D) are D_1, \dots, D_n (read from left to right).
 - ▶ $\text{frontier}(D)$ denotes the frontier of D .
- ▶ If D_1, \dots, D_n are derivation trees, A a nonterminal, let $D := \text{tree}(A, D_1, \dots, D_n)$ be the derivation tree s.t.
 - ▶ $\text{label}(D) = A$,
 - ▶ $\text{children}(D) = [D_1, \dots, D_n]$.

Inductive Definition of Derivations

We can define the set of derivation trees as well inductively as follows:

Definition (Cont)

- ▶ If $A \in N$, D_1, \dots, D_n are derivation trees,

$A \rightarrow \text{label}(D_1).\text{label}(D_2).\dots.\text{label}(D_n)$ is a production

and $n = 1 \vee \forall i.\text{label}(D_i) \neq \epsilon$, then

- ▶ $D := \text{tree}(A, D_1, \dots, D_n)$ is a derivation tree,
- ▶ $\text{label}(D) := A$,
- ▶ $\text{children}(D) := [D_1, \dots, D_n]$,
- ▶ $\text{frontier}(D) := \text{frontier}(D_1).\text{frontier}(D_2).\dots.\text{frontier}(D_n)$.

Inductive Definition of Derivations

We can define the set of derivation trees as well inductively as follows:

Definition

We define the set of derivation trees D for a CFG $G = (T, N, S, P)$ inductively together with

- ▶ $\text{label}(D) \in T \cup N \cup \{\epsilon\}$,
- ▶ $\text{children}(D)$, a list of derivation trees,
- ▶ $\text{frontier}(D) \in (T \cup N)^*$.

as follows:

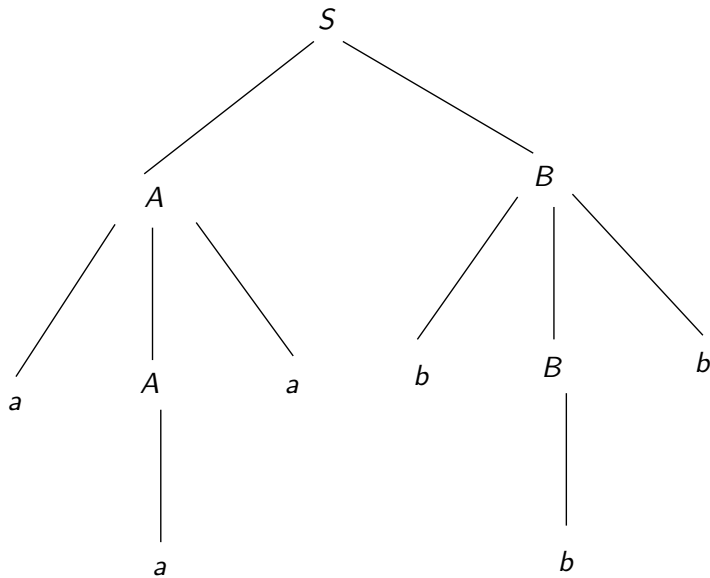
- ▶ If $A \in T \cup N \cup \{\epsilon\}$, then
 - ▶ $D := \text{tree}(A)$ is a derivation tree, (the trivial derivation tree),
 - ▶ $\text{label}(D) := A$,
 - ▶ $\text{children}(D) = []$,
 - ▶ $\text{frontier}(D) := A$.

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	$S \rightarrow AB,$ $A \rightarrow aAa, A \rightarrow a$ $B \rightarrow bBb, B \rightarrow b$

Example Derivation Tree



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. $w' = svt$.

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 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. $w' = svt$.

Left/Right-Most Derivation Sequence

Definition

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

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

Derivation Forest

For proving the equivalence of derivation trees and derivations, we need to deal with derivations $w \Rightarrow^* w'$ where w is a string. Such derivations correspond to derivation forests, as defined as follows:

Definition

Let $G = (T, N, S, P)$ be a CFG, $w, w' \in (T \cup N)^*$, $w \neq \epsilon$. Let $w = x_1, \dots, x_l$, $x_i \in T \cup N$.

A **derivation forest** with **root** w and frontier w' is a list of derivations $[D_1, \dots, D_l]$ s.t.

- ▶ $\text{label}(D_i) = x_i$,
- ▶ $\text{frontier}(D_1).\text{frontier}(D_2).\dots.\text{frontier}(D_l) = w'$.

Furthermore $\text{size}([D_1, \dots, D_l]) := \text{size}(D_1) + \dots + \text{size}(D_l)$.

Size of a Derivation Tree

We want show that for every derivation tree we can find a corresponding derivation and vice versa. For this we need a measure of the size of derivation tree.

Definition

The size $\text{size}(D)$ of a derivation tree D is defined by induction on the definition of trees:

- ▶ If $D = \text{tree}(x)$, then $\text{size}(D) := 1$.
- ▶ If $D = \text{tree}(A, D_1, \dots, D_n)$, then $\text{size}(D) := 1 + \text{size}(D_1) + \dots + \text{size}(D_n)$.

We define as well the height $\text{height}(D)$ of a derivation tree:

- ▶ If $D = \text{tree}(x)$, then $\text{height}(D) := 1$.
- ▶ If $D = \text{tree}(A, D_1, \dots, D_n)$, then $\text{height}(D) := 1 + \max\{\text{size}(D_1), \dots, \text{size}(D_n)\}$.

Lemma (Derivation Trees and Language Generation)

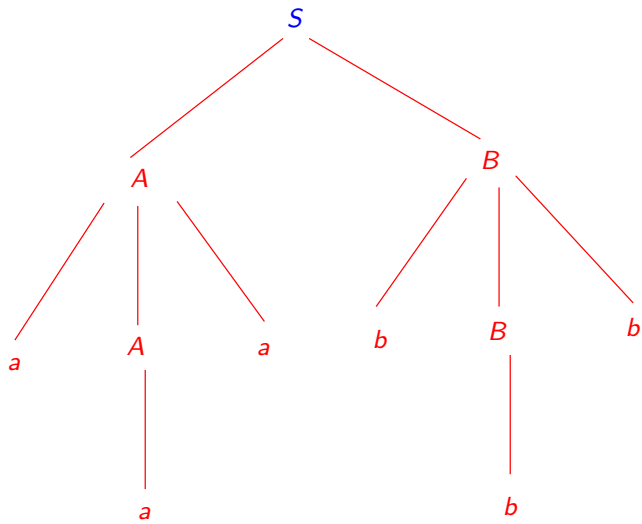
Theorem

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

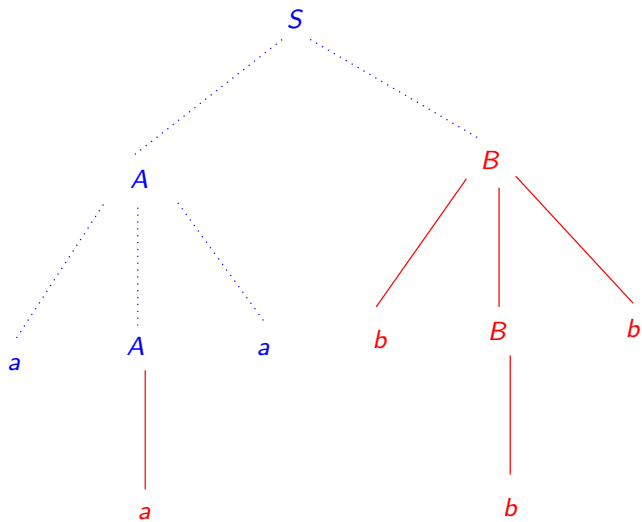
- (1) There exist a derivation forest D with root w and frontier w' .
- (2) $w \Rightarrow^* w'$.

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

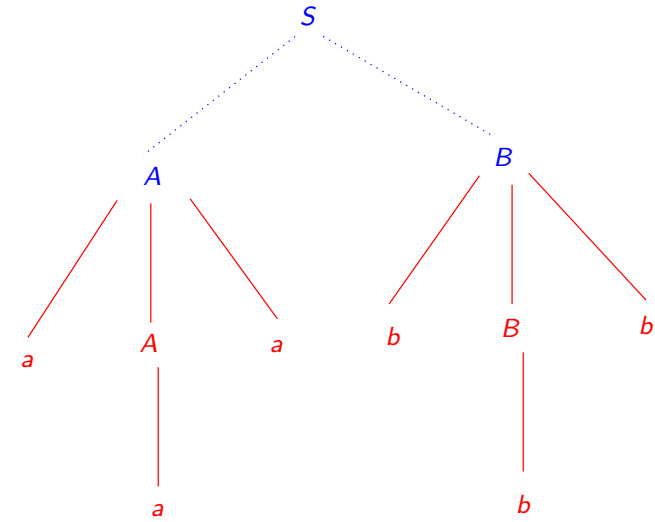
Example (Step 1)

 S 

Example (Step 3)

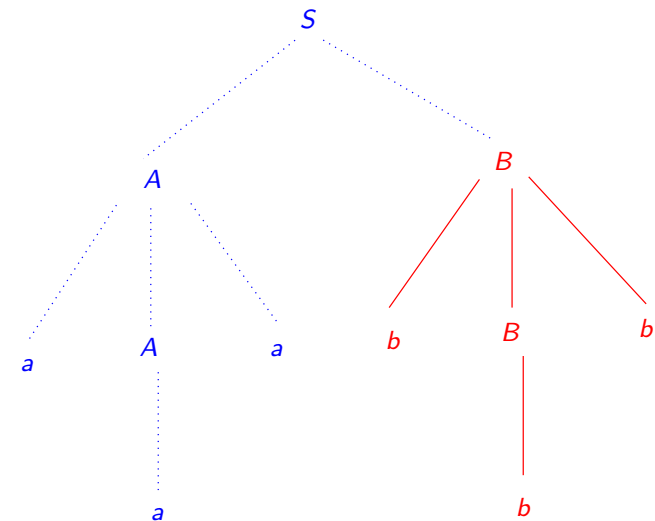
 $S \Rightarrow AB \Rightarrow aAaB$ Derivation tree for a is trivial.

Example (Step 2)

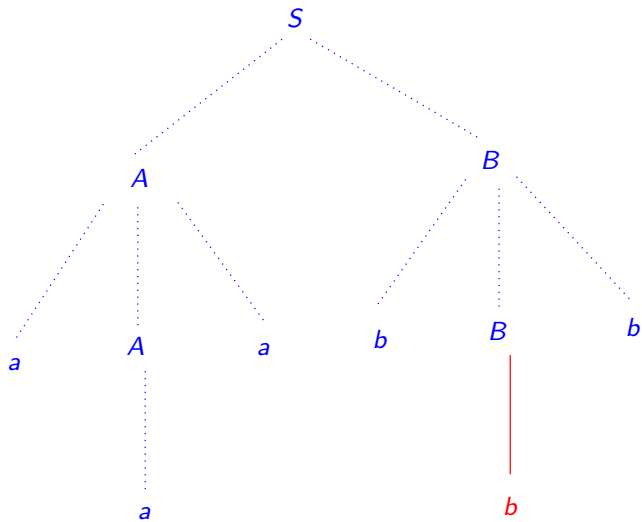
 $S \Rightarrow AB$ 

We obtain a derivation forest.

Example (Step 4)

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

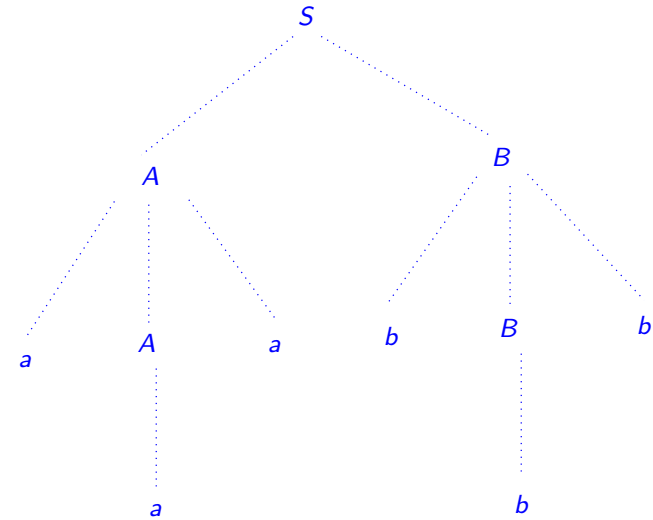
Example (Step 5)

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


Example

The above example illustrates both how to obtain from a derivation tree a derivation and from a derivation a derivation tree.

Example (Step 6)

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


Final derivation.

Proof (1) \Rightarrow (2)

- ▶ The proof is by induction on $\text{size}(D)$.
- ▶ We will do it in such a way that we get, in case all leaves are terminal symbols, a left-most derivation.
- ▶ A right most derivation can be obtained by choosing instead of the left-most the right most non-trivial derivation tree.
- ▶ Let $w = x_1 \cdots x_n$, $D = [D_1, \dots, D_n]$.
- ▶ If all D_i are trivial, then $w' = w$, $w \Rightarrow^* w'$.
- ▶ So assume at least one D_i is non-trivial. Let D_k be the left-most non-trivial derivation tree, i.e. D_1, \dots, D_{k-1} are trivial, D_k is non-trivial.
- ▶ Let $D_k = \text{tree}(A, D'_1, \dots, D'_l)$, $\text{label}(D'_i) = x'_i$. $x_k = A$.

Proof (1) \Rightarrow (2)

- ▶ Then

$$A \longrightarrow x'_1 \cdots x'_l$$

is a production, and we have that with

$$w_1 := x_1 \cdots x_{k-1} x'_1 x'_2 \cdots x'_l x_{k+1} x_{k+2} \cdots x_n$$

that

$$w = x_1 \cdots x_{k-1} A x_{k+1} \cdots x_n \Rightarrow w_1$$

is a one-step derivation.

In case $w' \in T^*$, we have $x_i \in T$, and this one-step derivation was left-most.

- ▶ We have that

$$[D_1, \dots, D_{k-1}, D'_1, \dots, D'_l, D_{k+1}, D_{k+2}, \dots, D_n]$$

is a derivation forest with root w_1 and frontier w' , and has size $\text{size}(D) - 1$.

(2) \Rightarrow (1)

- ▶ Proof is by induction on the length of the derivation $w \Rightarrow^* w'$.

