Monash University
Faculty of Information Technology

Lecture 11
Regular Grammars and Pushdown Automata

Slides by David Albrecht (2011) and Graham Farr (2013).

FIT2014 Theory of Computation

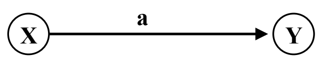
Overview

- Regular Languages $\subseteq$ CFL
- Regular Grammars
- Pushdown Automaton (PDA)
- Constructing PDA to accept a Regular Grammar
- CFL = PDA

Is every Regular Language a Context Free Language?

Nondeterministic Finite Automaton (NFA) to a Context Free Grammar (CFG)

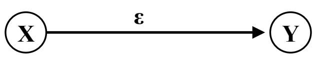
1. Name all the states in the NFA by a symbol.
   - Call the Start State S.
2. For each edge



write:

$$\mathbf{X \rightarrow aY}$$

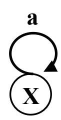
3. For each edge



write:

$$\mathbf{X \rightarrow Y}$$

4. For each edge



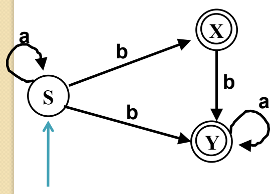
write:

$$\mathbf{X \rightarrow aX}$$

5. For each Final State X write:

$$\mathbf{X \rightarrow \varepsilon}$$

Example



$S \to aS$
$S \to bX$
$S \to bY$
$X \to bY$
$Y \to aY$
$X \to \varepsilon$
$Y \to \varepsilon$

Definitions

- **Semiwords** are of the form:
  terminal terminal … terminal <u>Nonterminal</u>

- A CFG is called a **Regular Grammar** if all its production rules are in one of the following forms:

  Nonterminal → semiword

  or

  Nonterminal → string of terminals

Theorem

Every **Regular Language** can be generated by a **Regular Grammar**.

Proof (sketch):

A regular language is recognised by some NFA.

Observe that our construction NFA ➔ CFG produces a regular grammar.

Q.E.D.

Theorem

Every **Regular Grammar** generates a **Regular Language**.

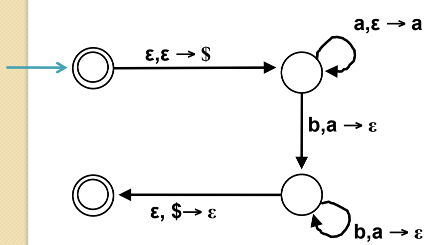
Proof: Exercise.

Is there a state machine for Context Free Languages?

Pushdown Automaton

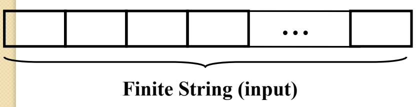
- A Nondeterministic Finite Automaton (NFA) with a Stack.
- Can be used to represent Context Free Languages.
- The parsers generated by some compiler-compilers are implemented by a Pushdown Automaton.

Example



INPUT TAPE

Where the input lives



The input tape is read in one direction, left to right.

STACK

- Store for letters
  - Serves as a memory
- Two Operations
- **Push**
  - Puts a letter at the top of the stack
- **Pop**
  - Takes a letter off the top of the stack.

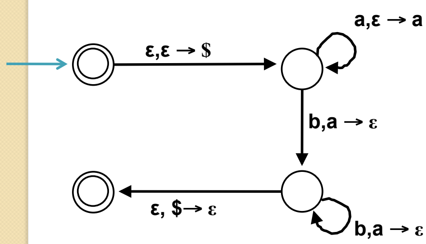
Transitions

a, b → c

*which means when the machine is reading an **a**, if there is a **b** on top of the stack it is popped, and **c** is pushed onto the stack.*

- If **a** is ε then no symbol is read from the tape.
- If **b** is ε then no symbol is popped from the stack.
- If **c** is ε then no symbol is pushed onto the stack.

Example



A **Pushdown Automaton** is a collection of:

- An alphabet of possible input letters
- An alphabet of possible stack letters
- An **INPUT TAPE** and a **STACK**
- A finite set of states
  - One called the **Start State**
  - Some (maybe none) called **Final States**
- A set of transitions between states

a, b → c

which means when the machine is reading an **a**, if there is a **b** on top of the stack, it is replaced by **c**.

Definitions

- A string is **accepted** by a PDA
  - if there exists at **least one** path through the PDA for this string that ends in a Final State
- A string is **rejected** by a PDA
  - if for **all** paths through the PDA for this string the PDA either crashes or ends in a non-Final State
- The set of strings accepted by the PDA is called the **language accepted** by the PDA.

