8 Formal Languages

Task: Use what we learned about structures in abstract algebra in order to make sense of formal languages and grammars.
Let A be a finite set. When studying formal languages, we call A an alphabet and the elements of A letters.

Examples:

1. $A = \{0, 1\}$ binary digits
2. $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ decimal digits
3. $A =$ letters of the English alphabet

Definition: $\forall n \in \mathbb{N}^*$, we define a word of length n in the alphabet A as being any string of the form $a_1 a_2 \cdots a_n$ *s.t.* $a_i \in A \quad \forall i, 1 \leq i \leq n$. Let A^n be the set of all words of length n over the alphabet A.

Remark: There is a one-to-one correspondence between the string $a_1 a_2 \cdots a_n$ and the ordered n-tuple $(a_1, a_2, ..., a_n) \in A^n = \underbrace{A \times ... \times A}_{n\ times}$, the Cartesian product of n copies of A.

Definition: Let $A^+ = \bigcup\limits_{n=1}^{\infty} A^n = A^1 \cup A^2 \cup A^3 \cup ....$ A^+ is the set of all words of positive length over the alphabet A.

Examples:

1. $A = \{0, 1\}, A^+$ is the set of all binary strings of finite length that is at least one, **i.e.** $0, 1, 01, 10, 00, 11$, etc.
2. If $A =$ letters of the English alphabet, then A^+ consists of all non-empty strings of finite length of letters from the English alphabet.

It is useful to also have the empty word ε in our set of strings. ε has length 0. Define $A^0 = \{\varepsilon\}$ and then adjoin the empty word ε to A^+. We get $A^* = \{\varepsilon\} \cup A^+ = A^0 \cup \bigcup\limits_{n=1}^{\infty} A^n = \bigcup\limits_{n=0}^{\infty} A^n$.

Notation: We denote the length of a word w by $\mid w \mid$.

Next introduce an operation on A^*.

Definition: Let A be a finite set, and let w_1 and w_2 be words in A^*. $w_1 = a_1 a_2 ... a_m$ and $w_2 = b_1 b_2 ... b_n$. The concatenation of w_1 and w_2 is the word $w_1 \circ w_2$, where $w_1 \circ w_2 = a_1 a_2 ... a_m b_1 b_2 ... b_n$. Sometimes $w_1 \circ w_2$ is denoted as just $w_1 w_2$. Note that $\mid w_1 \circ w_2 \mid = \mid w_1 \mid + \mid w_2 \mid$.
Concatenation of words is:

1. associative
2. NOT commutative if A has more than one element.

Proof of (1): Let $w_1, w_2, w_3 \in A^*$. $w_1 = a_1a_2...a_m$ for some $m \in \mathbb{N}$, $w_2 = b_1b_2...b_n$ for some $n \in \mathbb{N}$, and $w_3 = c_1c_2...c_p$ for some $p \in \mathbb{N}$. $(w_1 \circ w_2) \circ w_3 = w_1 \circ (w_2 \circ w_3) = a_1a_2...a_mb_1b_2...b_nc_1c_2...c_p$.

qed

Proof of (2): Since A has at least two elements, $\exists a, b \in A$ s.t. $a \neq b$.

$a \circ b = ab \neq ba = b \circ a$.

qed

A^* is closed under the operation of concatenation $\Rightarrow$ concatenation is a binary operation on A^* as $\forall w_1, w_2 \in A^*$, $w_1 \circ w_2 \in A^*$.

Theorem Let A be a finite set. $(A^*, \circ)$ is a monoid with identity element ε.

Proof: Concatenation $\circ$ is an associative binary operation on A^* as we showed above. Moreover, $\forall w \in A^*, \varepsilon \circ w = w \circ \varepsilon = w$, so ε is the identity element of A^*.

qed

Definition: Let A be a finite set. A <u>language</u> over A is a subset of A^*. A language L over A is called a <u>formal language</u> is $\exists$ a finite set of rules or algorithm that generates exactly L, **i.e.** all words that belong to L and no other words.

