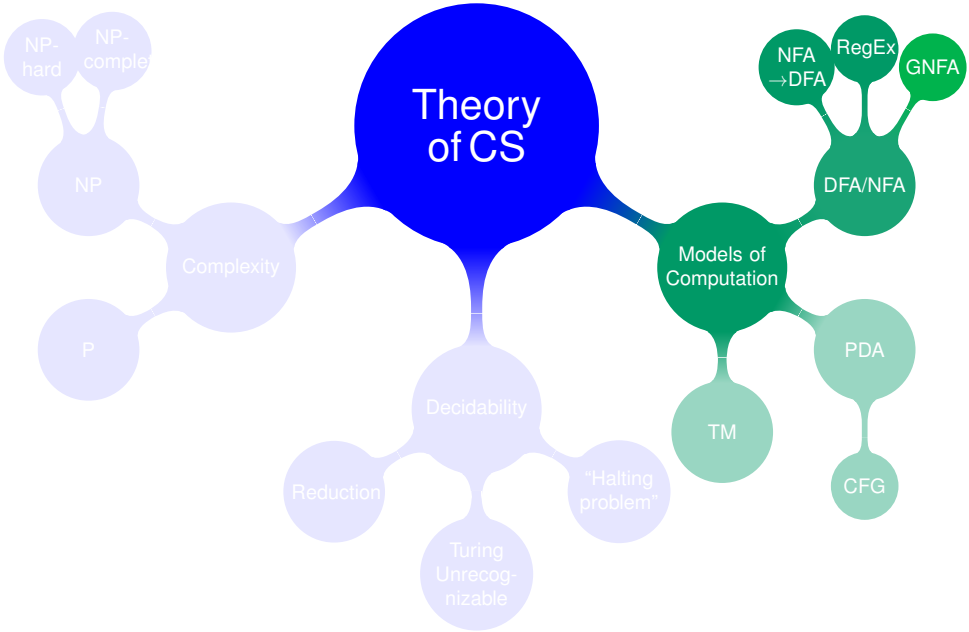
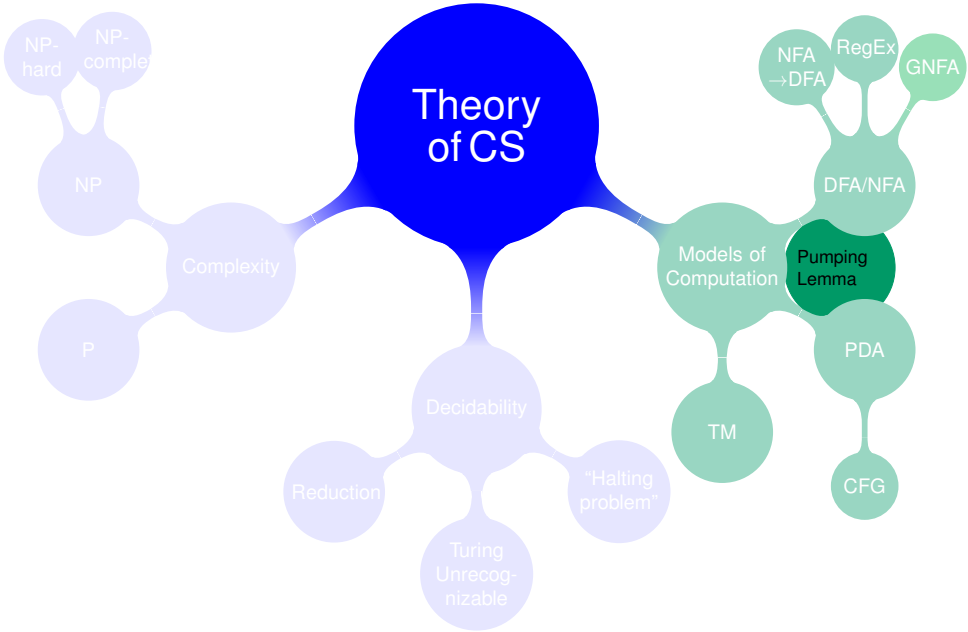
Context-Free Languages (CFLs) & Grammars