$\{a^n b^n : n \geq 0\}$

$\{\varepsilon,\ ab,\ aabb,\ aaabbb,\ \ldots\}$

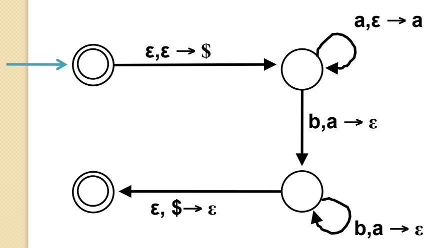
Using the Pumping Lemma we showed that this language was non-regular.

Consider:

$$S \rightarrow aSb \mid \varepsilon$$

So it is a Context-Free Language.

Example



Regular Languages ⊂ PDA

Exercise:

- What do you have to do to restrict a PDA so that it is just an NFA?

An NFA is a special case of a PDA.

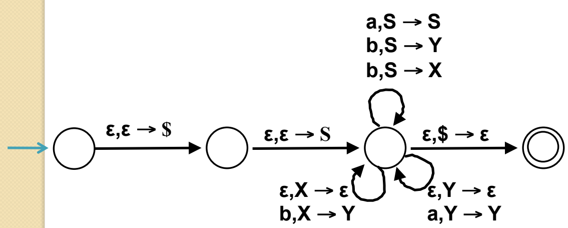
So

{ regular languages } ⊂

{ languages recognised by a PDA}

Regular Languages ⊂ PDA

$S \rightarrow aS \mid bY \mid bX \qquad X \rightarrow bY \mid \varepsilon \qquad Y \rightarrow aY \mid \varepsilon$



CFL = PDA

… or, to be more precise:

{CFLs} = { languages recognised by a PDA }

We will show

1. { CFLs } ⊆ { languages recognised by a PDA }
2. { languages recognised by a PDA } ⊆ { CFLs }

CFL ⊆ PDA

1. **Theorem**

   { CFLs } ⊆ { 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:
……
S → peX
X → YcZ
Y → ar
Z → ey
……

Derivation	Input	Stack
S	pearcey	
⇒ pe**X**	**pe**arcey	**X**
⇒ pe**Y**cZ	**pe**arcey	**YcZ**
⇒ pearc**Z**	**pearc**ey	**cZ**
	pearcey	**Z**
⇒ pearcey	**pearcey**	

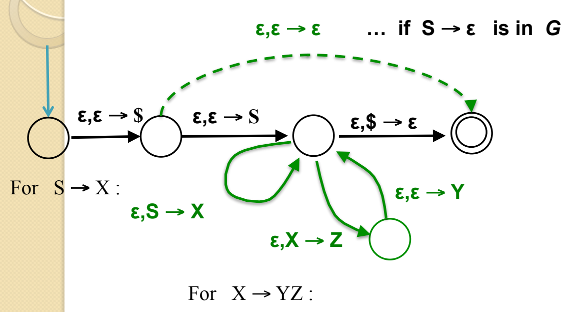
CFL ⊆ PDA

We construct the required PDA as follows.

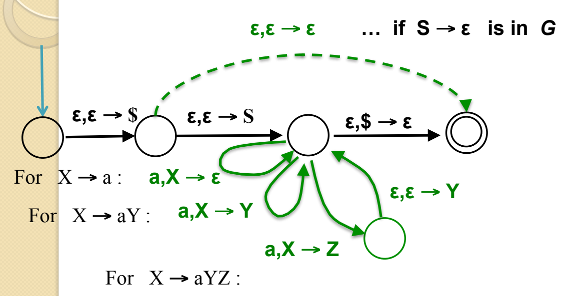
We start with four basic states, then add more states for each production in the CFL …