- ▶ Let $w = x_1, \dots, x_n$.

- ▶ In case the length is 0, $w' = w$, and we can choose

$$D = [\text{tree}(x_1), \dots, \text{tree}(x_n)].$$

- ▶ Otherwise, assume that $x_k = A$ is a non-terminal, $A \longrightarrow y_1 \cdots y_l$ (with $y_i \in T \cup N$ or $l = 1 \wedge y_1 = \epsilon$). Let

$$w_1 := x_1 \cdots x_{k-1} A x_{k+1} \cdots x_n \Rightarrow x_1 \cdots x_{k-1} y_1 y_2 \cdots y_l x_{k+1} \cdots x_n$$

- ▶ Assume that the derivation is

$$w = x_1 \cdots x_{k-1} A x_{k+1} \cdots x_n \Rightarrow w_1 \Rightarrow^* w'$$

- ▶ By IH there exist a derivation forest

$$[D_1, \dots, D_{k-1}, D'_1, \dots, D'_l, D_{k+1}, \dots, D_n]$$

with root w_1 and frontier w' .

Proof (1) \Rightarrow (2)

- ▶ By IH there exist a derivation $w_1 \Rightarrow^* w'$, which in case of $w' \in T^*$ can be chosen as a left-most derivation sequence.
- ▶ Therefore $w \Rightarrow w_1 \Rightarrow^* w'$ is a derivation, which in case of $w' \in T^*$ can be chosen as a left-most derivation sequence.

(2) \Rightarrow (1)

- ▶ Now

$$[D_1, \dots, D_{k-1}, (A, D'_1, \dots, D'_l), D_{k+1}, \dots, D_n]$$

is a forest with root w and frontier w' .

Uniqueness of Derivation (Trees)

Lemma

Let $G = (T, N, S, P)$ be a CFG, $w \in (T \cup N)^+$, $w' \in T^*$.

- (1) Assume there are two different derivation forests with root w and frontier w' . Then there exist two different left-most and two different right-most derivations of $w \Rightarrow^* w'$.
- (2) Assume there are two different left-most derivations or two different right-most-derivations of $w \Rightarrow^* w'$. Then there exist two different derivation forests of with root w and frontier w' .

Proof of the Lemma (1)

- ▶ We prove only that there exist two different left-most derivations.
- ▶ Let $[D_1, \dots, D_n]$ and $[D'_1, \dots, D'_n]$ be two different derivation forests with root w and frontier w' .
- ▶ Induction on the length of the first derivation forest.
- ▶ If all D_i are trivial, then $w \in T^*$, $w = w'$, but then $D'_i = D_i$, which is not possible.
- ▶ Let D_k be the first non-trivial derivation tree, i.e. D_1, \dots, D_{k-1} are trivial.
- ▶ Let $D_k = \text{tree}(A, D''_1, \dots, D''_l)$, $\text{label}(D''_i) = y_i$, $A \longrightarrow y_1 \cdots y_l$ a production.
- ▶ Let $w_1 := x_1 \cdots x_{k-1} y_1 \cdots y_l x_{k+1} \cdots x_n$.
- ▶ We have D'_i are trivial for $i < k$, D_k must be non-trivial (since it has as label a nonterminal).

Uniqueness of Derivation (Trees)

A direct consequence of the lemma is the following:

Theorem

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

- (1) For every $w \in T^*$ there exist at most one derivation tree with label S and frontier w .
- (2) For every $w \in T^*$ there exist at most one left-most derivation sequence $S \Rightarrow^* w$.
- (3) For every $w \in T^*$ there exist at most one right-most derivation sequence $S \Rightarrow^* w$.

Proof of the Lemma (1)

- ▶ Case 1: D'_k has the same production at the root, i.e. $D'_k = \text{tree}(A, D'''_1, \dots, D'''_l)$, $\text{label}(D'''_i) = y_i$.
 - ▶ Then

$$[D_1, \dots, D_{k-1}, D''_1, \dots, D''_l, D_{k+1}, \dots, D_n]$$
 and

$$[D'_1, \dots, D'_{k-1}, D'''_1, \dots, D'''_l, D'_{k+1}, \dots, D'_n]$$
 must be different.
 - ▶ By IH there exist two different left-most productions $w_1 \Rightarrow w'$.
 - ▶ Therefore we obtain two different left-most productions $w \Rightarrow w_1 \Rightarrow w'$.

Proof of the Lemma (1)

- ▶ Case 2: D'_k has a different production at the root, i.e.
 $D'_k = \text{tree}(A, D''_1, \dots, D''_m)$, $\text{label}(D''_i) = y'_i$. $y'_1 \cdots y'_m \neq y_1 \cdots y_m$.
 - ▶ But then the first steps in the derivations constructed in the lemma are different, and we obtain two different left-most derivations.

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 right-most derivation).

Proof of the Lemma (2)

This proof is similar, at the first place where the two derivations differ we construct two different derivation forests.

Example 1

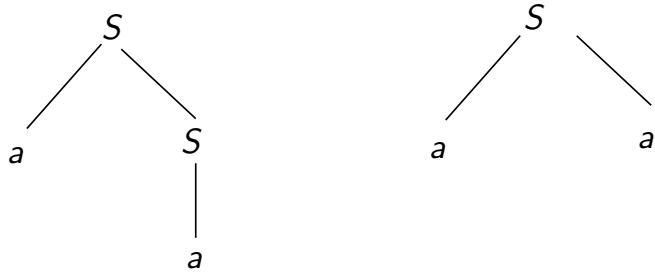
grammar	G
terminals	S
nonterminals	a, b
start symbol	S
productions	$S \longrightarrow aS$ $S \longrightarrow b$ $S \longrightarrow ab$

Example 1

There are left-most derivations of ab :

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

And two derivation trees:



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

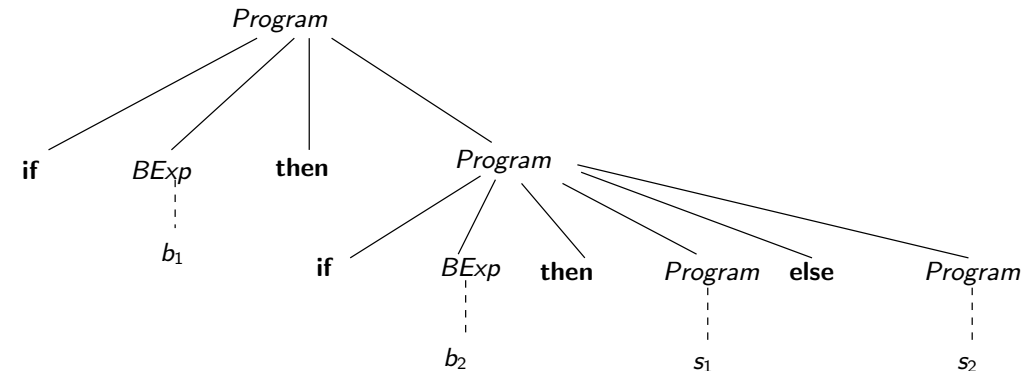
has two derivation trees (we omit the derivation trees for b_i, s_i .)

Example 2: Dangling Else

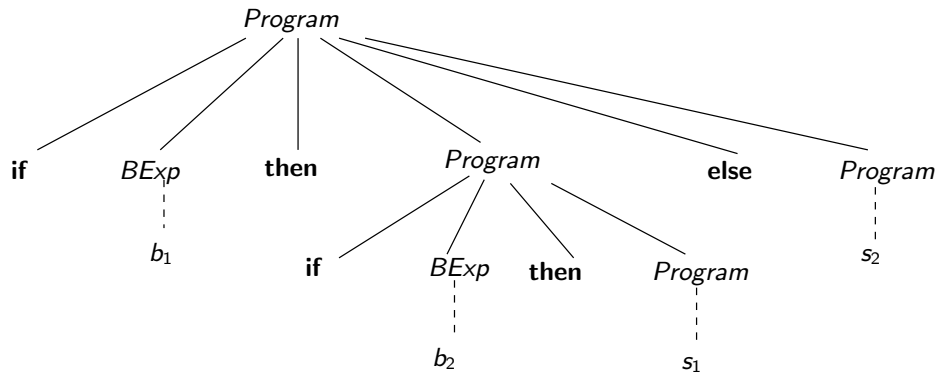
Assume the following grammar which is a cut down version of the grammar G^{while} introduced in 2.1.3 with **if_then_else_fi** replaced by **if_then** and a **if_then_else_**):

grammar	$G_{Dangling_if}$
import	$G_{Arithmetic_Expression}, G_{Boolean_Expression}$
terminals	if, then, else, :=, ;
nonterminals	$Program$
start symbol	$Program$
productions	$Program \rightarrow Id := AExp$ $Program \rightarrow \text{if } BExp \text{ then } Program \text{ else } Program$ $Program \rightarrow \text{if } BExp \text{ then } Program$

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 indentation be written as follows:

```

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

```

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 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 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 **end – if**), labelling the end of the statement.

grammar	$G_{Unambiguous_if}$
import	$G_{Arithmetic_Expression}, G_{Boolean_Expression}$
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

if b_1 **then** *if* b_2 **then** s_1 **else** s_2 *fi fi*

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

if b_1 **then** *if* b_2 **then** s_1 *fi* **else** s_2 *fi*

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 one we modify the grammar, 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

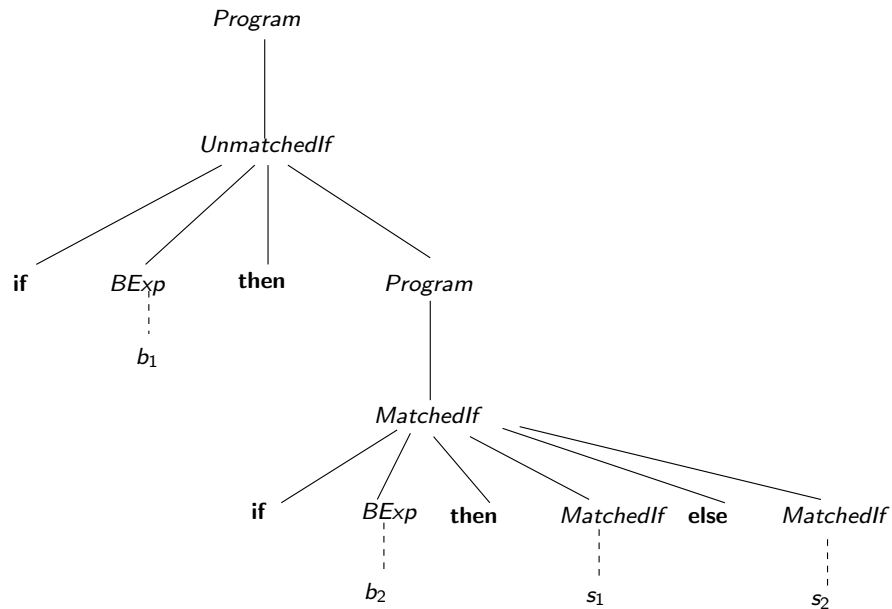
- ▶ For this we split **if_then_(else_)** into two groups:
 - ▶ MatchedIf. They match **if_then_else_**, and both the if-clause and else-clause can only be instantiated by a MatchedIf.
 - ▶ UnmatchedIf. They match **if_then_**, and an **if_then_else_** with a matched if and an unmatched else clause.
- ▶ Then an else statement will always belong to the if closest to it without yet an associated else, so the expression as above will be parsed as the first derivation tree.

Solution 2

Here is the grammar. We omit sequencing (combining programs using “;”), since it would result in an ambiguous grammar, which needs to be resolved.

grammar	$G_{Dangling_If}$
import	$G_{Arithmetic_Expression}, G_{Boolean_Expression}$
terminals	if, then, else, :=, ;
nonterminals	<i>Program</i>
start symbol	<i>Program</i>
productions	$Program \rightarrow UnmatchedIf$ $Program \rightarrow MatchedIf$ $MatchedIf \rightarrow Id := AExp$ $MatchedIf \rightarrow \mathbf{if} BExp \mathbf{then} MatchedIf \mathbf{else} MatchedIf$ $UnmatchedIf \rightarrow \mathbf{if} BExp \mathbf{then} Program$ $UnmatchedIf \rightarrow \mathbf{if} BExp \mathbf{then} MatchedIf \mathbf{else} UnmatchedIf$

Unique Derivation Tree 2nd Solution



Chomsky Normal Form

Definition

A CFG $G = (T, N, S, P)$ is in Chomsky Normal Form if all of its productions are

either of the form $A \rightarrow BC$ or of the form $A \rightarrow a$

where $A, B, C \in N$ and $a \in T$.

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

Chomsky Normal Form

Remark

If G is a CFG in Chomsky Normal Form, then $\epsilon \notin L(G)$.

Proof: Let $G = (T, N, S, P)$.

Since G has no production $A \rightarrow \epsilon$, we have that if $w \Rightarrow w'$ then the $|w'| \geq |w|$.

Therefore if $w \Rightarrow^* w'$ then $|w'| \geq |w|$.

Therefore, if $w \in L(G)$ then $S \Rightarrow^* w$ therefore $|w| \geq |S| = 1$, $w \neq \epsilon$.

Chomsky Normal Form

We are going to prove the following:

Theorem

For any CFG G there exist a CFG G' in Chomsky Normal Form s.t.

$$L(G') = L(G) \setminus \{\epsilon\}$$

Proof

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

We first define the set of nullable nonterminals of G .

A nonterminal A is nullable if $A \rightarrow \epsilon$.

They can be defined as follows:

- ▶ If $A \rightarrow \epsilon$, then A is nullable.
- ▶ If $A \rightarrow B_1 \cdots B_k$ for nullable nonterminals B_1, \dots, B_k , then A is nullable.

Step 1: Removal of null productions

Definition

A null-production of a grammar $G = (T, N, S, P)$ is a production of the form $A \rightarrow \epsilon$.

Lemma

If G is a CFG. Then there exist a CFG G' which doesn't have any null productions, and s.t. $L(G') = L(G) \setminus \{\epsilon\}$.