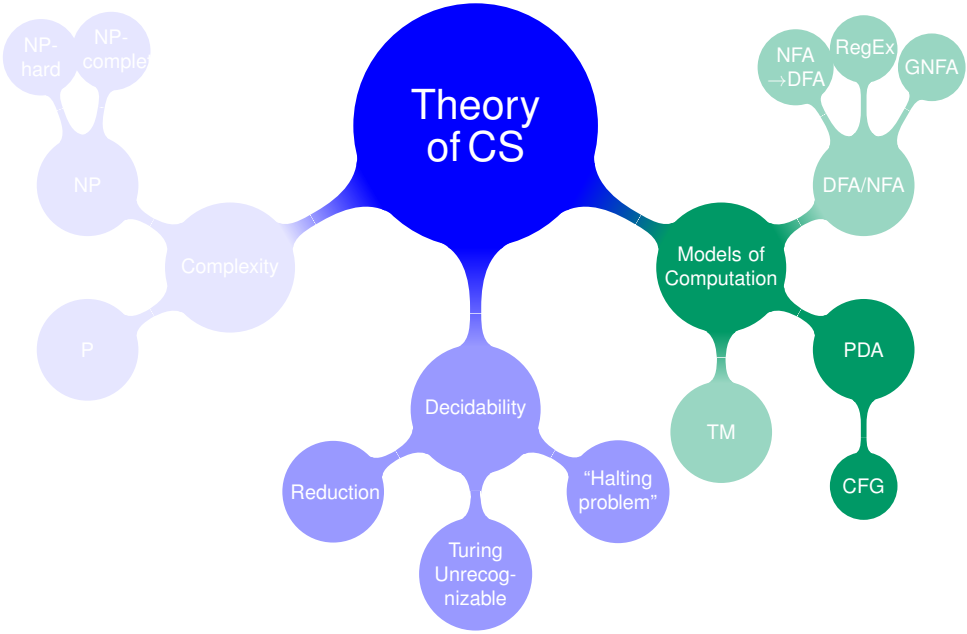
Dr Kamal Bentahar

School of Computing, Electronics and Mathematics
Coventry University

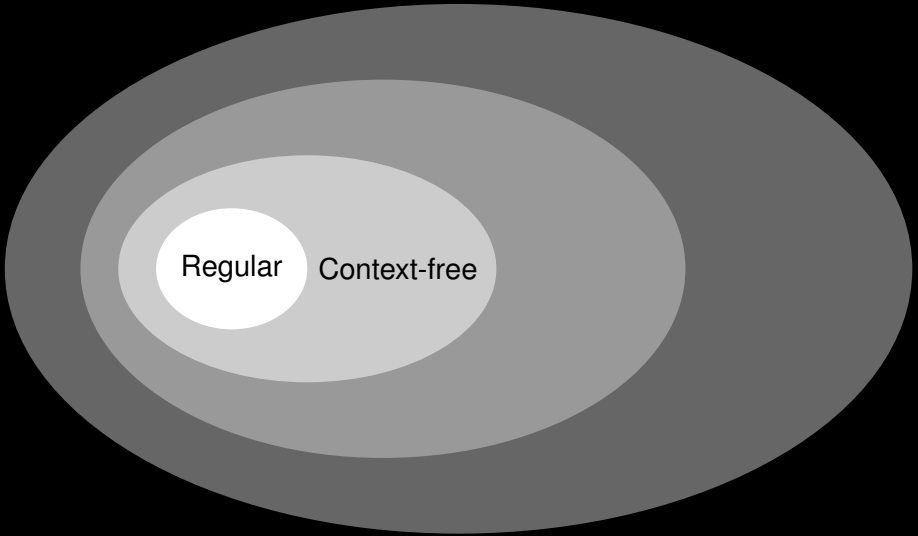
Lecture 5







Language classes



Important concepts developed so far. . .

