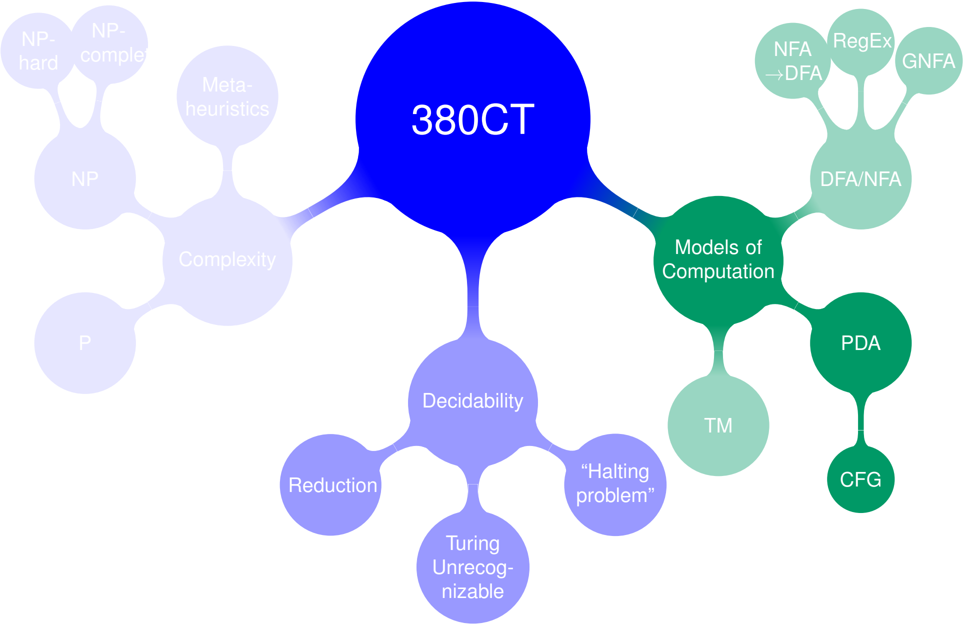
Context-Free Languages (CFLs)

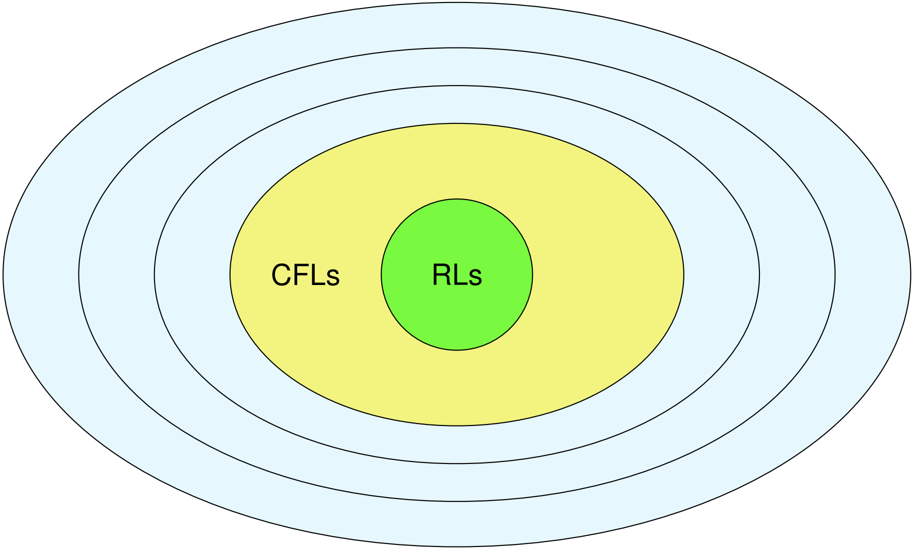
Dr Kamal Bentahar

School of Engineering, Environment and Computing
Coventry University

31/10/2016


380CT
Complexity
NP
NP-hard
NP-complet
Meta-heuristics
P
Models of Computation
DFA/NFA
NFA →DFA
RegEx
GNFA
TM
PDA
CFG
Decidability
Reduction
Turing Unrecog-nizable
“Halting problem”

Language types. . . Chomsky Hierarchy



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!