Obtaining G' from G

We obtain the grammar $G' = (T, N, S, P')$ from $G = (T, N, S, P)$, by defining P' from P as follows:

- ▶ We start by taking all productions of P .
- ▶ We remove all productions $A \rightarrow \epsilon$ from P .
- ▶ If $A \rightarrow w$ is a production in P , and w' is obtained by omitting some but not all occurrences of nullable nonterminals from w , then

$$A \rightarrow w'$$

is added to P .

Equivalence of G and G'

- ▶ We show that $L(G') = L(G) \setminus \{\epsilon\}$.
- ▶ First $L(G') \subseteq L(G)$, because if $A \rightarrow w'$ is a new production added to P' , then $A \Rightarrow_G^* w'$, and therefore from any derivation in G' we obtain a derivation in G .

Equivalence of G and G'

- ▶ Therefore $D' := \text{tree}(A, D'_1, \dots, D'_j)$ is a derivation tree of G' and $\text{frontier}(D) = \text{frontier}(D')$.

- ▶ Now

$$\begin{aligned} L(G) \setminus \{\epsilon\} &= \{w \in T^* \setminus \epsilon \mid S \Rightarrow_G w\} \\ &\subseteq \{w \in T^* \mid S \Rightarrow_{G'} w\} \\ &= L(G') \end{aligned}$$

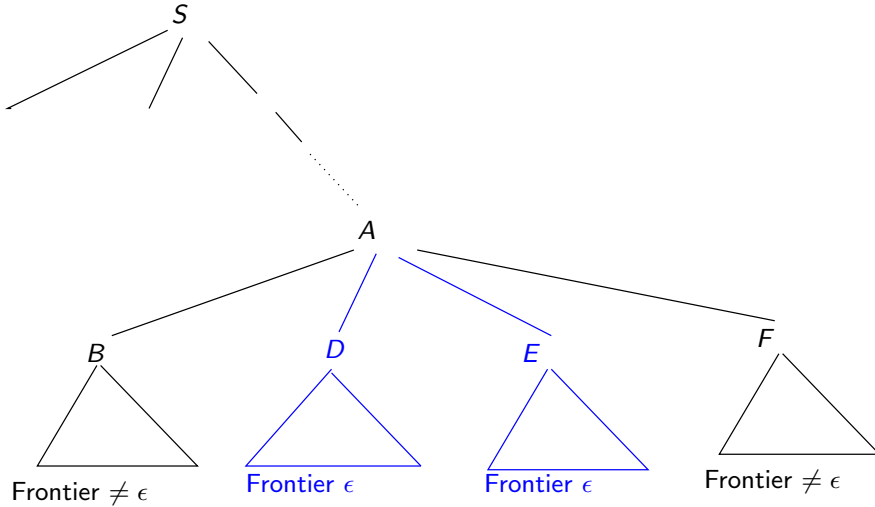
Equivalence of G and G'

- ▶ $L(G) \setminus \{\epsilon\} \subseteq L(G')$:
 - ▶ We show that from a derivation tree D of G with $\text{frontier}(D) \neq \epsilon$ we obtain a derivation tree D' of G' with the same frontier and the same root by induction on the derivations.
 - ▶ If D is trivial, let $D' = D$.
 - ▶ Otherwise let $D = (A, D_1, \dots, D_k)$.
 - ▶ Let $[D'_1, \dots, D'_j]$ be obtained by
 - ▶ Omitting any D_i s.t. $\text{frontier}(D_i) = \epsilon$.
 - ▶ Applying the IH to any D_i s.t. $\text{frontier}(D_i) \neq \epsilon$ in order to obtain a derivation tree in G' .
 - ▶ Since $\text{frontier}(D) \neq \epsilon$, the list $[D'_1, \dots, D'_j]$ is not empty.
 - ▶ For the D_i omitted we have that $\text{label}(D_i) \Rightarrow_G^* \epsilon$, so $\text{label}(D_i)$ was nullable.
 - ▶ Therefore $\text{label}(D'_1) \cdots \text{label}(D'_j)$ is obtained from $\text{label}(D_1) \cdots \text{label}(D_k)$ by omitting some nullable nonterminals.
 - ▶ $A \rightarrow \text{label}(D_1) \cdots \text{label}(D_k)$ was a production of G , and therefore $A \rightarrow \text{label}(D'_1) \cdots \text{label}(D'_j)$ is a production of G' .

- ▶ The modification done can be described as follows:
 - ▶ Assume a derivation tree D in G with root A and $\text{frontier} \neq \epsilon$.
 - ▶ Take any subtree with frontier ϵ , which is not contained in a larger subtree with same frontier ϵ .
 - ▶ So any largest subtree with frontier ϵ .
 - ▶ The root of subtree is nullable, since from it we derive ϵ .
 - ▶ Omit this subtree with its root.
 - ▶ The result of removing all such subtrees is a derivation tree in D' .

Picture

The blue subtrees will be removed:



Step 2: Removal of Silent Productions

Lemma

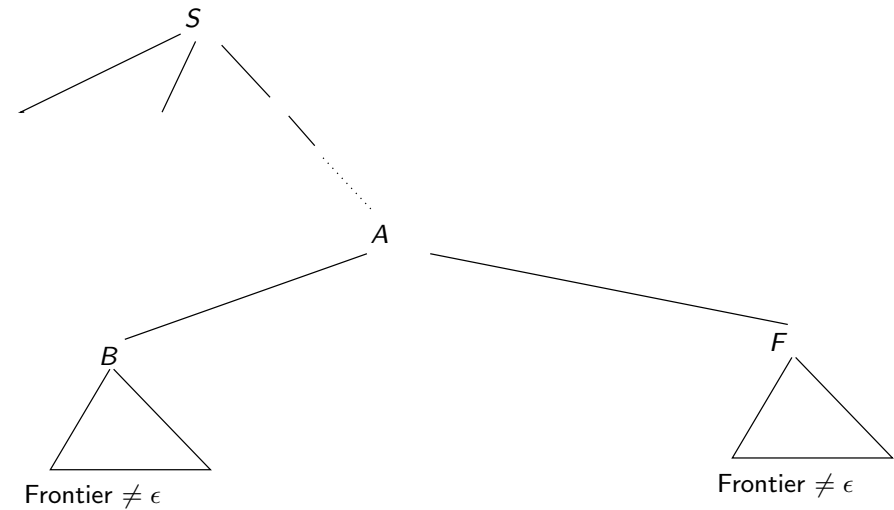
Assume G a CFG. Then there exist a CFG G' with no silent productions, i.e. no productions of the form $A \rightarrow A'$ for nonterminals A, A' s.t.

$$L(G) = L(G')$$

If G has no null productions, G' can be chosen to have no null productions as well.

Picture

After removing the subtrees:



Proof

- ▶ Let $G = (T, N, S, P)$.
- ▶ Let $G = (T, N, S, P')$ where P' consists of all productions $A \rightarrow w$ where $w \neq A$ and

$$A \Rightarrow_G^* B \rightarrow w$$

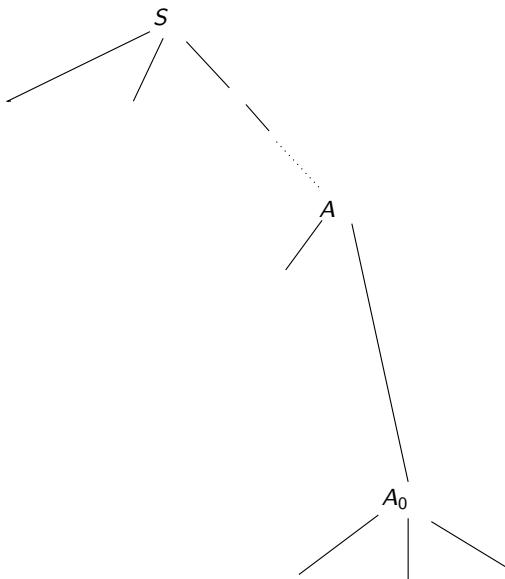
- ▶ $L(G') \subseteq L(G)$, since any derivation step $uAv \Rightarrow_{G'} uwv$ where $A \rightarrow_G w$ is as above can be replaced by $uAv \Rightarrow_G^* uBv \Rightarrow_G uwv$.

Proof of $L(G) \subseteq L(G')$

- ▶ We prove that if D is a derivation tree in G with frontier $w \in T^*$, then there exists a derivation tree D' in G' with the same root and frontier.
- ▶ Intuitively D' is obtained by contracting in D any chains of silent productions $A \rightarrow A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n$ where A_n is the head of a non-trivial derivation tree to A .

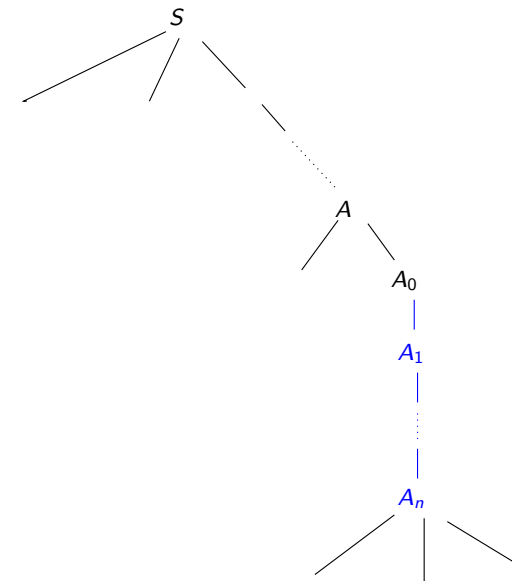
Picture

After removing the subtrees:



Picture

The blue chain of silent productions will be removed:



Removing Silent Productions

- ▶ Formally this can be done by induction on the derivation trees.
- ▶ Let D be a derivation tree with frontier w and root A .
- ▶ If $D = \text{tree}(A)$, Let $D' = D$.
- ▶ If $D = \text{tree}(A, D')$ with $\text{label}(D') \in A$, let $D' = \text{tree}(B, D_1, \dots, D_n)$ be the derivation obtained from D' by IH.

▶ Then we have

$$A \Rightarrow_G B \Rightarrow_G^* C \rightarrow \text{label}(D_1) \cdots \text{label}(D_k)$$

so

$$A \rightarrow \text{label}(D_1) \cdots \text{label}(D_k)$$

is a production of G' .

- ▶ Therefore $\text{tree}(A, D_1, \dots, D_n)$ is a derivation as desired.

Removing Silent Productions

- ▶ Otherwise $D = \text{tree}(A, D_1, \dots, D_n)$ with $n \geq 2$ or $\text{label}(D_1) \in T$.
 - ▶ Let D'_i a derivation of $\text{frontier}(D_i)$ with $\text{label}(D'_i) = \text{label}(D_i)$ obtained by the IH from D_i .
 - ▶ Then $\text{tree}(A, D'_1, \dots, D'_n)$ is a derivation as desired.

Proof

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

- ▶ Let $G' = (T, N', S, P')$ where
 - ▶ N' is obtained by adding to N for each $a \in T$ a new symbol T_a .
 - ▶ P' contains
 - ▶ Productions $T_a \rightarrow a$ for $a \in T$,
 - ▶ For any production $T \rightarrow w$ a production $T \rightarrow w'$ where w' is obtained from w by replacing each terminal a by T_a .
- ▶ It is easy to see that $L(G') = L(G)$.

Step 3

Lemma

Let $G = (T, N, S, P)$ be a CFG. Then we can obtain a grammar G' s.t.

- ▶ Productions in G' are either of the form $A \rightarrow a$ for $A \in N$ and $a \in T$, or of the form $A \rightarrow w$ where $w \in N^*$, so w consists only of nonterminals.
- ▶ $L(G') = L(G)$.
- ▶ If G has no null or silent productions, so does G' .

Proof of Chomsky Normal Form Theorem

- ▶ We prove now the Chomsky Normal Form Theorem.
- ▶ By step 1 - 3 there exists a grammar $G' = (T, N', S', P')$ s.t.
 - ▶ $L(G') = L(G) \setminus \{\epsilon\}$,
 - ▶ G' has no null or silent productions,
 - ▶ the productions of G' are either $A \rightarrow a$ or $A \rightarrow w$ with $w \in N^*$.
- ▶ Let $G'' = (T, N'', S', P'')$ where
 - ▶ P'' is obtained by replacing a production $A \rightarrow B_1 B_2 \dots B_n$ with $n \geq 3$ by

$$A \rightarrow B_1 C_1, C_1 \rightarrow B_2 C_2, \dots, C_{n-3} \rightarrow B_{n-3} C_{n-2}, \\ C_{n-2} \rightarrow B_{n-1} B_n$$

for new symbols C_1, \dots, C_{n-2} (where for each production different symbols are added).

- ▶ N'' is obtained from N by adding all new symbols C_i .

Proof of Chomsky Normal Form Theorem

- ▶ G'' is in Chomsky Normal Form.
- ▶ Obviously $L(G') \subseteq L(G'')$. since derivation steps

$$vAw \Rightarrow vB_1 \cdots B_n w$$

using production $A \rightarrow B_1 \cdots B_n$ as above can be replaced by

$$\begin{aligned} vAw &\Rightarrow vB_1 C_1 w \Rightarrow vB_1 B_2 C_2 w \Rightarrow \cdots \Rightarrow vB_1 B_2 \cdots B_{n-3} C_{n-2} w \\ &\Rightarrow vB_1 B_2 \cdots B_{n-3} B_{n-2} B_{n-1} w \end{aligned}$$

A recognition Algorithm for CFG (14.3.2)

- ▶ We present an algorithm for deciding for a CFG G in Chomsky Normal Form and a string w whether $w \in L(G)$.
- ▶ This algorithm is called the Cocke-Younger-Kasami (CYK) algorithm.