- Languages
  Alphabet, sets and subsets.
- Finite State Machines (FSMs: DFA/NFA).
  States, transitions, initial and accept states.
- Nondeterminism.
- Equivalence of models.
- Accepters vs generators.
  FSM vs RegEx/Grammar.

Making NFAs more powerful. . .

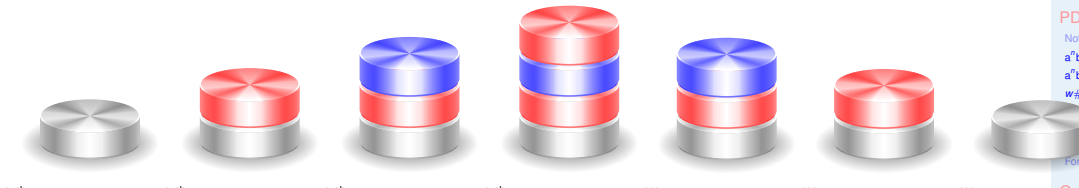
We have seen that $\{a^n b^n \mid n \geq 0\}$ is **not regular**. (Pumping Lemma)
What can we add to NFAs to enable them to recognize this language?

Idea!

- **Count** how many symbols have been seen.
- We can use a **stack**!
- What is a “stack”?
  - LIFO memory (Last-In First Out).
  - Can **push** & **pop**.
  - **Infinite** (structured) memory!

Counting using a stack

Remembering **patterns** using a stack

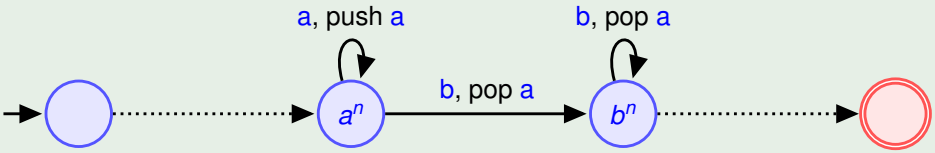


Push-Down Automata (PDAs)

- Largely the same as NFAs but with the addition of stack memory
- More powerful than NFAs: can recognize some non-regular languages.
- On transition, the machine does not just change state: it also **pushes and/or pops an item on/off the stack**.

Example ($\{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, \ldots\}$)

Push all the a's onto the stack, and then pop one off each time a b is read.
If the stack is empty at the end then the machine accepts.



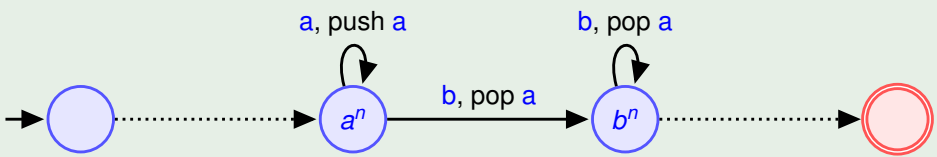
Notation and technicalities. . .

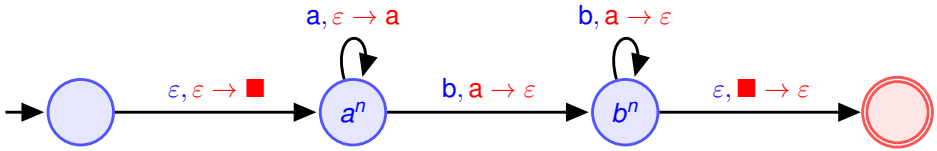
- We label transitions with: a, b → c
  - a: input symbol read from the input string
  - b: symbol popped off the stack
  - c: symbol which replaces it
  - Either a, b or c may be ε

a must be read from the string **and** *b* must be present on the stack.

