CITS2211 Discrete Structures
Non-regular languages and push-down automata

Unit coordinator: Débora Corrêa
Lecturer: Matt Ryan

Highlights

We have seen that FSMs are surprisingly powerful

But we also saw some languages FSMs can not recognise

Now we will learn a useful theorem for testing whether a language is regular or not: **the pumping lemma for regular languages**

We will study a new type of automata: **pushdown automata**

And a new class of languages: **context-free languages**

Reading

Introduction to the Theory of Computation by Michael Sipser

Chapter 1: Regular Languages
Section 1.4 The pumping lemma for regular languages

Chapter 2: Context-free languages
Section 2.2 Pushdown automata

Non-regular languages

Lecture Outline

1. Non-regular languages
2. The pumping lemma for regular languages
3. Push-down automata
4. Context-free languages

Non-regular languages

Q: Are all languages regular?
A: No

We saw in the lectures last week that we can use a *diagonalization argument* to show that there are non-regular languages. (We made a non-regular language by using an enumeration of all the regular languages.)

Thus, by Kleene's theorem, there are languages which can not be recognized by finite state machines.

Pumping lemma

The pumping lemma

We now turn to the problem of determining when a language is non-regular, and hence *cannot* be recognized by a finite state machine.

The main tool for this is a result called the *pumping lemma* which can be proved using the Pigeonhole Principle we studied a few weeks ago.

The pumping lemma for regular languages was first stated by Y. Bar-Hillel, Micha A. Perles, and Eli Shamir in 1961.

Theorem: Pumping Lemma (for regular languages)

Theorem: If L is a regular language then

$\exists$ *an integer* p *called the* pumping length *of* L *such that*

$\forall$ *words* $w \in L$ *where* $|w| \geq p$

$\exists$ *an expression* $w = xyz$ *where*

1. $\forall\ i \geq 0.\ xy^iz \in L$
2. $|y| \geq 1$
3. $|xy| \leq p$

That is, if L is regular then any sufficiently long word in the language contains a non-empty substring that can be repeated an arbitrary number of times (0,1 or more).

Finite languages

Finite languages satisfy the pumping lemma by setting p to the length of the longest string in the language.

Then it is vacuously true that – for all strings longer than p (of which there are none), the remainder of the proposition follows.

How to use the pumping lemma

The pumping lemma is of the form $R \rightarrow P$. That is IF a language is regular (R) THEN certain properties must hold (P).

- Usage 1: Assume R. From the lemma we also have P. Now derive a contradiction. The contradiction tells us that the original assumption R must be false. This way we can prove that a language is *NOT* regular.
- Usage 2: $R \rightarrow P \leftrightarrow \neg P \rightarrow \neg R$. So if we can show $\neg P$ then by the pumping lemma we can deduce that the language is *NOT* regular.
- Note, that we **can not** say that if the pumping properties hold then the language is regular, because we **do not have** $P \rightarrow R$

Use of pumping lemma

The trickiness of applying pumping lemma is using the $\forall$ and $\exists$ conditions correctly. You need to practice!

To prove that a language is *not* regular, find some sufficiently long string that is in the language, but which cannot be pumped.

The existence of such a string shows that the given language is not regular.

We use an “adversary" game argument [Hopcroft and Ullman]

Your choices in the game correspond to the $\forall$ quantifiers in the statement of the Pumping Lemma (see above) and adversary choices correspond to $\exists$ lines.

Proof of pumping lemma

RTP.

$$\forall L.\ \exists p.\ \forall w.((\ w \in L) \wedge (|w| \geq p))\ \rightarrow$$

$$\exists xyz.\ w = xyz\ \wedge |xy| \leq p\ \wedge |y| \geq 1 \wedge \forall i \geq 0.\ xy^i z \in L$$

Proof. Suppose L is a regular language. Then it can be recognized by a finite state machine M with p states.

Now suppose that $w = c_1 c_2 \ldots c_n$ is a word of length $n \geq p$. Then consider the states that occur when M is run with input string w.

$$s_0 \longrightarrow_{c_1} s_1 \longrightarrow_{c_2} s_2 \cdots \longrightarrow_{c_p} s_p \cdots \longrightarrow_{c_n} s_n$$

By the pigeonhole principle, at least two of the states from $\{s_0, ..., s_p\}$ must be the same so let s_i and s_j be the first two occurrences of the first repeated state.