Proof of Chomsky Normal Form Theorem

- ▶ On the otherhand, let for $w \in (T \cup N'')$ \hat{w} be defined as the result of replacing occurrences of
 - ▶ C_1 by $B_2 B_3 \cdots B_n$,
 - ▶ C_2 by $B_3 B_3 \cdots B_n$,
 - ▶ etc.
 - ▶ C_{n-2} by $B_{n-1} B_n$.
- ▶ Then for any production $D \rightarrow_{G''} w$ of G'' we have either $D \rightarrow_{G'} \hat{w}$ or $D = \hat{w}$.
- ▶ Therefore, if $S' \Rightarrow_{G''}^* w$, then $S' \Rightarrow_{G'}^* \hat{w}$.
- ▶ Therefore using that for $w \in T^*$ we have $w = \hat{w}$:

$$\begin{aligned} L(G'') &= \{w \in T^* \mid S' \Rightarrow_{G''}^* w\} \\ &= \{\hat{w} \mid w \in T^* \wedge S' \Rightarrow_{G''}^* w\} \\ &\subseteq \{\hat{w} \mid w \in T^* \wedge S' \Rightarrow_{G'}^* \hat{w}\} \\ &= \{w \in T^* \mid S' \Rightarrow_{G'}^* w\} \\ &= L(G') \end{aligned}$$

Cocke-Younger-Kasami Algorithm

- ▶ Let $G = (T, N, S, P)$.
- ▶ For a word $w = t_1 \cdots t_n$ let for $1 \leq i \leq j \leq n$ $w_{i,k}$ be the subword starting from t_i of length k , i.e.

$$w_{i,k} = t_i t_{i+1} \cdots t_{i+k-1}$$

- ▶ We decide more generally for any $1 \leq i \leq n$, $1 \leq k \leq n - i + 1$ and $A \in N$ whether $T \Rightarrow^* w_{i,j}$.

- ▶ Let

$$N_{i,j} := \{A \in N \mid A \Rightarrow^* w_{i,j}\}$$

Cocke-Younger-Kasami Algorithm

► We have

- $N_{i,1} = \{A \in N \mid A \rightarrow w_{i,1}\}$, since a derivation $A \Rightarrow w_{i,1}$ cannot start with $A \Rightarrow BC$ since otherwise we would have $1 = |w_{i,1}| \geq |BC| = 2$.
- We have

$$N_{i,j+1} = \{A \mid \exists A \rightarrow BC \in P. \exists k < j+1. 1 \leq k \wedge B \in N_{i,k} \wedge C \in N_{i+k,j+1-k}\}$$

A derivation tree of $w_{i,j}$ must start with $A \rightarrow BC$. Then B, C derive two subwords of $w_{i,j}$ which together form $w_{i,j}$, both of which are non-empty, so $B \in N_{i,k}, C \in N_{i+k,j+1-k}$ for some $1 \leq k \leq j$ as above.

Cocke-Younger-Kasami Algorithm

► Now we have for $w \in T^*$

$$\begin{aligned} w \in L(G) &\Leftrightarrow S \Rightarrow^* w \\ &\Leftrightarrow S \in N_{1,|w|} \end{aligned}$$

► The above algorithm runs in cubic time in $|w|$.

Cocke-Younger-Kasami Algorithm

► So we can define $N_{i,j}$ by the following algorithm:**for** $i := 1$ **to** n **do begin**

$$N_{i,1} := \{A \mid A \rightarrow w_{i,1} \in P;\}$$
end**for** $j := 2$ **to** n **do begin****for** $i := 1$ **to** $n+1-j$ **do begin**

$$N_{i,j} := \emptyset;$$
for $k := 1$ **to** $j-1$ **do begin**

$$N_{i,j} := N_{i,j} \cup \{A \mid \exists B, C. A \rightarrow BC \in P \wedge B \in N_{i,k} \wedge C \in N_{i+k,j-k}\}$$
end**end****end**[2.4.1. Derivation Trees for Context-Free Grammars \(14.1\)](#)[2.4.2. Normal Forms for Context-Free Grammars \(14.2\)](#)[2.4.3. The Pumping Lemma for CFG \(14.4\)](#)[2.4.4. Limitations of Context Free Grammars](#)[2.4.5. Push Down Automata](#)[2.4.6. Equivalence of Final-State and Empty-Stack-PDAs](#)[2.4.7. Equivalence of CFG and PDA](#)

The Pumping Lemma for CFG

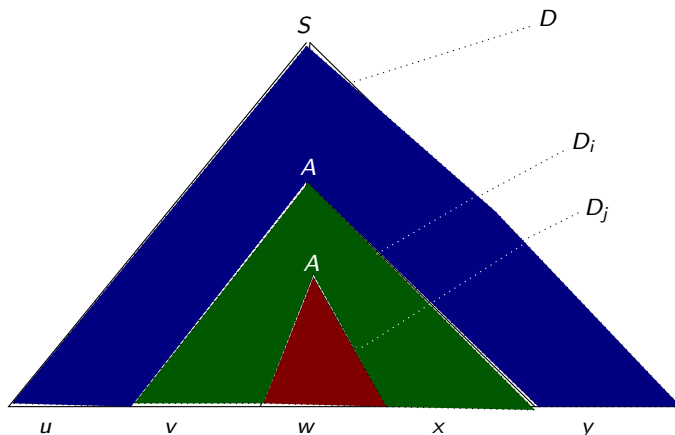
- ▶ As for regular languages we use the fact that there are only finitely many nonterminals in a CFG.
- ▶ So every sufficiently big tree in a CFG must repeat one nonterminal.
- ▶ For simplicity we consider CFGs in Chomsky-Normal-Form.

Idea of the Pumping Lemma for CFG

The idea is to find for a CFG G a constant n s.t.

- ▶ in any derivation tree of a word $w \in L(G)$ s.t. $|w| \geq n$,
- ▶ we can
 - ▶ decompose w as $uvwxy$
 - ▶ 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 $\leq n$, and $xv \neq \epsilon$.

Picture

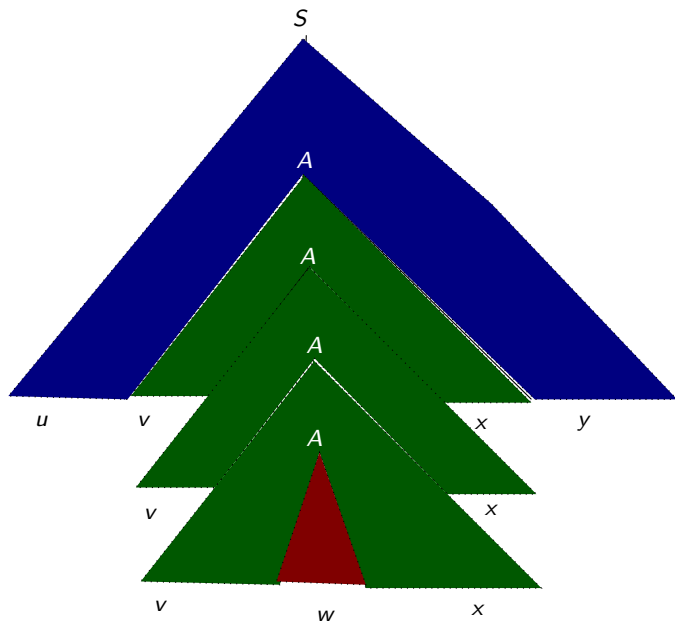


Idea of the Pumping Lemma for CFG

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

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

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

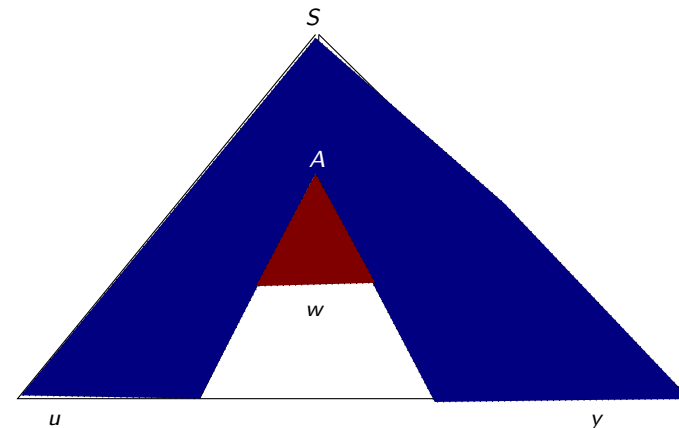
Pumping it up to uv^3wx^3y 

Pumping Lemma for CFG

Theorem

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

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

Pumping it down to uv^0wx^0y 

Lemma (Height of Derivation)

In order to prove the pumping lemma, we need to prove some lemmas.

The first one relates the height of a derivation to the length of the derived string:

Lemma

Let $G = (N, V, S, P)$ be a CFG in Chomsky Normal Form, D a derivation tree of $w \in T^*$ of height $n \geq 1$. Then $|w| \leq 2^{n-2}$.

Proof of Lemma (Height of Derivation)

Proof by induction on n .

- ▶ If $n = 2$, then $D = \text{tree}(A, \text{tree}(w))$, $w \in T$. $|w| = 1 = 2^{n-2}$.
- ▶ $2 \leq n \rightarrow n + 1$: The rule used at the root must have been $A \rightarrow BC$, since, if we had used $A \rightarrow a$, we would get a tree of height 2. Let $D = \text{tree}(A, D_1, D_2)$. Then

$$\begin{aligned} |w| &= |\text{frontier}(w_1).\text{frontier}(w_2)| \\ &= |\text{frontier}(w_1)| + |\text{frontier}(w_2)| \\ &\leq 2^{n-2} + 2^{n-2} = 2^{n+1-2} \end{aligned}$$

Existence of Subderivations of smaller heights

Lemma

Let D be a derivation of height n , $1 \leq k < n$. Then there exist a proper subderivation D' of D of height k .

Definition of Subderivations

Definition

Let $G = (T, N, S, P)$ be a grammar, D', D derivations in G . We define what it means for D' to be a subderivation of D :

- ▶ D is a subderivation of itself.
- ▶ If D'' is a child of D , and D' is a subderivation of D'' then D' is a subderivation of D .

D' is a proper subderivation of D if D' is a subderivation of D , but $D' \neq D$.

Proof of Lemma (Existence of Subderivations)

- ▶ The proof is by induction on the height n of D :
- ▶ In case $n = 1$, no $1 \leq k < n$ exists.
- ▶ In case $n = 2$, $D = \text{tree}(A, D_1, \dots, D_l)$. We must have $l \geq 1$, otherwise $\text{height}(D) = 1$. $\text{height}(D_1) = 1 = k$, let $D' = D_1$.
- ▶ Induction step $n \rightarrow n + 1$, assuming $n \geq 2$.
 - ▶ Let $D = \text{tree}(A, D_1, \dots, D_l)$.
 - ▶ $n + 1 = \text{height}(D) = \max\{\text{height}(D_1), \dots, \text{height}(D_l)\} + 1$,
 - ▶ So there exists an i s.t. $\text{height}(D_i) = n$.
 - ▶ If $k = n$, choose $D' := D_i$.
 - ▶ If $k < n$, then by IH there exist a proper subderivation D'' of D_i of height k , which is a proper subderivation of D as well. $D' := D''$.

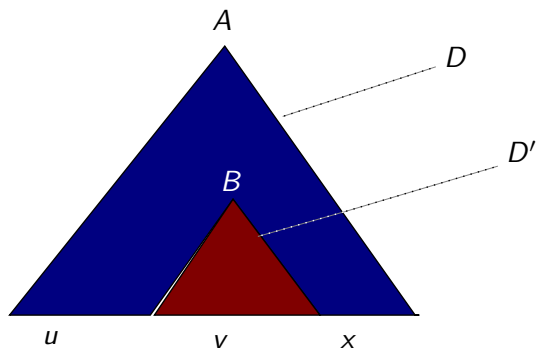
Derivations Deriving Non-Empty Strings

Lemma

Assume $G = (T, N, S, P)$ is a CFG with no null-productions. Then if $A \Rightarrow^* w$, then $|w| \geq 1$.

Proof: If $v \Rightarrow v'$, then $|v'| \geq |v|$. Therefore if $v \Rightarrow^* v'$, then $|v'| \geq |v|$. Now $|w| \geq |A| = 1$.

Picture



Cutting out Subderivations

Lemma

Let G be a CFG.

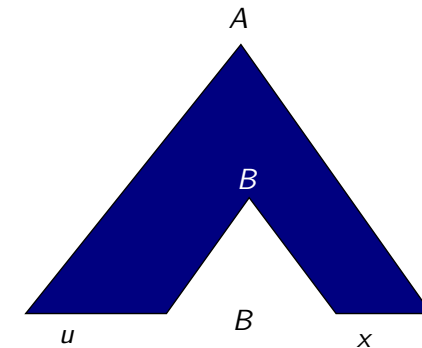
Let D be a derivation tree with label A and frontier w . Let D' be a proper subderivation D' of D with label B and frontier v .

Then there exist u, x s.t. $w = uvx$, and a derivation $A \Rightarrow^* uBx$.

Furthermore, if $\text{height}(D) \geq 2$ and D has no null or silent productions, then $|u| + |x| \geq 1$.

Picture

After cutting out the derivation:



Proof (Cutting out Subderivations)

Induction on D' being a subderivation of D .

We write “the special case” for the condition

“height(D) ≥ 2 and D has no null or silent productions”.

Let $D = \text{tree}(A, D_1, \dots, D_n)$.

In the special case we get $n \geq 2$.

- (If $n = 1$, we would obtain $D_1 = \text{tree}(a, D'_1, \dots, D'_l)$, a must be a nonterminal, since G has no silent productions, and therefore $l = 0$. But then height(D) = 1).

Proof (Cutting out Subderivations)

Case 1: D' is a proper subderivation of a child D_i of D . Let C be the label of D_i .

Let $w' := \text{frontier}(D_i)$. By IH there exists u', x' s.t. $w' = u'vx'$, and a derivation tree D' with label C and frontier $u'Bx'$.

$$\begin{aligned} u'' &:= \text{frontier}(D_1).\text{frontier}(D_2).\dots.\text{frontier}(D_{i-1}) \\ x'' &:= \text{frontier}(D_{i+1}).\text{frontier}(D_{i+2}).\dots.\text{frontier}(D_n) . \end{aligned}$$

Then $w = u''w'x'' = (u''u')v(x'x'')$. Let $u := u''u'$, $x := x'x''$. Let

$$D'' := \text{tree}(A, D_1, \dots, D_{i-1}, D', D_{i+1}, \dots, D_n) .$$

D'' is a derivation tree of $u''u'Bx'x'' = uBx$.

Furthermore, in the special case we have as before $u'' \neq \epsilon$ or $x'' \neq \epsilon$, therefore $u \neq \epsilon$ or $x \neq \epsilon$.

Proof (Cutting out Subderivations)

Case 1: D' is a child of D .