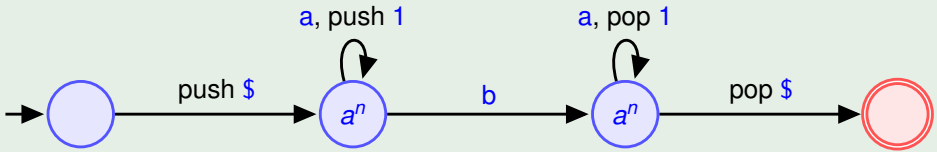
- **There is no special feature for checking if the stack is empty.**
  → Push a delimiting character (e.g. ■ or $) onto the stack at the beginning, then test for this character to see if it is empty.
- **There is no specific way to test for the end of the input.**
  → Have no transitions out of the accept state (i.e. only the last character can reach it).

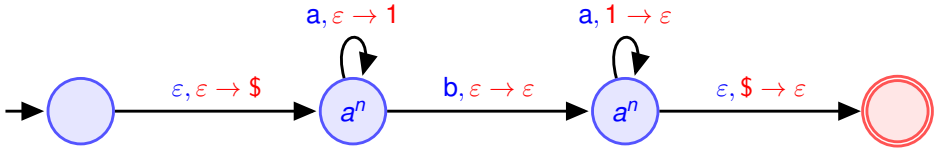
Example ($\{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, \ldots\}$)





Example ($\{a^n b a^n \mid n \geq 0\}$)

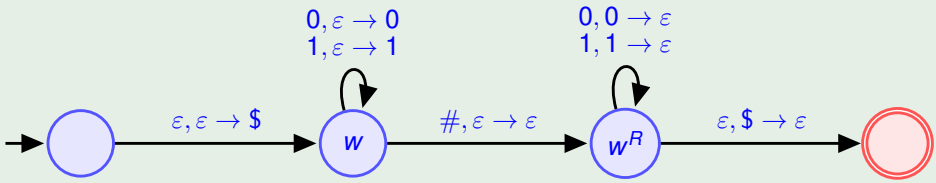




Example ($\{w\#w^R \mid w \in \{0,1\}^*\}$)

Recall: w^R is "w in reverse" (backwards), e.g. if $w = 1011$ then $w^R = 1101$.

Memorise the part before $\#$ then match in **reverse order** to the part after $\#$.



Closures & Nondeterminism

Like NFAs:

- Closed under union, concatenation, and star operations.
- Non-determinism behaviour, and each branch of the computation gets its **own stack**!

Different!

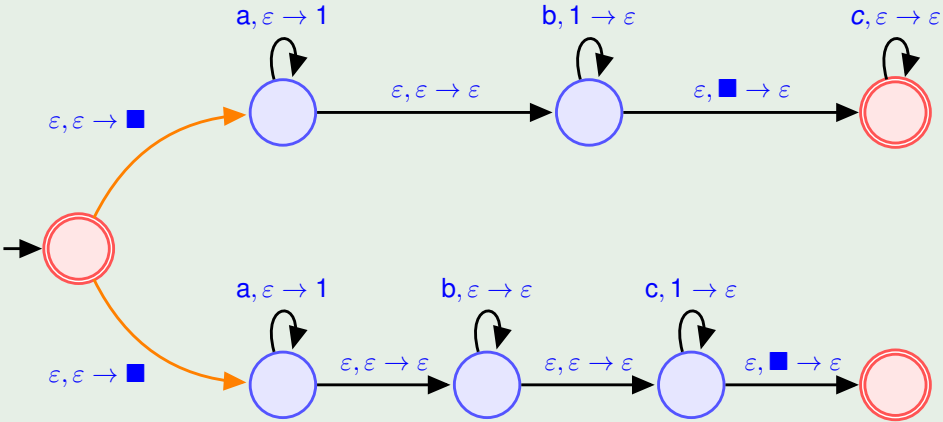
Unlike DFAs vs NFAs, deterministic PDAs are **less powerful** than non-deterministic PDAs (i.e., they recognize less languages).

Some CFLs can only be recognized by non-deterministic PDAs.