Theorem: Let A be a finite set.

1. If L_1 and L_2 are languages over $A, L_1 \cup L_2$ is a language over A.
2. If L_1 and L_2 are languages over $A, L_1 \cap L_2$ is a language over A.
3. If L_1 and L_2 are languages over A, the concatenation of L_1 and L_2 given by $L_1 \circ L_2 = \{w_1 \circ w_2 \in A^* \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ is a language over A.
4. Let L be a language over A. Define $L^1 = L$ and inductively for any $n \geq 1$, $L^n = L \circ L^{n-1}$. L^n is a language over A. Furthermore, $L^* = \{\varepsilon\} \cup L^1 \cup L^2 \cup L^3 \cup ... = \bigcup\limits_{n=0}^{\infty} L^n$ is a language over A.

Proof: By definition, a language over A is a subset of A^*. Therefore, if $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$, then $L_1 \cup L_2 \subseteq A^*$ and $L_1 \cap L_2 \subseteq A^*$. $\forall w_1 \circ w_2 \in L_1 \circ L_2$, $w_1 \circ w_2 \in A^*$ because $w_1 \in A^n$ for some n and $w_2 \in A^m$ for some m, so $w_1 \circ w_2 \in A^{m+n} \subseteq A^* = \bigcup\limits_{n=0}^{\infty} A^n$.

Applying the same reasoning inductively, we see that $L \subset A^* \Rightarrow L^* \subseteq A^*$ as $L^n \subseteq A^*\ \forall n \geq 0$.

qed

Remark: This theorem gives us a theoretic way of building languages, but we need a practical way. The practical way of building a language is through the notion of a grammar.

Definition: A (formal) grammar is a set of production rules for strings in a language.
To generate a language we use:

1. the set A, which is the alphabet of the language;
2. a start symbol <s>;
3. a set of production rules.

Example: $A = \{0, 1\}$; start symbol <s>; 2 production rules given by:

1. <s> $\rightarrow$ 0<s>1
2. <s> $\rightarrow$ 01

Let's see what we generate: via rule 2, <s> $\rightarrow$ 01, so we get <s> $\Rightarrow$ 01
Via rule 1, <s> $\rightarrow$ 0<s>1, then via rule 2, 0<s>1 $\rightarrow$ 0011. We write the process as <s> $\Rightarrow$ 0<s>1 $\Rightarrow$ 0011.
Via rule 1, <s> $\rightarrow$ 0<s>1, then via rule 1 again 0<s>1 $\rightarrow$ 00<s>11, then via rule 2, 00<s>11 $\rightarrow$ 000111.
We got <s> $\Rightarrow$ 0<s>1 $\Rightarrow$ 00<s>11 $\Rightarrow$ 000111.
The language L we generated thus consists of all strings of the form 0^m1^m (m 0's followed by m 1's) for all $m \geq 1, m \in \mathbb{N}$
We saw 2 types of strings that appeared in this process of generating L:

1. terminals, **i.e.** the elements of A
2. nonterminals, **i.e.** strings that don't consist solely of 0's and 1's such as <s>, 0<s>1, 00<s>11, etc.

The production rules then have the form:

nonterminal $\rightarrow$ word over the alphabet V = {terminals, non-terminals}

<T> $\rightarrow$ w

In our notation, the set of nonterminals is $V \backslash A$, so <T>$\in V \backslash A$ and $w \in V^* = \bigcup_{n=0}^{\infty} V^n$. To the production rule <T>$\rightarrow w$, we can associate the ordered pair $(\text{<T>}, w) \in (V \backslash A) \times V^*$, so the set of production rules, which we will denote by P, is a subset of the Cartesian product $(V \backslash A) \times V^*$.
Grammars come in two flavours:

1. Context-free grammars where we can replace any occurrence of <T> by w if <T>$\rightarrow w$ is one of our production rules.
2. Context-sensitive grammars only certain replacements of <T> by w are allowed, which are governed by the syntax of our language L.

