Monash University
Faculty of Information Technology

FIT2014 Theory of Computation

Lecture 14

Context-Free Languages and Pushdown Automata

slides by Graham Farr

Overview

- CFL $\longrightarrow$ PDA
- PDA $\longrightarrow$ CFL

CFL $\longleftrightarrow$ PDA

We will show . . .

$$\{\,\text{CFLs}\,\} \;=\; \{\,\text{languages recognised by a PDA}\,\}$$

. . . in two parts:

1. $\{\,\text{CFLs}\,\} \;\subseteq\; \{\,\text{languages recognised by a PDA}\,\}$

2. $\{\,\text{languages recognised by a PDA}\,\} \;\subseteq\; \{\,\text{CFLs}\,\}$

CFL $\longrightarrow$ PDA

Theorem.
$\{\text{CFLs}\} \subseteq \{\text{languages recognised by a PDA}\}$

Proof outline and main ideas:

Let L be a CFL.
Let G be a CFG for L.

We need to show that there is a PDA that recognises L.

If $w \in L$ then w has a leftmost derivation.

Idea: leftmost derivation may be viewed as

- growing a prefix of w that we know to be correct, and
- managing the rest of w (including all nonterminals) with a stack.

Leftmost derivation: stack view

Grammar fragment:

$\dots$	$\dots$	$\dots$
S	$\longrightarrow$	$\mathtt{pe}X$
X	$\longrightarrow$	$Y\mathtt{c}Z$
Y	$\longrightarrow$	$\mathtt{ar}$
Z	$\longrightarrow$	$\mathtt{ey}$
$\dots$	$\dots$	$\dots$

S		pearcey	
$\Longrightarrow$	$\mathtt{pe}X$	pearcey	X
$\Longrightarrow$	$\mathtt{pe}Y\mathtt{c}Z$	pearcey	$Y\mathtt{c}Z$
$\Longrightarrow$	$\mathtt{pearc}Z$	pearcey	$\mathtt{c}Z$
		pearcey	Z
$\Longrightarrow$	pearcey	pearcey	

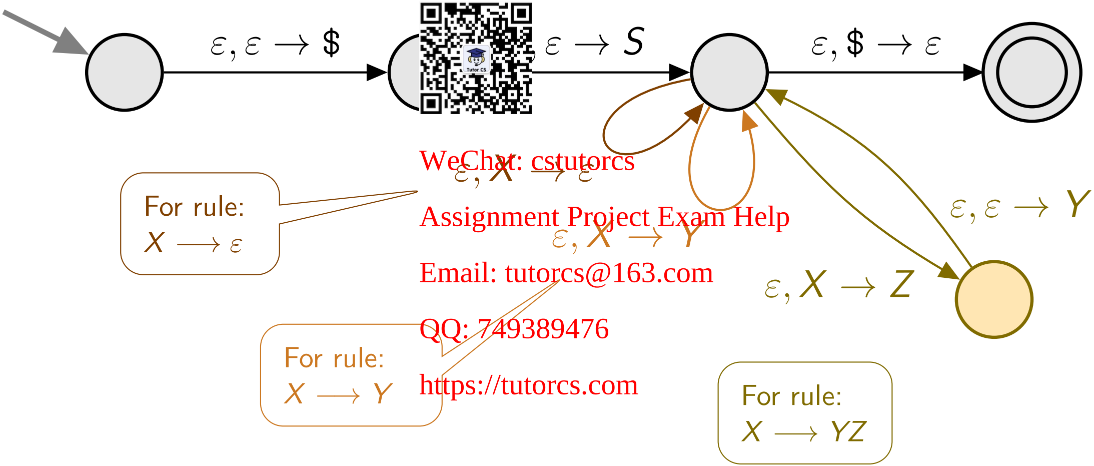
CFL $\longrightarrow$ PDA

We construct the required PDA as follows.