Pumping Lemma (cont)

Now set

$$x = c_1 c_2 \dots c_i$$

$$y = c_{i+1} c_{i+2} \dots c_j$$

$$z = c_{j+1} \dots c_p \dots c_n$$

Now we can see that y is a string that takes the finite state machine "in a circle" from a state back to itself.

This means that we can now "pump" the input string by repeating this portion as often as possible, and still get a string that is recognized by the machine.

Hence xz, $xyyz$, $xyyyz$ and in general $xy^i z$ are all recognized by M and hence in the language L.

■

Adversary Argument using the Pumping Lemma

1. Select the language L to be proven non-regular
2. The adversary picks the pumping constant p
3. You select some string $w \in L$ (based on your knowledge of p)
4. The adversary breaks w into any x, y, z she wants subject to the constraints $|xy| \leq p \ \wedge |y| \geq 1$
5. You achieve a contradiction to the Pumping Lemma by showing that for any x, y, z chosen by the adversary, $\exists i$ so that $xy^i z \notin L$. Your choice of i may depend on p, x, y, and w.
6. From this it can be concluded that L is not regular.

Adversary Argument Example

Example Show that the language L is not regular where,

$$L = \{w \mid w \text{ has an equal number of 0s and 1s}\}$$

Proof

Suppose L is regular. Let the pumping length be p. Choose $w = 0^p1^p$. Clearly $w \in L$. We will see this is a useful example of w for the proof (not all words in L are useful choices). For any adversary choice of $xyz = w$, both x and y can only contain 0s since $|xy| \leq p$ (constraint 3). Say $x = 0^m$, $y = 0^n$ for some $m + n \leq p$. Now, the pumping lemma states that $xyyz \in L$ but we know $xyyz \notin L$ because $xyyz$ has more 0s than 1s. We have derived a contradiction. Therefore L is not regular. QED

Take Care: Pumping Lemma uses $\rightarrow$ not $\leftrightarrow$

Note that while the pumping lemma states that all regular languages do satisfy the conditions described above, the converse of this statement is not true. A language that satisfies the pumping conditions may still be non-regular.

Example: $L = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = 1 \rightarrow j = k)\}$

a) show that L is not regular
b) show that $w = a^i b^j c^k$ satisfies the pumping lemma conditions (for some i,j,k)
c) explain why parts a) and b) do not contradict the pumping lemma

This question is an exercise in this week's worksheet.

Context-Free Languages

Context-Free Languages

Context-free languages can describe certain features with a recursive structure.

They include all regular languages and more.

What are non-regular languages like?

Context-free languages were first studied for understanding human languages.

For English we have the following loose rule

sentence $\rightarrow$ **noun-phrase verb-phrase**

which we interpret as saying

"A valid sentence consists of a noun-phrase followed by a verb-phrase"