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

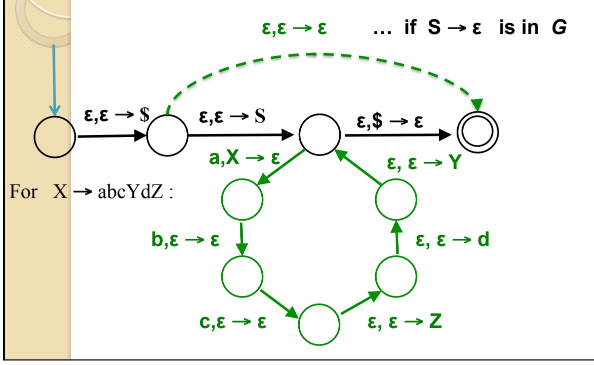
CFL ⊆ PDA



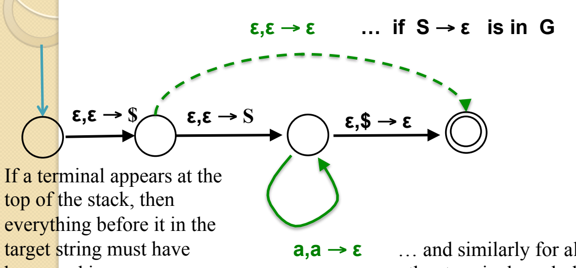
CFL ⊆ PDA



CFL ⊆ PDA



CFL ⊆ PDA



If a terminal appears at the top of the stack, then everything before it in the target string must have been read in.

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

… and similarly for all other terminal symbols.

CFL ⊆ 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.

Now for the other way round …

PDA ⊆ CFL

1. **Theorem**

{ languages recognised by a PDA } ⊆ { CFLs }

Proof ideas:

Let *L* be a langauge recognised by some PDA ***M***.
We need to show that ∃ a CFG *G* that generates ***L***.

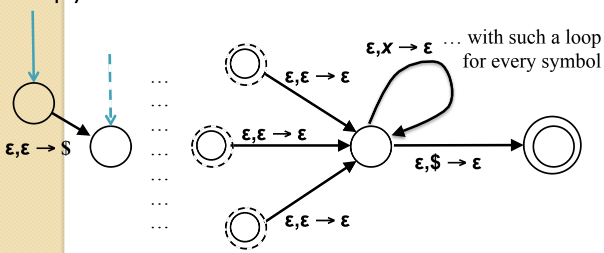
First, we make some simple modifications to ***M***.

Then we give productions that describe certain ways of going through the PDA …

PDA ⊆ CFL

Modifications to M:

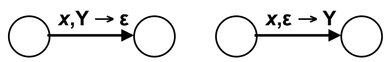
Ensure it has just one Final State, and that the stack is empty when it reaches the Final State.



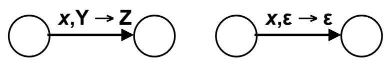
PDA ⊆ CFL

More modifications: suppose each transition of ***M*** *either* pushes *or* pops, but *not both*.

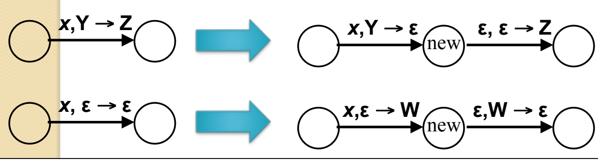
These are ok:



These are *not*:



But this is really no restriction. We can **modify** ***M*** …



PDA ⊆ CFL

A string is accepted by this (modified) ***M*** if one of its paths through ***M***, starting 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 $\mathbf{A}_{pq}$ which is 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 state.

Aim: a grammar such that the strings accepted by ***M*** are precisely those that can be derived from $\mathbf{A}_{st}$.

PDA ⊆ CFL

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

If the computation again has an empty stack at any other state ***r*** on the path, then we can break it into two parts:

- one going from ***p*** to ***r***
  - (starting and ending with empty stack),
- the other going from ***r*** to ***q***
  - (starting and ending with an empty stack).

We model this with the production

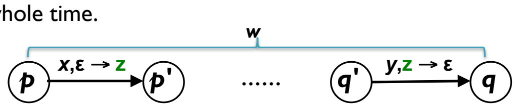
$$\mathbf{A}_{pq} \to \mathbf{A}_{pr}\mathbf{A}_{rq}$$

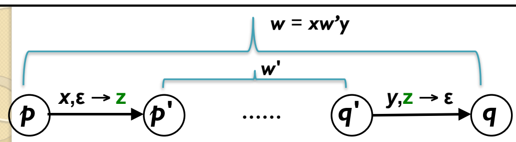
PDA ⊆ CFL

Now suppose the computation never has an empty stack, except at *p* and *q*.

Because it starts and finishes with an empty stack:

- first transtion must push a symbol onto the stack,
- 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.





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 for *w'* 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

$$A_{pq} \to x\, A_{p'q'}\, y$$

PDA ⊆ CFL

Also, for each state *p*, add the production

$$A_{pp} \to \varepsilon$$

Finally, add the production

$$S \to A_{st}$$

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

- Regular Grammars
  - Definition
  - How to define one for a regular language
- Pushdown Automaton
  - Definition and how they work.
  - The languages they recognise are precisely the CFLs

Revision

- Sipser, p107, and Section 2.1 (pp 111-125)