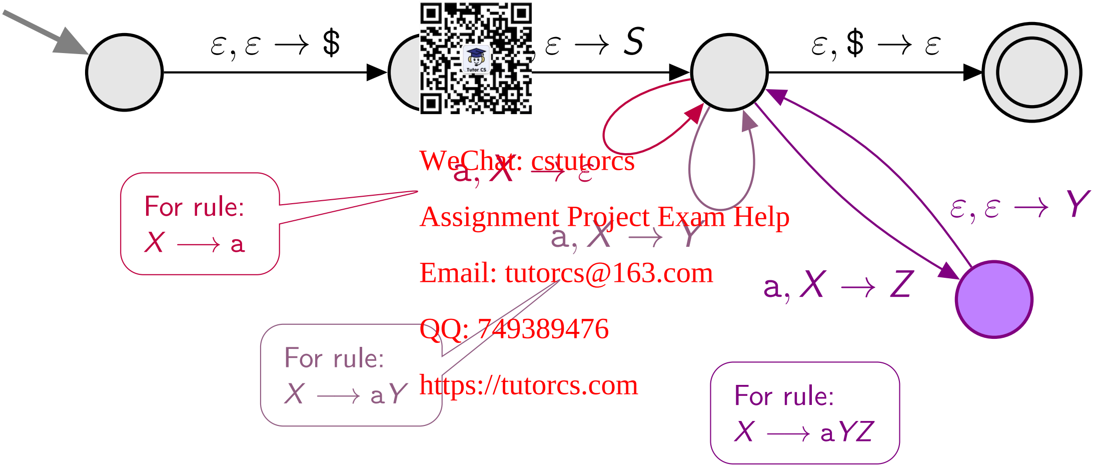
We start with four basic states, then add more states for each production rule . . .

We'll need a new character (not currently a terminal or non-terminal), to mark the end of our stack.
We'll use $\$$.

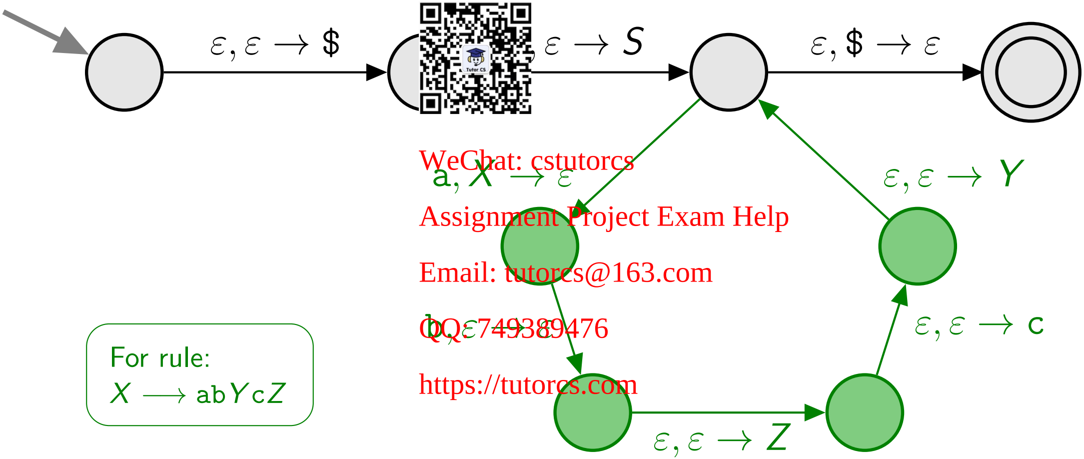
CFL $\longrightarrow$ PDA



CFL $\longrightarrow$ PDA

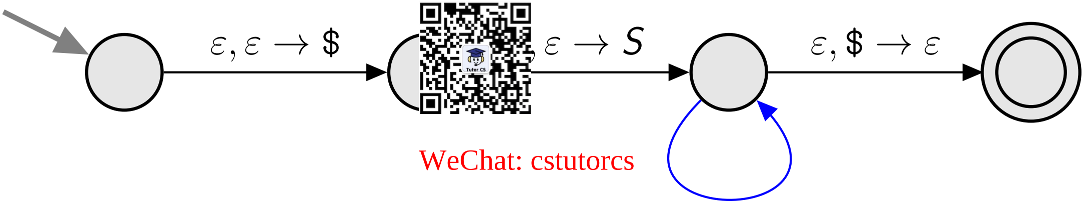


CFL $\longrightarrow$ PDA



For rule:
$X \longrightarrow \mathtt{ab}Y\mathtt{c}Z$

CFL $\longrightarrow$ PDA



If a terminal is on top of stack:
everything before it in target string
must have been read in.

So we need loop transitions
to check such letters off . . .

$a, a \to \varepsilon$
$b, b \to \varepsilon$
$c, c \to \varepsilon$
$\vdots \quad \vdots$

For terminals

CFL $\longrightarrow$ PDA

This construction gives a PDA that accepts precisely those strings with a leftmost derivation by G,

i.e., precisely those strings with a derivation by G,

i.e., precisely those strings in L.

Full formal proof: see Sipser, Ch. 2, Section 2.2.

Now for the other way round ...