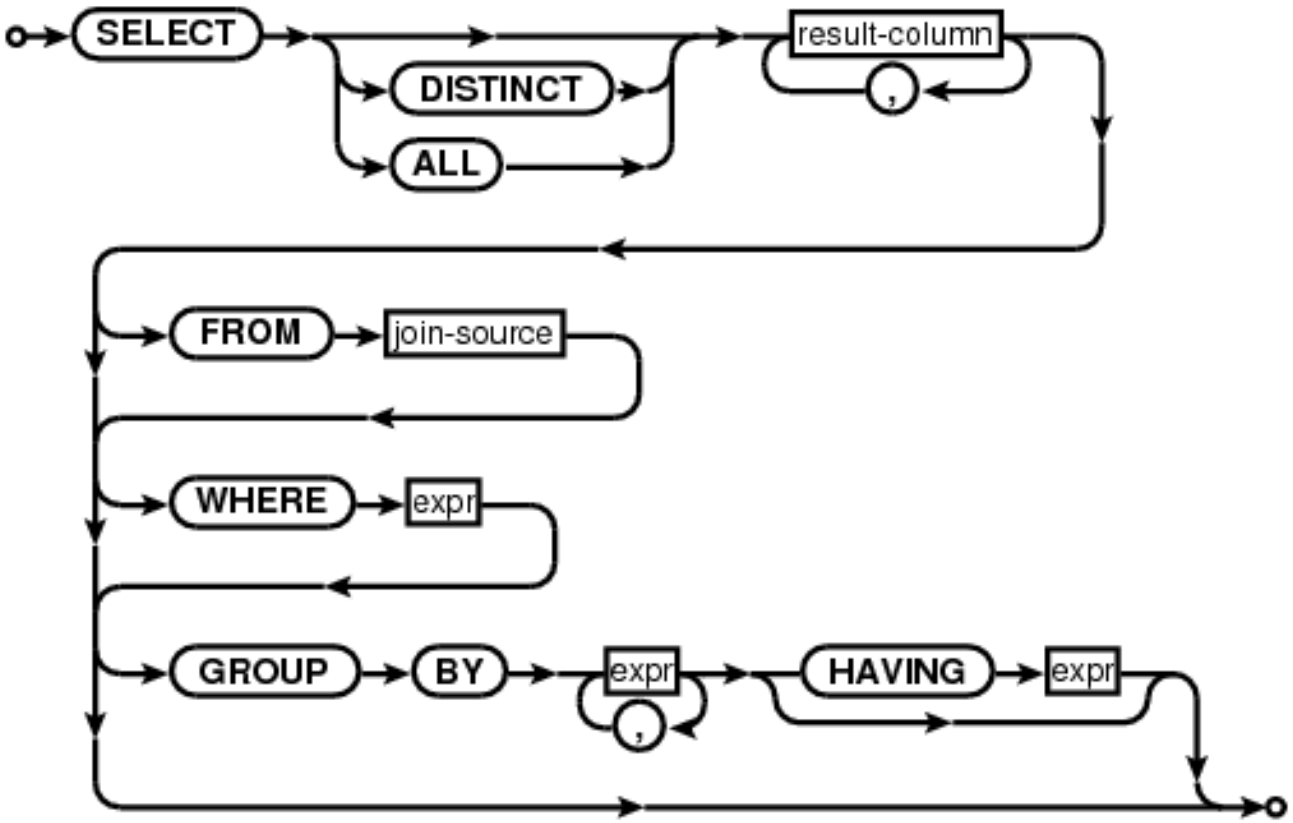
To complete the description, we then need to define **noun-phrase**, **verb-phrase** and so on, which are defined in the same way

noun-phrase $\rightarrow$ **article noun**

verb-phrase $\rightarrow$ **verb adverb**

Context Free languages for Computer Science

An important use of CF languages in Computer Science is the specification and compilation of languages such as Java or SQL.



Context Free languages and Automata

Context-free languages include all the regular languages and many more.

For example, we saw that 0^n1^n was not a regular language, but today we will learn that it is a context-free language.

Context-free languages are precisely the class of languages that can be recognised by **pushdown automata**, which are finite state machines that use a *stack* as a memory device.

The most important formal languages for Computer Science, are probably **context-free** languages, because many computer languages are context free languages (at least in part if not all).

Context Free Grammars

Grammars

Context-free languages can be described and specified using **Context Free Grammars** (CFG); which were invented by the Linguist Noam Chomsky. They were used to help develop most of the early high level programming languages and compiler systems (eg Fortran).

For example, the language $\{0^n1^n | n \geq 0\}$ is generated by the CFG

$$A \rightarrow 0A1 \mid \epsilon$$

Example construction sequence: $A, 0A1, 00A11, 000A111, 000111$

Grammars: the mechanics

- A context free grammar **generates** all the strings of a language.
- A context free grammar is a collection of substitution rules called **productions**.
- Each rule has a left hand side symbol, an arrow and a right hand side.
- Variable symbols are called **non-terminals** and usually represented by a capital letter.
- Other symbols are from the alphabet of the language called **terminals** and usually represented by a lower case letter.
- One symbol is designated the **start variable** usually written S.
- Strings in the language are generated by starting with the start symbol and then replacing non-terminals according to the production rules.