Then $D = \text{tree}(A, D_1, \dots, D_n)$, $D' = D_i$ some i .

Let

$$\begin{aligned} u &:= \text{frontier}(D_1).\text{frontier}(D_2).\dots.\text{frontier}(D_{i-1}) \\ x &:= \text{frontier}(D_{i+1}).\text{frontier}(D_{i+2}).\dots.\text{frontier}(D_n) . \end{aligned}$$

Then $w = uvx$. Let

$$D'' := \text{tree}(A, D_1, \dots, D_{i-1}, \text{tree}(B), D_{i+1}, \dots, D_n) .$$

D'' is a derivation tree of uBx with the label A .

Furthermore, in the special case we have $n \geq 2$, $\text{frontier}(D_i) \neq \epsilon$, therefore $u \neq \epsilon$ or $x \neq \epsilon$.

Chains of Derivations

Definition

Let G be a CFG. A **chain of derivations** is a list of derivations $[D_1, \dots, D_n]$ s.t. D_{i+1} is a child of D_i and height(D_n) = 1. n is called the **length** of the chain of derivations.

Existence of Chains of Derivations

Lemma

Let G be a CFG. If D has height n then there exists a chain of derivations $[D_1, \dots, D_n]$ of length n s.t. $D_1 = D$.

Proof (Existence of Chains of Derivations)

Induction on n : If $n = 1$, then $[D]$ is a chain of derivations as desired.
 $n \rightarrow n + 1$: Let D have height $n + 1$.

There exist a child D_2 of $D_1 := D$ which has height n .

By IH there exist a chain of derivations $[D_2, D_3, \dots, D_{n+1}]$ of length n starting with D_2 .

Then $[D_1, \dots, D_{n+1}]$ is a chain of derivations as desired.

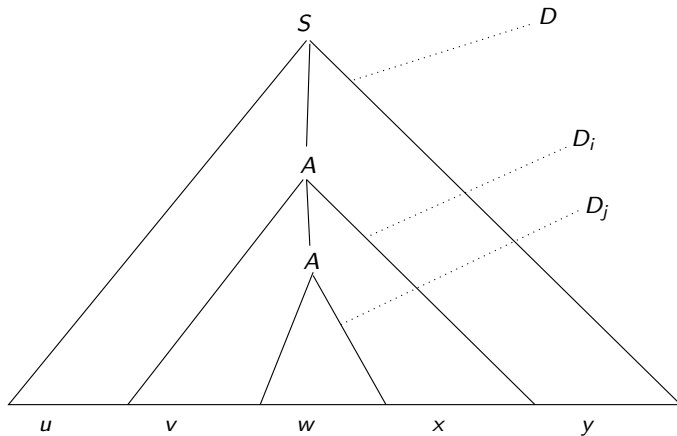
Proof of the Pumping Lemma

- ▶ Let $L = L(G)$. W.l.o.g. G is in Chomsky Normal Form.
 - ▶ There exist a grammar G' in Chomsky Normal Form s.t. $L(G') = L \setminus \{\epsilon\}$.
 - ▶ If the lemma holds for $L(G')$, then it holds for L as well (with the same constant n).
- ▶ Let $G = (T, N, S, P)$, and assume $|N| = m$.
- ▶ Let $n := 2^m$.

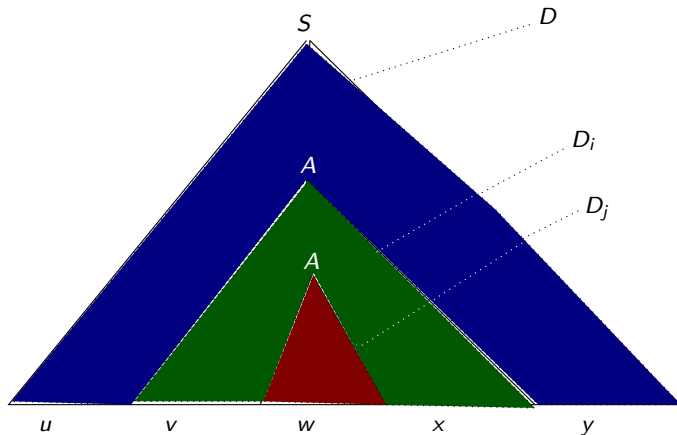
Proof of the Pumping Lemma

- ▶ Assume $z \in L(G)$, $|z| \geq n = 2^m$.
- ▶ Let D be a derivation tree for z .
- ▶ $\text{height}(D) \geq m + 2$.
- ▶ There exist a subderivation D' of D of height $m + 2$.
- ▶ Then there exist a chain of derivations $[D_1, D_2, \dots, D_{m+2}]$, s.t. $D' = D_1$.
- ▶ $\text{label}(D_i) \in N$ for $i = 1, \dots, m + 1$.
- ▶ Therefore there exists $1 \leq i < j \leq m + 1$ s.t. $\text{label}(D_i) = \text{label}(D_j) =: A$.
- ▶ So we have found a subderivation D_i of D of height $\leq m + 2$ which contains a proper subderivation D_j with the same label.

Picture



Picture



Proof of the Pumping Lemma

- ▶ Let $w = \text{frontier}(D_j)$.
- ▶ By the “Cutting out of Derivations” lemma applied to D_i and D_j , we have $\text{frontier}(D_i) = vwx$ s.t. $vx \neq \epsilon$, and $A \Rightarrow^* vAx$. (The green derivation on the next picture).
- ▶ By the “Cutting out of Derivations” lemma applied to D and D_i we have $z = uvwxy$ for some strings u, y , and we have $S \Rightarrow^* uAy$. (The blue derivation on the next picture).
- ▶ Furthermore by D_j we have $A \Rightarrow^* w$. (The red derivation on the next picture).
- ▶ $\text{height}(D_i) \leq m + 2$, therefore $|vwx| \leq 2^m = n$.
- ▶ We obtain

$$A \Rightarrow^* vAx \Rightarrow^* v^2Ax^2 \Rightarrow^* \dots$$

$$\forall i \geq 0. A \Rightarrow^* v^iAx^i$$

$$\forall i \geq 0. S \Rightarrow^* uAy \Rightarrow^* uv^iAx^i y \Rightarrow^* uv^iwx^i y$$

$$\forall i \geq 0. uv^iwx^i y \in L(G)$$

Example 1

Lemma

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

Proof (Example)

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

Floyd's Theorem

- ▶ We are going to show Floyd's Theorem, that a programming language with variable declarations cannot be defined in a context free way.
- ▶ We need some restrictions on what is meant by a programming language.
- ▶ In order to illustrate it, we consider a restricted form of the while language already considered.

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

Grammar for While Programs

Consider the grammar studied in 2.1.3:

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

Modified Grammar

We simplify it by adding variable declarations, and omit **if_then_else** and **while** and disambiguate the sequencing construct. We obtain the following grammar:

Grammar for Programs With Declarations

grammar	$G^{Programs_with_declarations}$
import	$G^{Arithmetic_Expression}, G^{Declarations}$
terminals	skip, begin, end, :=, ;
nonterminals	<i>Program, CommandList, Command</i>
start symbol	<i>Program</i>
productions	$Program \longrightarrow \mathbf{begin} \textit{Declaration} ; \textit{CommandList} \mathbf{end}$ $CommandList \longrightarrow \textit{Command}$ $CommandList \longrightarrow \textit{CommandList} ; \textit{Command}$ $Command \longrightarrow \mathbf{skip}$ $Command \longrightarrow \textit{Id} := \textit{AExp}$

Declarations

grammar	$G^{Declarations}$
import	$G^{Identifier}$
terminals	nat
nonterminals	<i>Declaration, IdentifierList</i>
start symbol	<i>Declaration</i>
productions	$Declaration \longrightarrow \mathbf{nat} \textit{IdentifierList}$ $IdentifierList \longrightarrow \textit{Identifier}$ $IdentifierList \longrightarrow \textit{Identifier} , \textit{IdentifierList}$

Floyd's Theorem

Theorem

Let

$$Decl_{n,m} := \mathbf{begin} \textit{nat} \ a^n b^m ; \ a^n b^m := 0 \ \mathbf{end}$$

Let

$$Decl_{n,m} \subseteq L$$

be a programming language.

Assume some sanity condition on what it means for L to be a programming language, including that in L all identifiers need to be declared before being used.

Then L is not context free.

Floyd's Theorem

Theorem (Cont)

The sanity conditions are the following:

1. In any program of L all identifiers of the program are declared by strings t where t is a type identifier, which are special elements of the alphabet.
2. **nat** is one type identifier.
3. The keywords are special elements of the alphabet.
4. Identifiers are strings.
5. **begin** and **end** need to be balanced in L .
6. For an identifier s $s := 0$ is an instruction using identifier s .
7. Removing of one or more of the symbols $;$, $=$, 0 from $s := 0$ yields a string which is not an instruction.
8. Programs in L contain at least one instruction.

Proof (Floyd's Theorem)

- ▶ Assume L is context free and let k be the constant of the pumping lemma for CFG.
- ▶ Let $n, m > k$.
- ▶ $Decl_{n,m} \in L$ and $|Decl_{n,m}| > k$.
- ▶ By the pumping lemma there exists u, v, w, x, y s.t.

$$Decl_{n,m} = uvwxy \wedge vx \neq \epsilon \wedge |vwx| \leq k \wedge \forall i \geq 0. uv^i wx^i y \in L$$

- ▶ Let $P_i := uv^i wx^i y$.
- ▶ So

$$uvwxy = Decl_{n,m} = \mathbf{begin\ nat\ } a^n b^m ; a^n b^m := 0 \mathbf{end}$$

Proof (Floyd's Theorem)

$$uvwxy = Decl_{n,n} = \mathbf{begin\ nat\ } a^n b^m ; a^n b^m := 0 \mathbf{end}$$

$$P_i := uv^i wx^i y \in L$$

- ▶ Because $|vwx| \leq k$, vwx cannot containing both **begin** and **end**.
- ▶ If v or x contained **begin**, or **end**, then P_0 would contain only one of these two keywords, contradicting the fact that **begin** and **end** need to be balanced.
- ▶ If v contains **nat**, then vwx must be part of **nat** $a^n b^n$.
Then P_0 wouldn't contain a declaration of $a^n b^m$ which is used in the statement $a^n b^m := 0$ which is part of P_0 .
- ▶ Therefore we have that vwx must be part of $a^n b^m ; a^n b^m := 0$.

Proof (Floyd's Theorem)

$$uvwxy = Decl_{n,n} = \mathbf{begin\ nat\ } a^n b^m ; a^n b^m := 0 \mathbf{end}$$

$$P_i := uv^i wx^i y \in L$$

$$vwx \text{ part of } a^n b^m ; a^n b^m := 0$$

- ▶ If $:= 0$ would overlap with x , then P_0 would not have one or more of those symbols, and the part after ; wouldn't be an instruction.
- ▶ So vwx is part of $a^n b^m ; a^n b^m$.

Proof (Floyd's Theorem)

$uvwxy = Decl_{n,n} = \text{begin nat } a^n b^m ; a^n b^m := 0 \text{ end}$

$P_i := uv^i wx^i y \in L$

vwx part of $a^n b^m$; $a^n b^m$

- ▶ If v or x contain $;$, then P_0 doesn't contain a $;$, so it consists only of a declaration and has no instruction.
- ▶ If w contains $;$, then v is contained in b^m , x contained in a^n , therefore in P_0 the declared identifier and used identifier are different, contradicting that identifiers need to be declared.
- ▶ So vwx is part of $a^n b^m$ before or after the $;$. But then in P_0 the identifiers before and after $;$ are different, contradicting that identifiers need to be declared.

Motivation

- ▶ There are context free languages which are not regular, so they can't be recognised by a NFA.
- ▶ The problem is that when parsing a language such as $L = \{ww^R \mid w \in \{a, b\}^*\}$, when checking the second half of the word, one needs to know the first half of the word in reverse order.
- ▶ An NFA has only finitely many states, and therefore only finite memory.
- ▶ We can repair the problem by adding a stack to memory, which allows us to record in case of L the first half of the word.
- ▶ The resulting machine will be called a push-down automaton (PDA).

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

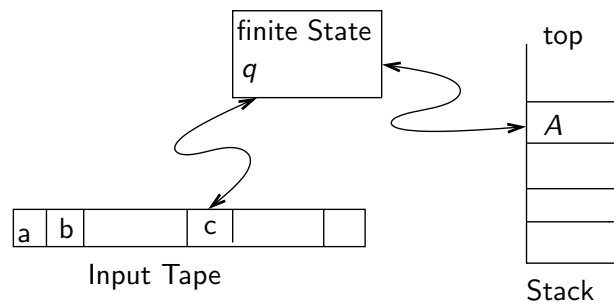
2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

Architecture of a PDA

- ▶ A PDA consists of
 - ▶ An input type, containing the input word.
 - ▶ The input tape will be read only from left to right.
 - ▶ A finite state q .
 - ▶ A stack.

Picture



Execution of a PDA

- ▶ The automaton starts with the head on the left most symbol of the input tape, with some initial state and a start symbol on the stack.
- ▶ It will then do the following non-deterministically:
 - ▶ If the stack is empty, it will get stuck.
 - ▶ Otherwise, depending on the state and the top stack symbol it will do one of the following:
 - ▶ It might read the next letter on the tape and move right, or not. If yes, the next operation depends on that letter.
 - ▶ It might get stuck because it has no transition.
 - ▶ Then it will change to a new state.
 - ▶ It will replace the top symbol of the stack by new symbols. This could be one symbol, many symbols, or none. In the latter case the top element of the stack is removed.

Execution of a PDA

- ▶ The PDA will stop if
 - ▶ it gets stuck (because of empty stack or having no operation possible) while still reading the word,
 - ▶ it wants to read a letter and has reached the end of the word.
- ▶ There are two kinds of languages defined from a PDA:
 - ▶ The language accepted by a final state.
A word is accepted by final state, if the PDA reads the complete word and reaches a state which is final (accepting).
 - ▶ The language accepted by empty stack.
A word is accepted by empty stack, if the PDA reads the complete word and then obtains empty stack.