Example ($\{a^i b^j c^k \mid i = j \text{ or } i = k\}$)

Rewrite as:

$$\{a^i b^j c^k \mid i = j\} \quad \cup \quad \{a^i b^j c^k \mid i = k\}$$



Formal definition of PDAs & CFLs

Push-Down Automata (PDAs)

A PDA is a 6-tuple $\{Q, \Sigma, \Gamma, \delta, q_{\text{start}}, F\}$ where

- Q is the set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to 2^{Q \times \Gamma_\varepsilon}$ is the transition function
- q_{start} is the start state
- F is the set of accept states

Context-Free Languages (CFLs)

A language is Context-Free **iff** it is recognized by a non-determinsitc PDA.

Grammars

Grammars are defined by **production rules** such as

$$\begin{aligned} A &\to \mathrm{a}\,A\,\mathrm{b} \\ A &\to B \\ B &\to \varepsilon \end{aligned}$$

The rules of the grammar represent possible *replacements*
e.g. $A \to \mathrm{a}\,A\,\mathrm{b}$ means the variable A may be replaced with the string $\mathrm{a}\,A\,\mathrm{b}$.

- **Lower case symbols** a and b are **terminals** (like symbols for NFAs). They constitute the alphabet for the grammar.
- **Upper case symbols** A and B are **variables** (or **non-terminals**). They are to be replaced by terminals or strings.
- A is the **start variable**.

Derivation of strings – generation of a language

Let G be the grammar

$$\begin{aligned} A &\to \mathtt{a}\,A\,\mathtt{b} \\ A &\to B \\ B &\to \varepsilon \end{aligned}$$

From the start variable A, we can iteratively replace variables until we get a string, e.g.

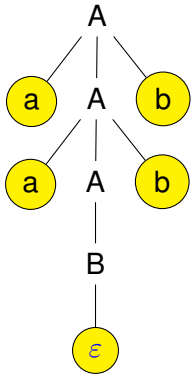
$$A \to \mathtt{a}\,A\,\mathtt{b} \to \mathtt{aa}\,A\,\mathtt{bb} \to \mathtt{aa}\,B\,\mathtt{bb} \to \mathtt{aa}\varepsilon\mathtt{bb} = \mathtt{aabb}$$

- This is called a **derivation** of the string aabb.
- Each step $\alpha \to \beta$ is read: α **yields** β.
- We say: A **derives** aabb, and write: $A \xrightarrow{*} \mathtt{aabb}$.
  ($\xrightarrow{*}$ means: in *zero or more* steps).
- So, **language of the grammar** G is $\{w \in \Sigma^* \mid A \xrightarrow{*} w\}$.

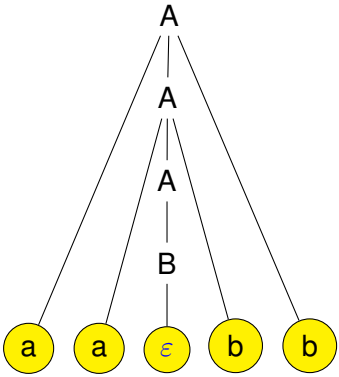
Parse Trees

Diagrammatic way of representing the derivation process.

$$A \to aAb \to aaAbb \to aaBbb \to aa\varepsilon bb = aabb$$



or



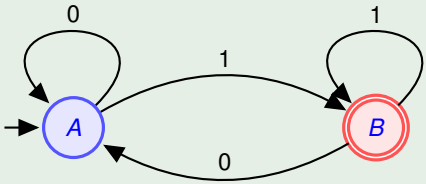
RL $\leftrightarrow$ DFA/NFA/RegEx $\leftrightarrow$ Regular Grammar

- Make a variable V_i for each state q_i
- Add a rule $V_i \to aV_j$ for each transition from q_i to q_j on symbol a.
- Add a rule $V_i \to \varepsilon$ if q_i is an accepting state

Example

Variables: A, B.