Grammar Example 1

The language of all expressions with balanced brackets is generated by the grammar

$$S \to SS \mid (S) \mid \epsilon$$

Example construction sequence:

$$S, SS, SSS, (S)SS, ((S))SS, (())(S)S, (())()S, (())()(S), (())()()$$

Grammar Example 2

$A \to 0A1$
$A \to B$
$B \to x$

Example derivation: $A, 0A1, 00A11, 00B11, 00x11$

Rules can be written on separate lines (Example 2), or using $|$ to denote a list of rules for the same non-terminal (Example 1).

Context-free grammar definition

Definition: A context-free grammar is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the variables (usually denoted by capital letters)
2. Σ is a finite set, $\Sigma \cap V = \varnothing$, called the terminals (usually denoted by lower case letters or symbols as the alphabet of the language)
3. R is a finite set of rules (ie productions), with each rule of the form $A \to X$ where $A \in V$ and X is a *string* of variables and terminals.
4. $S \in V$ is the start variable

Sipser Definition 2.2, page 104 in the 3rd edition

Pushdown Automata

Idea of Push-Down Automata (PDA)

Context-free languages can be recognised by **automata** called PDAs

PDAs are similar to *non-deterministic* FSMs (NFSMs)

but they have an extra component called a **stack**

The stack provides extra memory, in addition to states

This memory allows a PDA to do unbounded counting, and retrieval of that count, that an NFSM can not. A NFSM ability to count and "know" the count is always bounded by a need to pre-specify the state-memory and the way it reacts to inputs. No such pre-specification is needed in a PDA.

Formal defn of PDAs

Definition: A **pushdown automata (PDA)** is defined to be a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is a finite alphabet of *input symbols*
- Γ is the finite *stack alphabet*
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the *transition function*
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is a set of accepting states (F may be the empty set)

PDA transitions

Note this definition is nearly the same as for NFSMs with the exception of the transition function and the addition of a stack alphabet Γ.

As well as changing state for a given input, a PDA may

- **read and pop** a symbol from the stack
- **push** a symbol onto the top of the stack.

Writing PDA transitions

Transitions are written $a, b \to c$ where a is an input symbol

If the machine sees input a and has b on top of the stack, pop b, and replace it with c, in order to then make the transition.

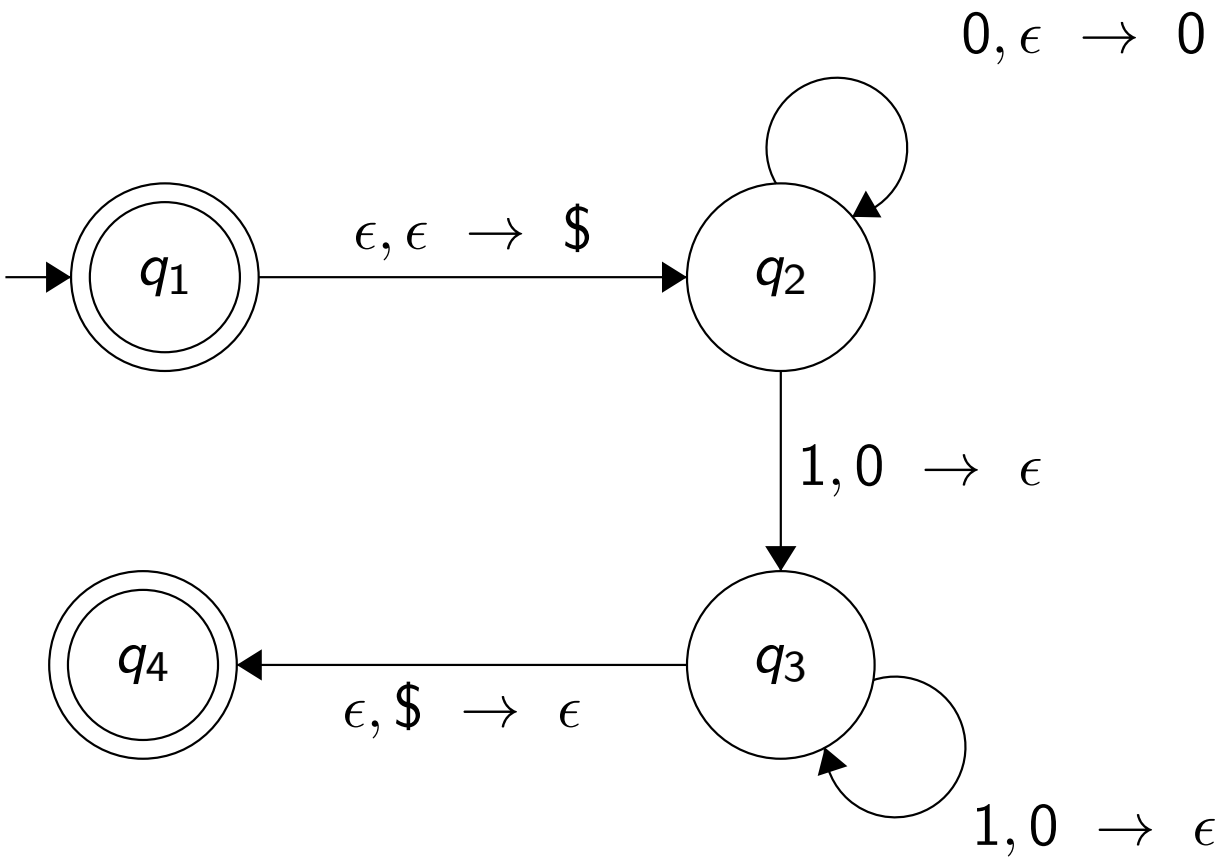
In other words, b is the symbol popped off the stack and c is the symbol pushed onto the stack