Acceptance by Final State vs Empty State

- ▶ Clearly, acceptance by final state is natural generalisation of a NFA.
- ▶ The standard $LL(k)$, $LR(k)$, LALR parsing algorithms use a deterministic PDA accepting by final state.
- ▶ CFG correspond in a natural way to nondeterministic PDA which accept by empty stack.
- ▶ We will see that the languages accepted by PDAs by empty stack and accepted by PDAs by final state are equivalent.
- ▶ Therefore PDAs accepting by empty state correspond to intermediate machines for proving the equivalence of context free languages and languages accepted by final state by a PDA.

Acceptance by Final State vs Empty State

- ▶ Deterministic PDAs accepting by empty stack only accept languages L which have the prefix property, i.e. if w is a proper prefix of a word $w' \in L$, then $w \notin L$.
- ▶ Since parsing should be ideally done by a deterministic machine, deterministic PDAs accepting by final state form a more natural model.

Interpretation of the Components

- ▶ The input alphabet T will be the alphabet on the tape; so the PDA will look at elements of T^* and check whether it accepts them or not;
- ▶ The stack will consist of elements of the stack alphabet Γ .
- ▶ The PDA will start in start state q_0 with stack consisting of the start stack symbol Z_0 .
- ▶ F will be the set of final states in case we consider the language accepted by final state.
- ▶ $(q, Z) \xrightarrow{a} (q', w)$ means that the PDA can, when in state q , and if the next tape symbol is a move on the tape once to the right, change to state q' , and replace the top stack symbol Z by w .
- ▶ $(q, Z) \xrightarrow{\epsilon} (q', w)$ means that the PDA can, when in state q and having top symbol Z on the stack, make a move without looking at the next letter, and switch to state q' , and replace the top symbol on the stack by w .

Definition of PDA

Definition

A push down automaton (PDA) $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ consists of

- ▶ a input alphabet T ;
- ▶ a finite set of states Q ;
- ▶ a stack alphabet Γ ;
- ▶ a start state $q_0 \in Q$;
- ▶ an start stack symbol $Z_0 \in \Gamma$.
- ▶ a set of accepting or final states $F \subseteq Q$;
- ▶ a transition relation $\delta \subseteq Q \times \Gamma \times (T \cup \{\epsilon\}) \times Q \times \Gamma^*$, where we write $(q, Z) \xrightarrow{a} (q', w)$ for $(q, Z, a, q', w) \in \delta$.

Example

- ▶ We define a PDA which accepts the language

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

- ▶ We will use the stack in order to record the first half of the word, so need the stack symbols a, b .
- ▶ In addition we need the bottom symbol of the stack Z_0 . After having finished parsing the word, if we read this symbol the PDA will move to an accepting state.
- ▶ There are 3 states:
 - ▶ Initial state q_0 .
In this state the PDA will read the first half of the word, and push it onto the stack.
 - ▶ An intermediate state q_1 .
When reading the second half of the word, the PDA will be in this state and pop symbols from the stack.
 - ▶ A final accepting state.

Example

- So our PDA will be

$$(\{a, b\}, \{q_0, q_1, q_2\}, \{Z_0, a, b\}, q_0, Z_0, \{q_2\}, \delta)$$

Transitions

- The PDA will guess when it has reached the middle of the stack, and then switch silently (making a ϵ -transition, i.e. a transition without reading the next symbol) to state q_1 without modifying the stack. This can happen at the beginning (without having read a symbol, so stack is Z_0), or when it has already pushed some symbol on the stack, so the stack symbol can be any of a, b, Z_0 :

$$\begin{aligned} (q_0, Z_0) &\xrightarrow{\epsilon} (q_1, Z_0) \\ (q_0, a) &\xrightarrow{\epsilon} (q_1, a) \\ (q_0, b) &\xrightarrow{\epsilon} (q_1, b) \end{aligned}$$

Transitions

We have the following transitions in δ :

- Initially the PDA is in state q_0 , sees stack symbol Z_0 , and then reads the first symbol on the tape and pushes it on the tape:

$$\begin{aligned} (q_0, Z_0) &\xrightarrow{a} (q_0, aZ_0) \\ (q_0, Z_0) &\xrightarrow{b} (q_0, bZ_0) \end{aligned}$$

- When reading future letters, the PDA will, when in state q_0 , push them on the stack:

$$\begin{aligned} (q_0, a) &\xrightarrow{a} (q_0, aa) \\ (q_0, b) &\xrightarrow{a} (q_0, ab) \\ (q_0, a) &\xrightarrow{b} (q_0, ba) \\ (q_0, b) &\xrightarrow{b} (q_0, bb) \end{aligned}$$

Transitions

- When in state q_1 , the PDA compares whether the letters it reads are identical to those it read in the first part, so whether the letter it reads is identical to the letter on the stack. If yes it empties the stack.

$$\begin{aligned} (q_1, a) &\xrightarrow{a} (q_1, \epsilon) \\ (q_1, b) &\xrightarrow{b} (q_1, \epsilon) \end{aligned}$$

Transitions

- ▶ When the stack is emptied, while in state q_1 , the PDA can move to the accepting state q_2 . It will as well empty the stack.
If there are more letters to be read, the PDA will get stuck in that state, because there will be no transitions in this state.

$$(q_1, Z_0) \xrightarrow{\epsilon} (q_2, \epsilon)$$

Graphical Presentation

We can present a PDA as well graphically by a transition diagram with

- ▶ states, start state, final states represented as before,
- ▶ transitions labelled with an expression

$$a, Z/w$$

where $a \in T$, $Z \in \Gamma$, $w \in \Gamma^*$,

where a transition from state q to q' labelled by $a, Z/w$ stands for

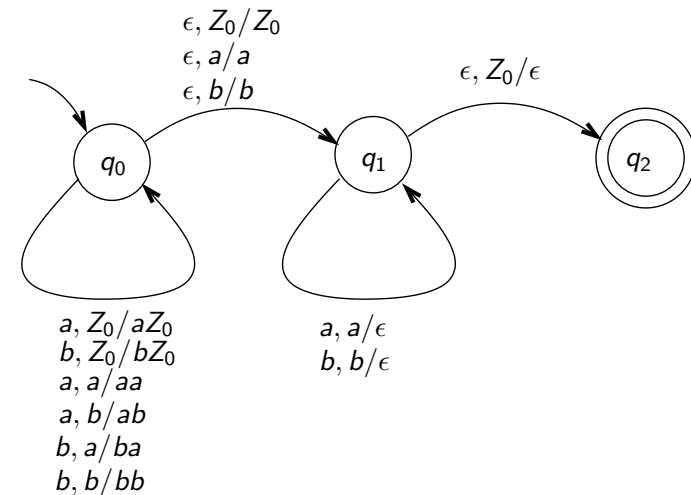
$$(q, Z) \xrightarrow{a} (q', w)$$

- ▶ The alphabet T is implicitly given by the elements of the alphabet appearing in the transitions.

Presentation of the PDA by Table

PDA	P
terminals	a, b
states	q_0, q_1, q_2
stack alphab.	Z_0, a, b
start state	q_0
start stack	Z_0
final	q_2
transitions	$(q_0, Z_0) \xrightarrow{a} (q_0, aZ_0)$ $(q_0, Z_0) \xrightarrow{b} (q_0, bZ_0)$ $(q_0, a) \xrightarrow{a} (q_0, aa)$ $(q_0, b) \xrightarrow{a} (q_0, ab)$ $(q_0, a) \xrightarrow{b} (q_0, ba)$ $(q_0, b) \xrightarrow{b} (q_0, bb)$ $(q_0, Z_0) \xrightarrow{\epsilon} (q_1, Z_0)$ $(q_0, a) \xrightarrow{\epsilon} (q_1, a)$ $(q_0, b) \xrightarrow{\epsilon} (q_1, b)$ $(q_1, a) \xrightarrow{a} (q_1, \epsilon)$ $(q_1, b) \xrightarrow{b} (q_1, \epsilon)$ $(q_1, Z_0) \xrightarrow{\epsilon} (q_2, \epsilon)$

Example in Graphical Notation



Configuration of a PDA

- ▶ The complete state of a PDA is given by its stack and an element of Q .
- ▶ We call this the configuration of a PDA.

Definition

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

A **configuration** of P is given by an element (q, z) where $q \in Q, z \in \Gamma^*$.

Transition of Configurations

- ▶ We extend the relation $(q, Z) \xrightarrow{a} (q', z)$ for $q, q' \in Q, Z \in \Gamma, z \in \Gamma^*, a \in T \cup \{\epsilon\}$ to a one step relation

$$(q, z) \xrightarrow{a^1} (q', z')$$

between configurations $(q, z), (q', z')$ and $w \in T^*$ expressing that:

- ▶ If the PDA has configuration (q, z) , then it can make a one step movement using a to configuration (q', z') .
- ▶ Furthermore we define an n -step transition relation

$$(q, z) \xrightarrow{w^n} (q', z')$$

and a transition relation expressing that we can move from one configuration to another in arbitrarily many steps

$$(q, z) \xrightarrow{w^*} (q', z')$$

One Step Transition of Configuration

Definition

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

for $q, q' \in Q, Z \in \Gamma, z \in \Gamma^*$, define the **one step transition relation** between configurations (q, z) and (q', z') and $a \in T \cup \{\epsilon\}$, written as

$(q, z) \xrightarrow{a^1} (q', z')$ by:

$$\text{If } (q, Z) \xrightarrow{a} (q', z), \text{ then } \forall z' \in Z^*(q, Zz') \xrightarrow{a^1} (q', zz')$$

Since $\xrightarrow{a^1}$ extends \xrightarrow{a} , we often write $(q, z) \xrightarrow{a} (q', z')$ instead of $(q, z) \xrightarrow{a^1} (q', z')$.

n -Step Transition of Configuration

Definition

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

We define for $n \in \mathbb{N}$ the n -step transition relation between configurations $(q, z), (q', z')$ and $w \in T^*$, written as $(q, z) \xrightarrow{w^n} (q', z')$ as follows:

$$(q, z) \xrightarrow{\epsilon^0} (q, z)$$

$$\text{If } (q, z) \xrightarrow{a^1} (q', z') \text{ and } (q', z') \xrightarrow{w^n} (q'', z''),$$

$$\text{then } (q, z) \xrightarrow{aw^{n+1}} (q'', z'')$$

Transition of Configuration

Definition

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

We define the transition relation between configurations (q, z) , (q', z') and $w \in T^*$, written as $(q, z) \xrightarrow{w} (q', z')$ as follows:

$$(q, z) \xrightarrow{w}^* (q', z') \text{ iff } \exists n \in \mathbb{N}. (q, z) \xrightarrow{w}^n (q', z')$$

Language Accepted by a PDA

Definition

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

- ▶ The language accepted by final state of P , denoted by $L_{final}(P)$, is defined as

$$L_{final}(P) := \{w \in T^* \mid (q_0, Z_0) \xrightarrow{w}^* (q, z) \text{ for some } q \in F, z \in \Gamma^*\}$$

A final-state PDA is a PDA P for which we define its language to be $L(P) = L_{final}(P)$.

Language Accepted by a PDA

Definition (Cont)

- ▶ The language accepted by empty stack of P , denoted by $L_{empty}(P)$, is defined as

$$L_{empty}(P) := \{w \in T^* \mid (q_0, Z_0) \xrightarrow{w}^* (q, \epsilon) \text{ for some } q \in Q\}$$

An empty-stack PDA is a PDA P for which we define its language to be $L(P) = L_{empty}(P)$.

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

Equivalence Acceptance by Final/Empty Stack

Theorem

Let L be a language. The following are equivalent:

- (1) $L = L(P)$ for a final state PDA P .
- (2) $L = L(P)$ for an empty stack PDA P .

Proof of (1) \Rightarrow (2)

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$.

The idea for constructing an empty stack PDA P' is as follows:

- ▶ P' operates essentially as P .
- ▶ If P reaches an accepting state, then P' can switch to a special state. In that state it empties using ϵ -transition the stack and therefore accepts the string.

Proof of (1) \Rightarrow (2)

- ▶ However it might be that P and therefore as well P' empties the stack without having reached an accepting state.
- ▶ In order to prevent that we make sure that there is a special symbol at the bottom of the stack, which can only be popped when we are in q_{final} .
- ▶ So we start in a new initial state q'_0 with a new initial stack symbol Z'_0 .
- ▶ From that initial configuration P' moves to state q_0 and pushes Z_0 on the stack.
- ▶ We denote the transition relation for P' by \xrightarrow{a}_0 , similarly for the derived relation $\xrightarrow{w^*}_0$.

Resulting PDA

PDA	P'
terminals	T
states	$Q' := Q \cup \{q'_0, q_{final}\}$
stack alphab.	$\Gamma' := \{Z'_0\} \cup \Gamma$
start state	q'_0
start stack	Z'_0
final	\emptyset
transitions	$(q'_0, Z'_0) \xrightarrow{\epsilon}_0 (q_0, Z_0 Z'_0)$ $(q, Z) \xrightarrow{a}_0 (q', z')$ if $(q, Z) \xrightarrow{a} (q', z')$ $(q, Z) \xrightarrow{\epsilon}_0 (q_{final}, Z)$ for $q \in F, Z \in \Gamma'$ $(q_{final}, Z) \xrightarrow{\epsilon}_0 (q_{final}, \epsilon)$ for $Z \in \Gamma'$

Proof of $L_{final}(P) = L_{empty}(P')$

We show that $L_{final}(P) = L_{empty}(P')$ by showing

- ▶ $L_{final}(P) \subseteq L_{empty}(P')$,
- ▶ $L_{empty}(P') \subseteq L_{final}(P)$.

Proof of $L_{final}(P) \subseteq L_{empty}(P')$

Assume $w \in L_{final}(P)$.

Then

$$(q_0, Z_0) \xrightarrow{w}^* (q, z) \text{ for some } q \in F, \quad z \in \Gamma^*$$

Then

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w}^* (q, z Z'_0) \xrightarrow{\epsilon}^* (q_{final}, z Z'_0) \xrightarrow{\epsilon}^* (q_{final}, \epsilon)$$

so $w \in L_{empty}(P')$.

Proof of $L_{empty}(P') \subseteq L_{final}(P)$

Assume $w \in L_{empty}(P')$.