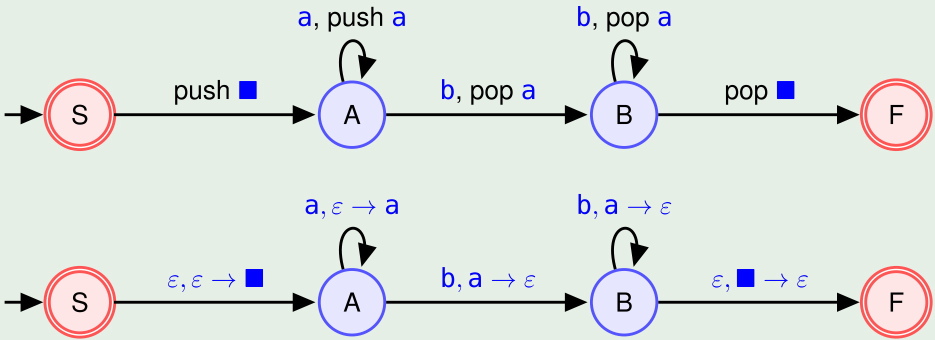
- Need to keep track of how many a's have been seen.
- We can use a **stack**!

Push-Down Automata (PDAs)

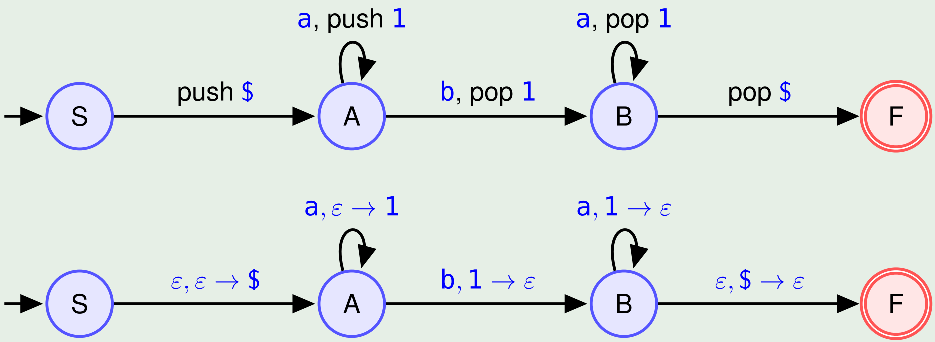
- Largely the same as NFAs but with the addition of a stack memory
  - LIFO memory (Last-In First Out).
  - Can **push** & **pop**.
  - Infinite (structured) memory! (Arbitrarily large n)
- More powerful than NFAs: can recognize some non-regular languages.
- On transition, the machine does not just change state: it also pushes and/or pops an item on/off the stack.

Example ($\{w \mid w = \mathrm{a}^n\mathrm{b}^n \text{ for } n \geq 0\}$)

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.



Example ($\{w \mid w = a^n b a^n, \quad n \geq 0\}$)



Notation and technicalities...

- We label transitions with: $a, b \rightarrow c$
  - 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
- PDAs have no special feature for checking if the stack is empty. Counter this by pushing a delimiting character (in this case: $\blacksquare$ or $\$$) onto the stack at the beginning, then test for this character to see if it is empty.
- Also there is no specific way to test for the end of the input. This is dealt with in the example by having no transitions out of the accept state (i.e. only the last character can reach it).