If b is ϵ **(the empty symbol)** then make the transition without any pop (read) operation

If c is ϵ then make the transition without any push (write) operation

$ is a special symbol used to denote the bottom of the stack: it means the stack is empty

PDA Example for 0^n1^n



Reading PDA transitions

$\epsilon, \epsilon \rightarrow \$$ Given no input and nothing to pop, put the empty stack symbol onto the stack. All PDAs start with this transition

$0, \epsilon \rightarrow 0$ On seeing an input 0, push a 0 onto the stack. Do not pop anything from the stack

$1, 0 \rightarrow \epsilon$ On seeing input 1 and a 0 on the stack, pop the 0 from the stack and do not push anything on. This step pairs off all the 1s with the previously stored 0s.

$\epsilon, \$ \rightarrow \epsilon$ On seeing no input and an empty stack, accept the string since we must now have seen the same number of 1s as 0s

Palindrome language

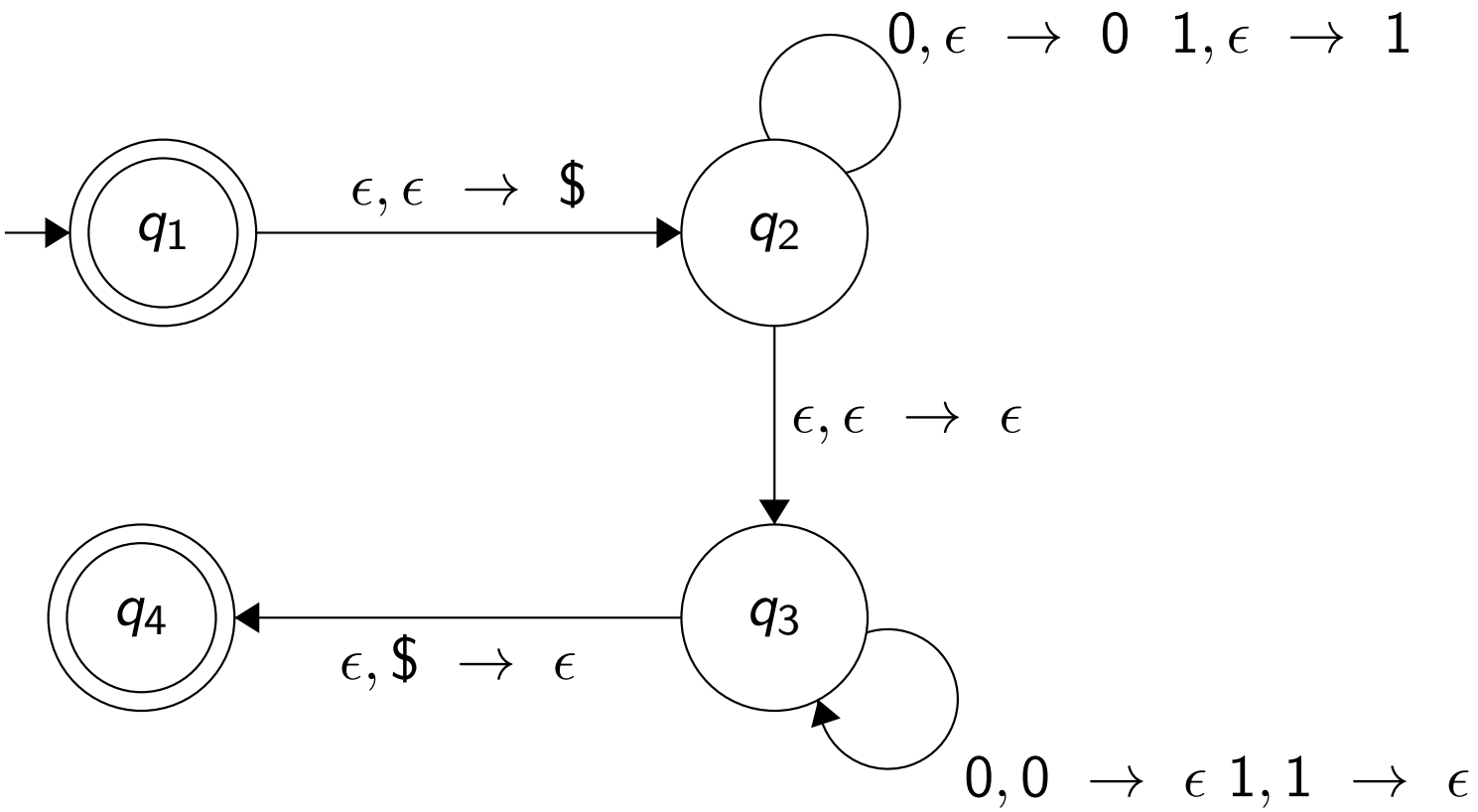
Design a PDA to recognize the language $\{ ww^R \mid w \in \{0,1\}^* \}$. w is any binary string, w^R means w written backwards. So for example, 001110011100 is in the language.

Here we will use non-determinism to “guess" when the middle of the string has been reached.

Approach:

1. Push all the symbols read onto the stack
2. Guess you have reached the middle of the word
3. Then pop elements off the stack if they match the next input symbol
4. Accept the string if every popped symbol matches the input, and the stack empties at the same time the end of the input is reached.
5. Reject otherwise

Palindrome language PDA



Designing PDAs

- See Sipser Lemma 2.21 for details (p117 in 3rd ed)
- Idea: PDA accepts input w if grammar G generates it, by following the derivations
- Use non-determinism to allow for choice of productions.