Then

$$(q'_0, Z'_0) \xrightarrow{w}^* (q, \epsilon) \text{ for some } q \in Q', \quad z \in \Gamma'^*$$

- ▶ Since $Z'_0 \neq \epsilon$, the first step must have been

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0)$$

- ▶ Then there must have been some transitions inherited from P .
- ▶ Since there are no transitions of this kind with stack symbol Z'_0 , the symbol Z'_0 must have stayed at the bottom.
- ▶ So we never reached an empty state while staying continuously in states in Q .

Proof of $L_{empty}(P') \subseteq L_{final}(P)$

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w}^* (q, \epsilon)$$

- ▶ Let $(q', z Z'_0)$ be the last configuration, where we stayed continuously since reaching state q_0 in a state in Q .
- ▶ Then we have

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w_0}^* (q', z Z'_0) \xrightarrow{a} (q'', z') \xrightarrow{w_1}^* (q, \epsilon)$$

s.t. $w = w_0 a w_1$,

$$(q_0, Z_0) \xrightarrow{w_0}^* (q', z)$$

and $q'' \notin Q$.

- ▶ Then state $q'' = q_{final}$, $q' \in F$, $a = \epsilon$. Furthermore we obtain that

$$(q_{final}, z') \xrightarrow{w_1}^* (q, \epsilon)$$

must have been a sequence of pops from the stack so we have $w_1 = \epsilon$, $q = q_{final}$.

Proof of $L_{empty}(P') \subseteq L_{final}(P)$

- Therefore $w = w_0aw_1 = w_0$, the overall transition was

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0Z'_0) \xrightarrow{w} (q', zZ'_0) \xrightarrow{\epsilon} (q_{final}, zZ'_0) \xrightarrow{\epsilon} (q_{final}, \epsilon)$$

and we have

$$(q_0, Z_0) \xrightarrow{w} (q', z)$$

Therefore $w \in L(P)^*$.

Proof of (2) \Rightarrow (1)

Let $P = (T, Q, \Gamma, q_0, Z_0, F, \delta)$.

The idea for constructing a final state PDA P' is as follows:

- P' keeps as before a special stack symbol Z'_0 at the bottom of the stack.
- It operates as P , until it observes that the top stack symbol is Z'_0 .
- This indicates that P would have reached an empty stack and therefore accepted the string.
- Therefore P' moves into a special accepting state q_{final} and terminates.

Resulting PDA

PDA	P'
terminals	T
states	$Q' := Q \cup \{q'_0, q_{final}\}$
stack alphab.	$\Gamma' := \{Z'_0\} \cup \Gamma$
start state	q'_0
start stack	Z'_0
final	q_{final}
transitions	$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0Z'_0)$ $(q, Z) \xrightarrow{a} (q', z')$ if $(q, Z) \xrightarrow{a} (q', z')$ $(q, Z_0) \xrightarrow{\epsilon} (q_{final}, \epsilon)$ for $q \in Q$

Proof of $L_{empty}(P) = L_{final}(P')$

Again we show

- $L_{empty}(P) \subseteq L_{final}(P')$.
- $L_{final}(P') \subseteq L_{empty}(P)$.

Proof of $L_{empty}(P) \subseteq L_{final}(P')$

Assume $w \in L_{empty}(P)$.

Then

$$(q_0, Z_0) \xrightarrow{w}^* (q, \epsilon) \text{ for some } q \in Q$$

Then

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w}^* (q, Z'_0) \xrightarrow{\epsilon} (q_{final}, \epsilon)$$

so $w \in L_{final}(P')$.

Proof of $L_{final}(P') \subseteq L_{empty}(P)$

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w}^* (q_{final}, z)$$

► Let (q', zZ'_0) be the last configuration, where we stayed continuously since reaching state q_0 in a state in Q .

► Then we have

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w_0}^* (q', z' Z'_0) \xrightarrow{a} (q'', z'') \xrightarrow{w_1}^* (q_{final}, z)$$

s.t. $w = w_0 a w_1$,

$$(q_0, Z_0) \xrightarrow{w_0}^* (q', z')$$

and $q'' \notin Q$.

► Then state $q'' = q_{final}$, $a = \epsilon$, $(q_{final}, z) = (q'', z'')$, $w_1 = \epsilon$, and since $z' \in \Gamma^*$ and we get into q_{final} only using stack symbol Z'_0 , we get $z' = \epsilon$.

Proof of $L_{final}(P') \subseteq L_{empty}(P)$

Assume $w \in L_{final}(P')$.

Then

$$(q'_0, Z'_0) \xrightarrow{w}^* (q_{final}, z) \text{ for some } z \in \Gamma^*$$

► Since q'_0 is not accepting, the first step must have been

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0)$$

► Then there must have been some transitions inherited from P .

► Since there are no transitions of this kind with stack symbol Z'_0 , the symbol Z'_0 must have stayed at the bottom while having transitions inherited from P .

Proof of $L_{final}(P') \subseteq L_{empty}(P)$

► So $w = w_0 a w_1 = w_0$, and we have

$$(q'_0, Z'_0) \xrightarrow{\epsilon} (q_0, Z_0 Z'_0) \xrightarrow{w}^* (q', Z'_0) \xrightarrow{\epsilon} (q_{final}, \epsilon)$$

and

$$(q_0, Z_0) \xrightarrow{w}^* (q', \epsilon)$$

► Therefore $w \in L(P)^*$.

2.4.1. Derivation Trees for Context-Free Grammars (14.1)

2.4.2. Normal Forms for Context-Free Grammars (14.2)

2.4.3. The Pumping Lemma for CFG (14.4)

2.4.4. Limitations of Context Free Grammars

2.4.5. Push Down Automata

2.4.6. Equivalence of Final-State and Empty-Stack-PDAs

2.4.7. Equivalence of CFG and PDA

Equivalence of empty stack PDA and CFG

Theorem

Let L be a language. The following are equivalent:

- (1) $L = L(G)$ for a CFG G .
- (2) $L = L(P)$ for an empty stack PDA P .

Equivalence Acceptance by Final/Empty Stack

We are going to show that the context free languages are exactly the languages which can be recognised by a (non-deterministic) empty stack PDA.

Since empty stack PDA are final state PDA recognise the same languages, the context free languages are exactly the languages which can be recognised by a PDA.

Corollary

Because empty stack and final state PDAs are equivalent, from the theorem follows immediately the following corollary:

Corollary

Let L be a language. The following are equivalent:

- (1) $L = L(G)$ for a CFG G .
- (2) $L = L(P)$ for an empty stack PDA P .
- (2) $L = L(P)$ for a final state PDA P .

Proof of the Theorem

We will only show $(1) \Rightarrow (2)$, $(2) \Rightarrow (1)$ is quite sophisticated.

PDA based on LL-Parsing

- ▶ LL-parsing stands for left-to-right parsing based on a leftmost derivation.
- ▶ It constructs a leftmost derivation top down, and is therefore an example of a top down parser.
- ▶ We take as example the grammar with the rules

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \\ A &\longrightarrow ab \\ C &\longrightarrow cCd \\ C &\longrightarrow cd \end{aligned}$$

- ▶ We consider a left-most derivation

$$S \Rightarrow AC \Rightarrow aAbC \Rightarrow aabbC \Rightarrow aabbcCd \Rightarrow aabbccdd$$

Proof of $(1) \Rightarrow (2)$

Assume $G = (T, N, S, P)$ is a CFG. We need to construct a PDA which simulates G .

There are two ways of constructing such a PDA, one follows the LL-parsing method, one uses the LR-parsing method.

LL parsing will result in a PDA with a single stack The configuration can be given by its stack.

We will introduce both methods, because deterministic versions of them are the basis for many parsers.

We will then carry the proof of $(1) \Rightarrow (2)$ out by showing for the PDA based on the LL parser that it is equivalent to the CFG.

Example of LL Parsing

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \quad A \longrightarrow ab \\ C &\longrightarrow cCd \quad C \longrightarrow cd \\ S &\Rightarrow AC \Rightarrow aAbC \Rightarrow aabbC \Rightarrow aabbcCd \Rightarrow aabbccdd \end{aligned}$$

- ▶ We start on our PDA with stack S for the start symbol.
 - ▶ So we have stack S
- ▶ We guess that the rule is $S \longrightarrow AC$, and replace the top symbol S on the stack by AC .
 - ▶ So we have stack AC (the two symbol is the left most one).
- ▶ We guess that the rule for the top symbol is $A \longrightarrow aAb$, and replace the top symbol A on the stack by aAb .
 - ▶ So we have stack $aAbC$ (the top of the stack is the symbol most to the left).

Example of LL Parsing

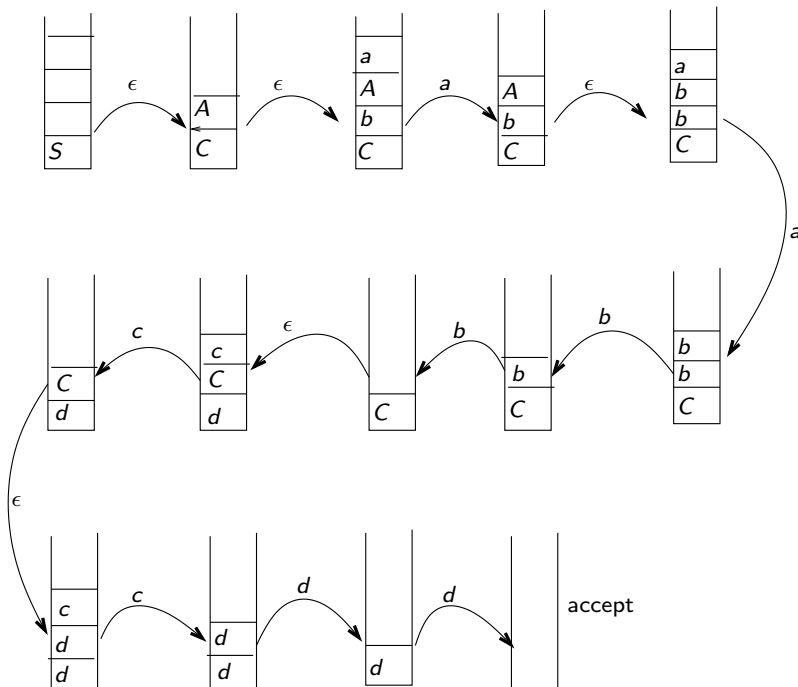
$$S \rightarrow AC$$

$$A \rightarrow aAb \quad A \rightarrow ab$$

$$C \rightarrow cCd \quad C \rightarrow cd$$

$$S \Rightarrow AC \Rightarrow aAbC \Rightarrow aabbC \Rightarrow aabbcCd \Rightarrow aabbccdd$$

- ▶ We have now stack $aAbC$. We can accept the letter a , and remove this symbol a from the stack.
 - ▶ So we have stack AbC
- ▶ We guess that the rule is $A \rightarrow ab$, and replace the top symbol A on the stack by ab .
 - ▶ So we have stack $abbC$
- ▶ Now we can 3 times accept a letter, namely a , b , b and remove them from the stack.
 - ▶ So we have stack C .



Example of LL Parsing

$$S \rightarrow AC$$

$$A \rightarrow aAb \quad A \rightarrow ab$$

$$C \rightarrow cCd \quad C \rightarrow cd$$

$$S \Rightarrow AC \Rightarrow aAbC \Rightarrow aabbC \Rightarrow aabbcCd \Rightarrow aabbccdd$$

- ▶ We have now stack C . We use rule $C \rightarrow cCd$ and replace C by cCd .
 - ▶ So we have stack cCd
- ▶ We consume the next letter c and accept it and remove c from the stack.
 - ▶ So we have stack Cd
- ▶ We expand C using $C \rightarrow cd$ to cd .
 - ▶ So we have stack cdd .
- ▶ Now we consume and accept letters c , d , d , clear them from the stack, and then accept the word because we have reached the empty stack.

LL(k) Parsing

- ▶ We had to guess which production to use.
- ▶ $LL(k)$ is an algorithm to decide by using a lookahead of the next k symbols, which production to use.
- ▶ For instance when the stack was AC we could have guessed by knowing that the next two letters are a , a , that we need to use the production $A \rightarrow aAb$ and not $A \rightarrow ab$, since the latter would give as next two letters ab .
- ▶ Usually only a lookahead of 1 symbol is used, but most standard grammars (e.g. Java) have no $LL(1)$ grammars.

Invariant

- ▶ The invariant kept above was that when having consumed string w and obtained stack v , then we have $S \Rightarrow^* wv$.

Resulting PDA

- ▶ The PDA has the following productions:
 - ▶ If the top symbol is a non-terminal A , and there was a rule $A \rightarrow w$, we could replace A by w , and make an ϵ -transition. So we have in this case a transition

$$A \xrightarrow{\epsilon} w$$

- ▶ If the top symbol was a terminal a , then we could accept a word a and pop the symbol from the stack. So we have transitions

$$a \xrightarrow{a} \epsilon$$

Resulting PDA

- ▶ We obtain a PDA with a single state. Therefore we omit the state, and ignore the start state, set of states, final states, the state in configurations, and have transitions

$$Z \xrightarrow{a} z$$

for $Z \in \Gamma$, $a \in T$, $z \in \Gamma^*$.

- ▶ The stack symbols were the alphabet used in strings occurring in the derivations, i.e. $T \cup N$.
- ▶ The start stack symbol was S .

Resulting Empty Stack Single State PDA

The empty stack PDA derived from $G = (T, N, S, P)$ is as follows:

PDA	P
terminals	T
states	single state
stack alphanb.	$T \cup N$
start stack	S
transitions	$A \xrightarrow{\epsilon} w$ if $A \rightarrow w \in P$ $a \xrightarrow{a} \epsilon$ if $a \in T$

We will show later that $L(P) = L(G)$.

PDA based on LR-Parsing

- ▶ LR-parsing stands for left-to-right parsing based on a rightmost derivation.
- ▶ LR parsing constructs a right most derivation bottom up.
- ▶ It is therefore an example of a bottom up parser.

Principles of LR Parsing

- ▶ Whereas LL parsing has no state, for LR parsing we need a state, since we need sometimes to pop several elements from the stack in sequence in order to replace them by a nonterminal.
- ▶ This state will occur in the final PDA.