PDA $\longrightarrow$ CFL

Theorem.
$\{$ languages recognised by a PDA $\} \subseteq \{$ CFLs $\}$

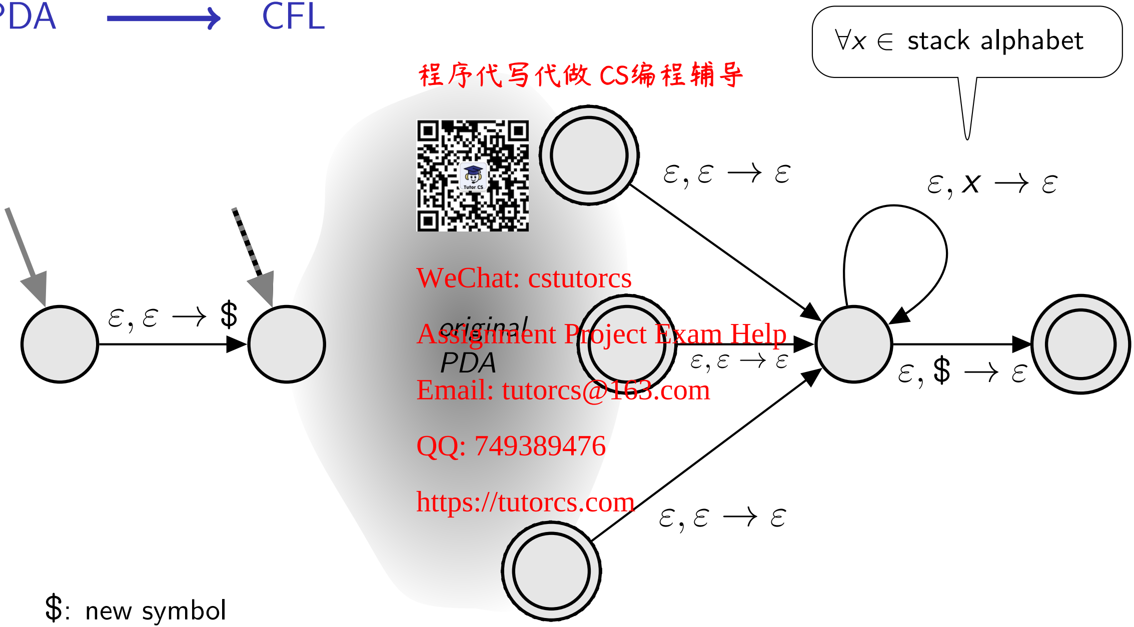
Proof ideas:

Let L be a langauge recognised by some PDA M.
We need to show that $\exists$ a CFG G that generates L.

First, we make some simple modifications to M.
Then we give productions that describe certain ways of going through the PDA ...

First, modifications to M:
Ensure it has just one Final State,
and that the stack is empty when it reaches the Final State.

PDA $\longrightarrow$ CFL

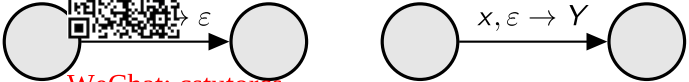


\$: new symbol

PDA ⟶ CFL

More modifications: ensure that each transition *either* pushes *or* pops, but *not both*.

These are ok:



These are *not*:

So we change them . . .

PDA ⟶ CFL

$x, Y \to Z$
$x, \varepsilon \to \varepsilon$
$x, Y \to \varepsilon$
new
$\varepsilon, \varepsilon \to Z$
$x, \varepsilon \to W$
new
$\varepsilon, W \to \varepsilon$

PDA $\longrightarrow$ CFL

A string is accepted by this (modified) M if one of its paths through M

- starts in the Start State s,
- finishes in the Final State t,
- with the stack empty at start and finish.

For every pair of states p, q, define a non-terminal symbol A_{pq}:

- intended to generate all strings which, starting at p with an empty stack, can take some path through M which ends at q with an empty stack.

Aim: a grammar such that, for every string,

$$\text{it is accepted by } M \iff \text{it can be derived from } A_{st}.$$

PDA ⟶ CFL

Consider how a computation in M, for a string w, moves from p to q, with empty stack at start and finish.

We have two cases:

Case 1:
The computation also has an empty stack at some other state r on the path.

