Definition: A context-free grammar $(V, A, <s>, P)$ is called a regular grammar is every production rule in P is of one of the three forms:

(i) <A>→ b<B>

(ii) <A>→ b

(iii) <A>→ ε

where <A> and <B> are nonterminals, b is a terminal, and ε is the empty word. A regular grammar is said to be in normal form if all its production rules are of types (i) and (iii).

Remark: In the literature, you often see this definition labelled left-regular grammar as opposed to right-regular grammar, where the production rules of types 1 have the form <A>→<B>b, (**i.e.** the terminal is one the right of the nonterminal). This distinction is not really important as long as we stick to one type throughout since both left-regular grammars and right-regular grammars generate regular languages.

Lemma: Any language generated by a regular grammar may be generated by a regular grammar in normal form.

Proof: Let <A>→b be a rule of type (ii). Replace it by two rules: <A>→b<F> and <F>→ ε, where <F> is a new nonterminal. Add <F> to the set V. We do the same for every rule of type (ii) obtaining a bigger set V, but now our production rules are only of type (i) and (iii) and we are generating the same language.

qed

Example: Recall the regular language $L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$. We can generate it from the regular grammar in normal gorm given by production rules:

1. <s>→ 0<A>
2. <A>→ 0<A>
3. <A>→ ε
4. <s>→ ε

5. <A>$\to$ 1<B>
6. <B>$\to$ 1<B>
7. <s>$\to$ 1 <B>
8. <B>$\to \varepsilon$

Rules (1), (2), (5), (6), (7) are of type (i), where rules (3), (4) and (8) are of types (iii).
(1) and (3) gives 0. (1), (2) applied $m-1$ times and (3) gives 0^m for $m \geq 2$.
(7) and (8) give 1. (7), (6) applied $n-1$ times and (8) give 1^n for $n \geq 2$.
(1), (5) and (8) give 01. (1), (5), (6) applied $n-1$ times and (8) gives 01^n for $n \geq 2$.
(1), (2) applied $m-1$ times, (5) and (8) gives 0^m1 for $m \geq 2$.
(1), (2) applied $m-1$ times, (5), (6) applied $n-1$ times, and (8) gives 0^m1^n for $m \geq 2, n \geq 2$.
Rule (4) gives the empty word $\varepsilon = 0^0 1^0$.

Q: Why does a regular grammar yield a regular language, **i.e.** one recognised by a finite state acceptor?

A: Not obvious from the definition, but we can construct the finite state acceptor from the regular grammar as follows: our regular grammar is given by $(V, A, <s>, P)$. Want a finite state acceptor (S, A, i, t, F). Immediately, we see the alphabet A is the same and $i =$<s>. This gives us the idea of associating to every nonterminal symbol in $V \backslash A$ a state. <s>$\in V \backslash A$, so that's good. Next we ask:

Q: Is it sufficient for $S = V \backslash A$?

A: No! Our set F of finishing/accepting states should be nonempty. So we add an element $\{f\}$ to $V \backslash A$, where our acceptor will end up when a word in our language. Thus, $S = (V \backslash A) \cup \{f\}$ and $F = \{f\}$. $F \subseteq S$ as needed.

Q: How do we define t?

A: Use the production rules in P! For every rule of type (i), which is of the form <A>$\to$ b<B> set t(<A>,b) =<B>. This works out well because our nonterminals <A> and <B> are states of the acceptor and the terminal $b \in A$ so t takes an element of $S \times A$ to an element of S as needed. Now look at production rules of type (ii), <A>$\to b$ and of types (iii), <A>$\to \varepsilon$. Those are applied when we finish constructing a word w in our language L, **i.e.** at the very last step, so our acceptor should end up in the finishing state f whenever a production rule of type (ii) or (iii) is applied. Write a production rule of type (ii) or (iii) as <A>$\to w$, then we can set $t(<A>, w) = f$. We have finished constructing t as well. Technically, $t : S \times (A \cup \{\varepsilon\}) \to S$ instead of $t : S \times A \to S$, but we can easily fix the transition function t by combining the last two transitions for each accepted word.

Remark: The same general principles as we used above allow us to go from a finite state acceptor to a regular grammar. This gives us the following theorem:

Theorem: A language L is regular $\Leftrightarrow$ L is recognised by a finite state acceptor $\Leftrightarrow$ L is generated by a regular grammar.

Regular expressions

<u>Task</u> Understand another equivalent way of characterizing regular languages due to Kleene in the 1950's.

<u>Def</u>: Let A be an alphabet.

1. $\emptyset$, ε, and all elements of A are regular expressions;
2. If w and w' are regular expressions, then $w \circ w'$, $w \cup w'$, and w^* are regular expressions.

<u>Remark</u> This definition is an inductive one.

<u>NB</u> It is important not to confuse the regular expressions $\emptyset$ and ε. The expression ε represents the language containing a single string, namely ε the empty string, whereas $\emptyset$ represents the language that doesn't contain any strings. Recall that a language L is any subset of

$$A^* = \bigcup_{n=0}^{\infty} A^n = \underbrace{A^0}_{\substack{= \{\varepsilon\} \\ \text{set of words} \\ \text{of length } 0}} \cup \underbrace{A^1}_{\substack{\text{set of words} \\ \text{of length } 1}} \cup \underbrace{A^2}_{\substack{\text{set of words} \\ \text{of length } 2}} \cup \dots$$

<u>Precedence order of operations if parentheses aren't present:</u>

First $*$, then $\circ$ (concatenation), then $\cup$ (union).

<u>Examples</u>

(1) $A = \{0, 1\}$

$$1^* \circ 0 = \{ w \in A^* \mid w = 1^n 0 \text{ for } n \in \mathbb{N}, n \geq 0 \} = \{0, 10, 110, 1110, \dots\}$$

1^*0 we can omit the concatenation symbol

(2) $A = \{0, 1\}$ ...

(3) $A = \{0, 1\}$ $(A \circ A)^* = \{ w \in A^* \mid w \text{ is a word of even length} \}$

Recall $L^* = \bigcup_{n=0}^{\infty} L^n$ where $L^0 = \{\varepsilon\}$

$L^1 = L$ and inductively

$L^n = L \circ L^{n-1}$ Here $L = A \circ A = \{00, 01, 10, 11\}$

(3)' $(A^* \circ A^*)^* = A^*$

(4) $A = \{0, 1\}$ $(0 \cup \varepsilon) \circ (1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$

(5) $\varepsilon^* = \{\varepsilon\}$

(6) $\emptyset^* = \{\varepsilon\}$ The star operation concatenates any number of words from the language. If the language is empty, then the star operation can only put together 0 words, which yields only the empty word.

<u>Use of regular expressions in programming</u>

→ design of compilers for programming languages

Elemental objects in a programming language, which are called tokens (for example variables names and constants) can be described w/ regular expressions ⇝ get the syntax of a programming language this way. There exists an algorithm for recognizing regular expressions that has been implemented ⇒ an automatic system generates the lexical analyzer that checks the input in a compiler.

→ eliminate redundancy in programming

The same regular expression can be generated in more than one way (obvious from the definition of a regular expression) ⇒ there exist an equivalence relation on regular expressions and algorithms that check when two regular expressions are equivalent