Principles of LR Parsing

- ▶ Remember that for LL parsing we had the invariant
 - ▶ Having consumed string w and stack v then $S \Rightarrow^* wv$.
- ▶ For LR parsing the invariant will be
 - ▶ After having consumed w and having stack v then $v^R \Rightarrow^* w$.
So from the stack in reverse order we can derive the string consumed so far.
- ▶ Since we are generating a nondeterministic PDA, we don't need information to decide which action to take.
- ▶ LR parsers constructed by compiler generators are deterministic. In order to allow a decision, they put between the symbols from $L \cup N$ elements representing a state, which gives information about what is on the stack in order to give the information needed to decide which action is to be taken.

Principles of LR Parsing

- ▶ Initially we haven't consumed anything yet, but we need a start stack symbol. We call this symbol Z_0 .
- ▶ This stack symbol will stay at the bottom of the stack, and will only be popped if we accept the string.
- ▶ So the invariant needs to be modified in so far that from the stack excluding this bottom symbol in reverse order we can derive the string consumed so far.

PDA based on LR-Parsing

- ▶ We take as example the same grammar as before, so we have the grammar with the rules

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \\ A &\longrightarrow ab \\ C &\longrightarrow cCd \\ C &\longrightarrow cd \end{aligned}$$

- ▶ We consider a right-most derivation

$$S \Rightarrow AC \Rightarrow AcCd \Rightarrow Accdd \Rightarrow aAbccdd \Rightarrow aabbccdd$$

Example of LL Parsing

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \quad A \longrightarrow ab \\ C &\longrightarrow cCd \quad C \longrightarrow cd \\ S &\Rightarrow AC \Rightarrow AcCd \Rightarrow Accdd \Rightarrow aAbccdd \Rightarrow aabbccdd \end{aligned}$$

- ▶ We have stack AZ_0 .
- ▶ We consume the letters c, c, d and push them on the stack.
 - ▶ So we have stack $dccAZ_0$.
- ▶ Now we reduce dc to C by using the rule $C \longrightarrow cd$.
- ▶ So we pop d and c from the stack and push C on to it.
 - ▶ So we have stack $CcAZ_0$.
- ▶ We consume letter d and push it on the stack.
 - ▶ We have stack $dCcAZ_0$.
- ▶ Now we reduce dCc to C by using the rule $C \longrightarrow dCc$.
- ▶ So we pop dCc from the stack and replace it by C .
 - ▶ We have stack CAZ_0 .

Example of LL Parsing

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \quad A \longrightarrow ab \\ C &\longrightarrow cCd \quad C \longrightarrow cd \\ S &\Rightarrow AC \Rightarrow AcCd \Rightarrow Accdd \Rightarrow aAbccdd \Rightarrow aabbccdd \end{aligned}$$

- ▶ We start on our PDA with stack Z_0 for the start symbol.
- ▶ We consume the first letter a and push it on the stack.
 - ▶ So we have stack aZ_0 .
- ▶ We consume the next letters a, b and push them on the stack.
 - ▶ So we have stack $baaZ_0$.
- ▶ Now we reduce ba to A by using the rule $A \longrightarrow ab$.
- ▶ So we pop a and b from the stack and push A on to it.
 - ▶ So we have stack AaZ_0 .
- ▶ We consume letter b and push it on the stack.
 - ▶ We have stack $bAaZ_0$.
- ▶ Now we reduce bAa to A by using the rule $A \longrightarrow aAb$.
- ▶ So we pop bAa from the stack and replace it by A .
 - ▶ We have stack AZ_0 .

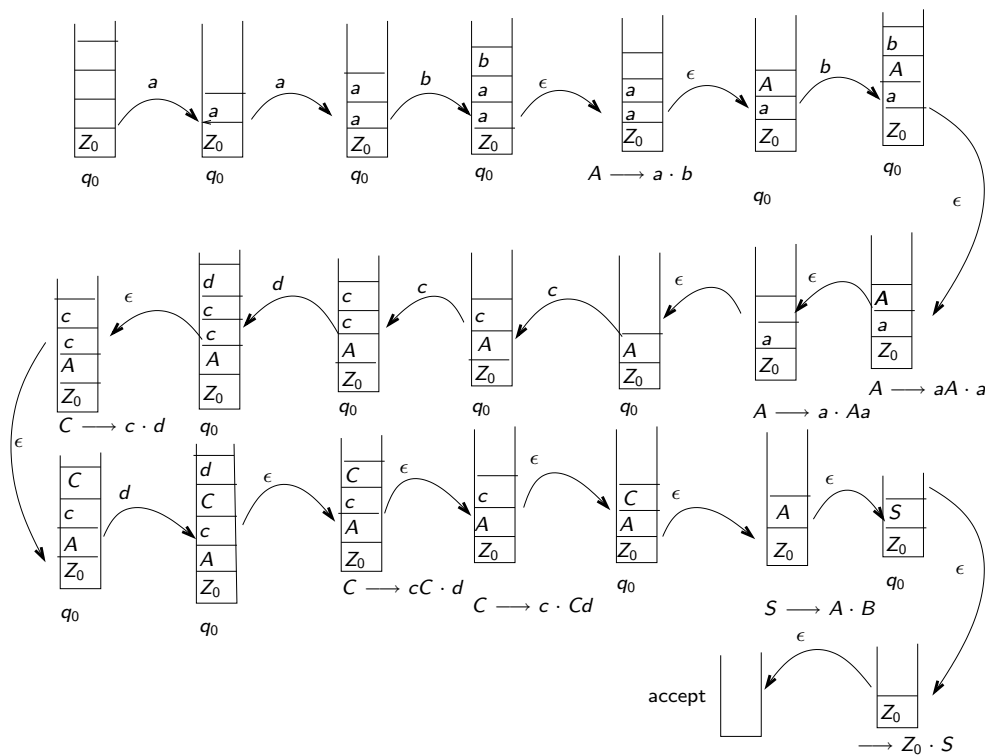
Example of LL Parsing

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \quad A \longrightarrow ab \\ C &\longrightarrow cCd \quad C \longrightarrow cd \\ S &\Rightarrow AC \Rightarrow AcCd \Rightarrow Accdd \Rightarrow aAbccdd \Rightarrow aabbccdd \end{aligned}$$

- ▶ We have stack CAZ_0 .
- ▶ Now we reduce CA to S by using the rule $S \longrightarrow AC$.
- ▶ So we pop CA from the stack and replace it by S .
 - ▶ We have stack SZ_0 .
- ▶ Now we see that we have the start symbol and below the bottom of the stack.
 - ▶ We pop those two symbols and accept with empty stack.

Simplification

- ▶ In the above example we have optimized the PDA by when using a rule immediately popping a symbol, and when the last symbol is popped immediately replacing it by the non terminal to which it was reduced.
- ▶ It is easier to present the PDA without this optimization, and we do so in the following.
- ▶ This means that we have as well states $A \rightarrow w \cdot$, and $A \rightarrow \cdot w$, for the beginning and end of a reduce sequence (where we replace a stack consisting of w by A).



Resulting PDA (LR Parsing)

- ▶ We obtain a PDA with states.
- ▶ Stack symbols are $\Gamma := \{Z_0\} \cup T \cup N$.
- ▶ States are
 - ▶ q_0 ,
 - ▶ $A \rightarrow v \cdot w$ where $A \rightarrow vw$ is a production,
 - ▶ $\rightarrow Z_0 \cdot S$.
- ▶ Start state is q_0 , start stack symbol is Z_0 .

Resulting PDA (LR Parsing)

We obtain the following transitions ($a \in T, Z \in \Gamma, w, v \in (T \cup N)^*, A, B \in N, A \rightarrow w, B \rightarrow uZv \in P$)

$$\begin{aligned}
 (q_0, Z) & \xrightarrow{a} (q_0, aZ) \\
 (q_0, Z) & \xrightarrow{\epsilon} (A \rightarrow w \cdot, Z) \\
 (B \rightarrow uZ \cdot v, Z) & \xrightarrow{\epsilon} (B \rightarrow u \cdot Zv, \epsilon) \\
 (A \rightarrow \cdot w, Z) & \xrightarrow{\epsilon} (q_0, AZ) \\
 (q_0, S) & \xrightarrow{\epsilon} (\rightarrow Z_0 \cdot S, \epsilon) \\
 (\rightarrow Z_0 \cdot S, Z_0) & \xrightarrow{\epsilon} (q_0, \epsilon)
 \end{aligned}$$

Correctness of the LL Parser

We are going to show that for the LL Parser P we had given $L(P) = L(G)$.
We repeat the definition of P :

PDA	P
terminals	T
states	single state
stack alphan.	$T \cup N$
start stack	S
transitions	$A \xrightarrow{\epsilon} w$ if $A \rightarrow w \in P$ $a \xrightarrow{a} \epsilon$ if $a \in T$

Proof of $L(P) \subseteq L(G)$

Show:

$S \xrightarrow{x}^* w$ implies $S \Rightarrow^* xw$

- ▶ Base case: Length zero.
Then $x = \epsilon$, $S = w$, and we trivially get $S \Rightarrow^* xw$.
- ▶ Induction step.
Assume

$$S \xrightarrow{x_1}^* w_1 \xrightarrow{a} w$$

where $w_1 \in (T \cup N)^*$, $x_1 \in T^*$, $a \in T \cup \{\epsilon\}$, $x = x_1 a$.

- ▶ Case 1: $w_1 = Aw_2$, $A \rightarrow u$, $a = \epsilon$, $w = uw_2$, $x = x_1$ and the transition is

$$S \xrightarrow{x}^* w_1 = Aw_2 \xrightarrow{\epsilon} uw_2$$

By IH

$$S \Rightarrow^* xw_1 = xAw_2$$

and by $A \rightarrow u$ we have

$$S \Rightarrow^* xw_1 = xAw_2 \Rightarrow xuw_2 = xw$$

Proof of $L(P) \subseteq L(G)$

- ▶ Remember that because of single state configurations are given by the stack only.
- ▶ We show by induction on the length of the derivation:
 - ▶ If in P we have for $x \in T^*$, $w \in (T \cup N)^*$

$$S \xrightarrow{x}^* w$$

then we have in G

$$S \Rightarrow^* xw$$

- ▶ Then we get

$$\begin{aligned} w \in L(P) &\text{ implies } S \xrightarrow{w}^* \epsilon \\ &\text{ implies } S \Rightarrow^* w \\ &\text{ implies } w \in L(G) \end{aligned}$$

so $L(P) \subseteq L(G)$.

Proof of $L(P) \subseteq L(G)$

Show:

$S \xrightarrow{x}^* w$ implies $S \Rightarrow^* xw$

We have $S \xrightarrow{x_1}^* w_1 \xrightarrow{a} w$

- ▶ Case 2: $w_1 = aw$, $a \in T$, and the transition is

$$S \xrightarrow{x_1}^* w_1 = aw \xrightarrow{a} w$$

Then by IH

$$S \Rightarrow^* x_1 w_1 = x_1 aw = xw$$

Proof of $L(G) \subseteq L(P)$

- ▶ We show by induction on the length of the left-most derivation:

- ▶ If in G we have for $x \in T^*$, $A \in N$, $w \in (T \cup N)^*$

$$S \Rightarrow^* xAw$$

for a left-most derivation, then we have in P

$$S \xrightarrow{x}^* Aw$$

- ▶ If in G we have for $x \in T^*$ for a left-most derivation

$$S \Rightarrow^* x$$

then we have in P

$$S \xrightarrow{x}^* \epsilon$$

- ▶ Then we get

$$\begin{aligned} w \in L(G) & \text{ implies } S \Rightarrow^* w \\ & \text{ implies } S \xrightarrow{w}^* \epsilon \\ & \text{ implies } w \in L(P) \end{aligned}$$

so $L(G) \subseteq L(P)$.

Proof of $L(G) \subseteq L(P)$

Show:

$$S \Rightarrow^* xAw \text{ implies } S \xrightarrow{x}^* Aw$$

$$S \Rightarrow^* x \text{ implies } S \xrightarrow{x}^* \epsilon$$

We have

$$S \Rightarrow^* x_1Bw_1 \Rightarrow xAw \text{ or } S \Rightarrow^* x_1Bw_1 \Rightarrow x$$

$$\text{and by IH } S \xrightarrow{x_1}^* Bw_1.$$

- ▶ Case 1: $S \Rightarrow^* x_1Bw_1 \Rightarrow xAw$. Then there exist a production $B \rightarrow u$, $x_1uw_1 = xAw$. Therefore there exists x_2 s.t. $x = x_1x_2$. Now we get

$$S \xrightarrow{x_1}^* Bw_1 \xrightarrow{\epsilon} uw_1 = x_2Aw \xrightarrow{x_2}^* Aw$$

and by $x = x_1x_2$ follows the assertion.

Proof of $L(G) \subseteq L(P)$

Show:

$$S \Rightarrow^* xAw \text{ implies } S \xrightarrow{x}^* Aw$$

$$S \Rightarrow^* x \text{ implies } S \xrightarrow{x}^* \epsilon$$

- ▶ Base case: Length zero.

Then $x = w = \epsilon$, $S = A$, and we trivially get $S \xrightarrow{\epsilon}^* Aw$

- ▶ Induction step.

Assume

$$S \Rightarrow^* x_1Bw_1 \Rightarrow xAw \text{ or } S \Rightarrow^* x_1Bw_1 \Rightarrow x$$

where $x_1 \in T^*$, $B \in N$, $w_1 \in (T \cup N)^*$.

By IH

$$S \xrightarrow{x_1}^* Bw_1$$

Proof of $L(P) \subseteq L(G)$

Show:

$$S \Rightarrow^* xAw \text{ implies } S \xrightarrow{x}^* Aw$$

$$S \Rightarrow^* x \text{ implies } S \xrightarrow{x}^* \epsilon$$

We have

$$S \Rightarrow^* x_1Bw_1 \Rightarrow xAw \text{ or } S \Rightarrow^* x_1Bw_1 \Rightarrow x$$

$$\text{and by IH } S \xrightarrow{x_1}^* Bw_1$$

- ▶ Case 2: $S \Rightarrow^* x_1Bw_1 \Rightarrow x$.

Then there exist a production $B \rightarrow u$, $x_1uw_1 = x$.

Now we get

$$S \xrightarrow{x_1}^* Bw_1 \xrightarrow{\epsilon} uw_1 \xrightarrow{uw_1}^* \epsilon$$

and by $x_1uw_1 = x$ follows the assertion.