- Push the start symbol S onto the stack $\epsilon, \epsilon \rightarrow S$
- If top of stack is a non-terminal (A) then non-deterministically choose any production and substitute A by the rule.
- Since the rules generate more symbols we need a string of pushes. No inputs are consumed at this stage.
- If top of stack is a terminal symbol (0 or 1) then check whether it matches the next symbol in the input string
- If they don't match then go to a non-accept state, if they do match then continue

Designing PDAs (cont)

(See the worksheet questions for examples)

How to show a language is context-free

Theorem: PDAs are equivalent in power to context-free grammars.

This is a useful result because it gives 2 options for proving that a language is CF

1. specify a PDA for the language
2. specify a CF grammar for language

Backus-Naur form (for information)

Context-free grammars related to computer languages are often given in a special shorthand notation known as Backus-Naur form.

```
<identifier> ::  = <letter> | <identifer> <letter> |
             <identifier> <digit>
<letter> ::  = a | b | c | ...  | z
<digit> ::  = 0 | 1 | ...  | 9
```

In BNF, the non-terminals are identifed by the angle brackets, and productions with the same left-hand side are combined into a single statement with the OR symbol.

Summary

- The class of context Free languages encapsulates the class of regular languages (All regular languages are context free languages, but not all context free languages are regular languages).
- Context free languages are described by context free grammars and recognised by PDAs.
- CFGs and PDAs are therefore equivalent in their descriptive power.
- Regular Pumping lemma can be used to test if a language is regular.