Closures & Nondeterminism

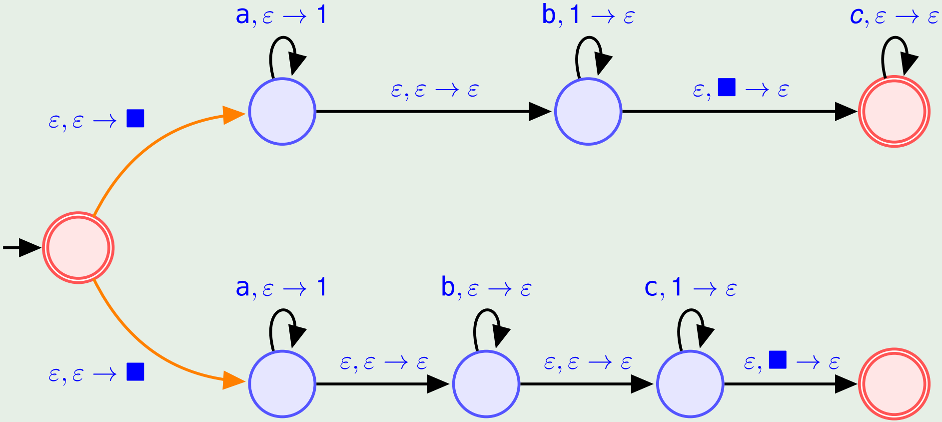
- Like NFAs, closed under union, concatenation, and star operations.
- Like NFAs, non-determinism, and each branch of the computation gets its **own stack**!
- Unlike DFAs vs NFAs, deterministic PDAs are **less powerful** than non-deterministic PDAs. (i.e., they recognize less languages)

Some CFLs can only be recognized by PDAs using nondeterminism.

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

Rewrite as:

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



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.

“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

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

The rules of the grammar represent possible *replacements*
e.g. $A \rightarrow \mathrm{a}A\mathrm{b}$ 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

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

Commencing with the start variable, these replacements can be used iteratively to produce strings e.g.

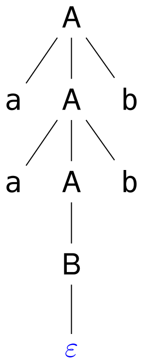
$$A \rightarrow \mathrm{a}A\mathrm{b} \rightarrow \mathrm{aa}A\mathrm{bb} \rightarrow \mathrm{aa}B\mathrm{bb} \rightarrow \mathrm{aa}\varepsilon\mathrm{bb} = \mathrm{aabb}$$

This is called a **derivation** of the string aabb.

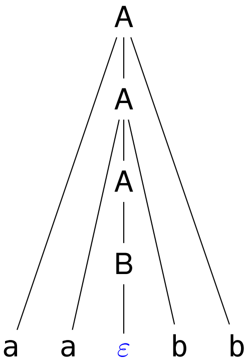
Parse Trees

Diagrammatic way of representing the derivation process.

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



or



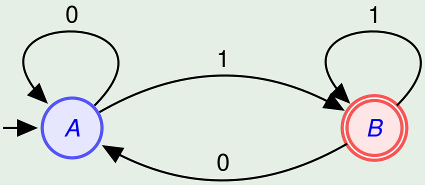
RL $\leftrightarrow$ DFA/NFA/RegEx $\leftrightarrow$ Regular Grammar

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

Example

A, B

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



CFGs – notation

To make writing CFGs more compact, we can combine rules starting with the same variable:

$$\begin{cases} A \to \mathrm{a}A\mathrm{b} \\ A \to B \\ B \to \varepsilon \end{cases} \quad \text{becomes} \quad \begin{cases} A \to \mathrm{a}A\mathrm{b} \mid B \\ B \to \varepsilon \end{cases}$$

Here the $\mid$ symbol means "or" or "union".
We have combined the first two rules into a single line.

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

$$a^n b a^n = \begin{cases} a(a^{n-1} b a^{n-1})a & \text{for } n \geq 1 \\ b & \text{for } n = 0 \end{cases} \qquad (\text{Note that } a^{n-1} b a^{n-1} \in L)$$

CFG:

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

Equivalence of PDAs and CFGs $\rightarrow$ Context-Free Languages (CFLs)

- 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

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 $k \geq 0$: $ux^kyz^kv \in L$
2. $|xz| > 0$
3. $|xyz| \leq p$

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	$A \to aB \mid a$

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