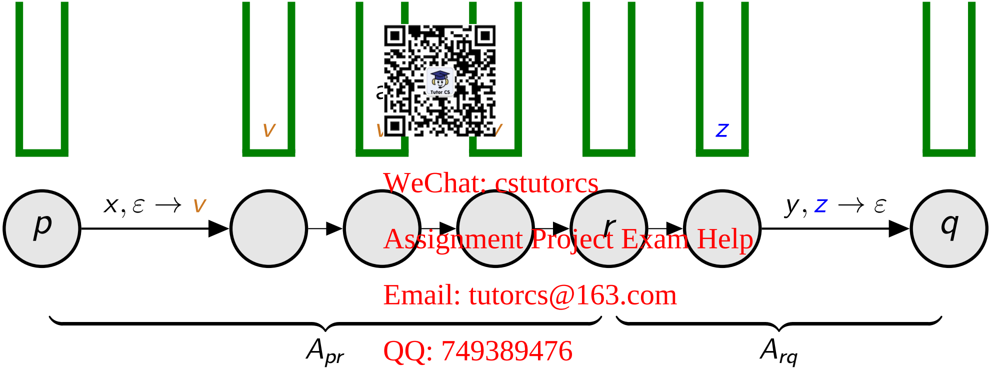
Then we can break the computation from p to q into two parts:

- the first part, going from p to r (starting and ending with empty stack),
- the second part, going from r to q (starting and ending with an empty stack).

We model this with the production

$$A_{pq} \longrightarrow A_{pr} A_{rq}$$

PDA $\longrightarrow$ CFL



$$A_{pq} \longrightarrow A_{pr}A_{rq}$$

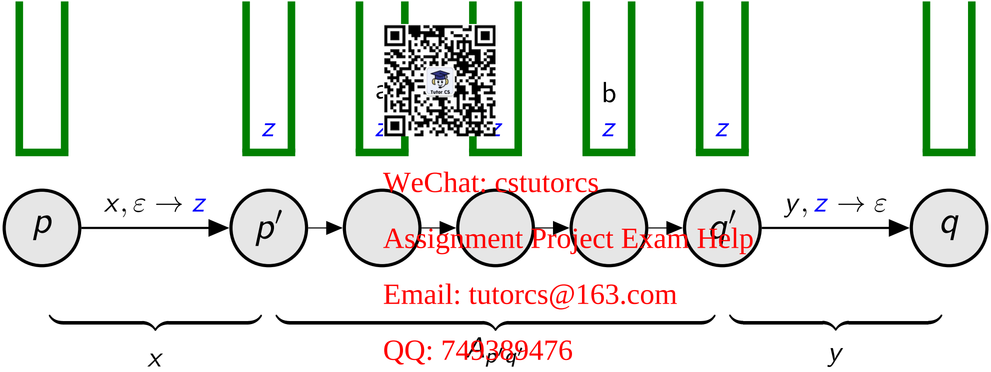
PDA $\longrightarrow$ CFL

Case 2:
The computation never has an [illegible]ack, except at p and q.

Because it starts and finishes with an empty stack:

- the first transition must push a symbol onto the stack,
- the last transition must pop a symbol from the stack,
- the two symbols must be the same (call it z)
  - …else the stack would have to have been emptied at some stage, to remove the first symbol before the last symbol arrives.
- and this symbol stays at the bottom of the stack the whole time.

PDA $\longrightarrow$ CFL



$$A_{pq} \longrightarrow xA_{p'q'}y$$

PDA $\longrightarrow$ CFL

In the computation from p' to q', the stack is not empty, but it always has z sitting at the bottom.

The "substack" above z is empty at p' and q'.

The computation path from p' to q' starts and ends with a stack containing just z, with z on the bottom of every stack along the way.

This is equivalent to starting and ending with an empty stack.

We model this with the production

$$A_{pq} \longrightarrow xA_{p'q'}y$$

PDA $\longrightarrow$ CFL

Also, for each state p, add the production

$$A_{pp} \longrightarrow \varepsilon$$

Finally, add the production

$$S \longrightarrow A_{st}$$

where, as usual, the non-terminal S is the Start symbol.

This set of productions give a CFG for L.

For formal proof (making good use of induction), see Sipser.

Revision

Some things to think about:

- CFG $\longrightarrow$ PDA:
  - What conditions would [illegible]ave to satisfy, so that the PDA we construct is *deterministic*?
  - If the PDA produced by this construction *only has the four states we started with* — so that the extra transitions we added are *all loops* — what can we say about the language we started with?
- CFG $\longrightarrow$ PDA $\longrightarrow$ CFG:
  - If you start with a CFG and then do the construction both ways to get another CFG, will it ever be the same as the CFG you started with?

Reading: Sipser, pp. 117–125