$$
\begin{aligned}
A &\to 0A \\
A &\to 1B \\
\\
B &\to 1B \\
B &\to 0A \\
\\
B &\to \varepsilon
\end{aligned}
$$



Notation

To make writing grammars compact, we combine rules starting with the same variable:

$$\left.\begin{array}{lcl} A & \to & 0A \\ A & \to & 1B \\ & & \\ B & \to & 1B \\ B & \to & 0A \\ B & \to & \varepsilon \end{array}\right\} \quad \text{become} \quad \left\{\begin{array}{lcl} A & \to & 0A \mid 1B \\ B & \to & 1B \mid 0A \mid \varepsilon \end{array}\right.$$

Here the | symbol means “or” or “union”.

“Language recognition” and “Language generation”

	Regular Languages	**Context-Free Languages**
Recognizer:	NFA/DFA	PDA
Generator:	RegEx / **Regular** Grammar	**Context-Free** Grammar

Context-Free Grammars (CFGs):

- More powerful at describing languages than RegEx’s.
  Can be used to describe all RLs, as well as some non regular ones
- First used in the study of natural languages.

CFGs: Context-Free Grammars

Context-Free Grammars (CFGs) are defined by **production rules** such as

$$
\begin{aligned}
A &\to \mathrm{a}A\mathrm{b} \\
A &\to B \\
B &\to \varepsilon
\end{aligned}
$$

- Only **one variable to the left** of the arrow.
  $\to$ "context free."

Design of CFGs

Common rules:

- $S \to aS$ generates $\{a, aa, aaa, \ldots\}$
- $S \to aSb$ generates $\{ab, aabb, aaabbb, \ldots\}$
- $S \to AB$: concatenation.
- $S \to A \mid B$: union.
- $S \to SS$ produces $SS, SSS, SSSS, \ldots$.
- $S \to SS \mid \varepsilon$ produces $\varepsilon, SS, SSS, SSSS, \ldots$: star.

Design of CFGs: look for **recursive structures**

Example

Design a CFG to represent the language L over $\Sigma = \{a, b\}$, given by

$$L = \{w \mid w = a^n b a^n, \quad n \geq 0\}$$

We note that

$$a^n b a^n = \begin{cases} a(a^{n-1} b a^{n-1})a & \text{for } n \geq 1 \quad \text{(Recursive case)} \\ b & \text{for } n = 0 \quad \text{(Base case)} \end{cases}$$

CFG:

$$\begin{aligned} A &\rightarrow aAa \\ A &\rightarrow b \end{aligned}$$

Grammars

Chomsky Hierarchy

Grammar	Languages	Automaton	Production rules
Type-0	Recursively Enumerable	Turing Machine (TM)	$\alpha \to \beta$ (no restrictions)
Type-1	Context Sensitive	Linear-bounded TM	$\alpha A \beta \to \alpha \gamma \beta$
Type-2	Context Free	PDA	$A \to \gamma$
Type-3	Regular	NFA	$A \to \mathrm{a}B \mid \mathrm{a}$

Context-Free Grammars (CFGs)

A Context Free Grammar (CFG) is a 4-tuple $\{V, \Sigma, R, S\}$ where

- V is the set of variables
- Σ is the set of terminals
- R is the set of production rules
- S is the start variable

Equivalence of PDAs and CFGs
→ Context-Free Languages (CFLs)

Context-Free Languages (CFLs)

The class of languages recognized by PDAs is the same as the one generated by CFGs.

- Can be shown by providing methods to convert one to the other – refer to the textbook for a demonstration.
- We call this class: **Context-Free Languages** (CFLs).

Pumping Lemma for CFLs (Not examinable)

For the curious – not examinable!

Pumping Lemma for CFLs

If L is a CFL then there is a number p where: if w is any string in L of length at least p then w may be divided into **five** pieces $w = uxyzv$ satisfying the conditions

1. for each $k \geq 0$: $ux^k yz^k v \in L$
2. $|xz| > 0$
3. $|xyz| \leq p$