The example we had was of a context-free grammar. We can now finally define context free-grammars.

Definition: A context-free grammar $(V, A, \text{<s>}, P)$ consists of a finite set V, a subset A of V, an element <s> of $V\backslash A$, and a finite subset P of the Cartesian product $V\backslash A \times V^*$.

Notation: $(\underbrace{V}_{set\ of\ terminals\ and\ non\ terminals}, \underbrace{A}_{set\ of\ terminals}, \underbrace{<s>}_{start\ symbol}, \underbrace{P}_{set\ of\ production\ rules})$

Example: $A = \{0, 1\}$; start symbol <s>; 3 production rules given by:

1. <s>$\rightarrow$ 0<s>1
2. <s>$\rightarrow$ 01
3. <s>$\rightarrow$ 0011

We notice here that the word 0011 can be generated in 2 ways in this context free grammar:

By rule 3, <s>$\rightarrow$ 0011 so <s>$\Rightarrow$ 0011

$\vee$

By rule 1, <s>$\rightarrow$ 0<s>1 and by rule 2, 0<s>1 $\rightarrow$ 0011. Therefore, <s>$\Rightarrow$ 0<s>1 $\Rightarrow$ 0011.

Definition: A grammar is called ambiguous if it generates the same string in more than one way.
Obviously, we prefer to have unambiguous grammars, else we waste computer operations.
Next, we need to spell out how words relate to each other in the production of our language via the grammar:

Definition: Let w' and w'' be words over the alphabet V = {terminals, nonterminals}. We say that w' directly yields w'' if $\exists$ words u and v over the alphabet V and a production rule <T>$\rightarrow$ w of the grammar s.t. $w' = u$ <T> v and $w'' = uwv$, where either or both of the words u and v may be the empty word.
In other words, w' directly yields $w'' \Leftrightarrow \exists$ production rule <T>$\rightarrow$ w in the grammar s.t. w" may be obtained from w' by replacing a single occurrence of the nonterminal <T> within the word w' by the word w.

Notation: w' directly yields w'' is denoted by $w' \Rightarrow w''$

Definition: Let w' and w'' be words over the alphabet V. We say that w' yields w'' if either $w' = w''$ or else $\exists$ words $w_0, w_1, ...w_n$ over the alphabet V s.t. $w_0 = w', w_n = w'', w_{i-1} \Rightarrow w_i$ for all $i, 1 \leq i \leq n$. In other words, $w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w_{n-1} \Rightarrow w_n$

Notation: w' yields w'' is denotes by $w' \overset{*}{\Rightarrow} w''$.

Definition: Let $(V, A, \text{<s>}, P)$ be a context-free grammar. The <u>language</u> generated by this grammar is the subset L or A^* defined by $L = \{w \in A^* \mid \text{<s>} \overset{*}{\Rightarrow} w\}$

In other words, the language L generated by a context-free grammar $(V, A, \text{<s>}, P)$ consists of the set of all finite strings consisting entirely of terminals that may be obtained from the start symbol <s> by applying a finite sequence of production rules of the grammar, where the application of one production rule causes <u>one and only one nonterminal</u> to be replaced by the string in V^* corresponding to the right-hand side of the production rule.

8.1 Phrase Structure Grammars

Definition: A phrase structure grammar $(V, A\ \text{<s>}, P)$ consists of a finite set V, a subset A of V, an element <s> of $V \backslash A$, and a finite subset P of $(V^* \backslash A^*) \times V^*$

In a context-free grammar, the set of production rules $P \subset (V \backslash A) \times V^*$.

In a phrase structure grammar, $P \subset (V^* \backslash A^*) \times V^*$. In other words, a production rule in a phrase structure grammar $r \to w$ has a left-hand side r that may contain more than one nonterminal. It is required to contain <u>at least one</u> nonterminal.

For example, if $A = \{0, 1\}$ and <s> is the start symbol in a phrase structure grammar grammar, $0\text{<s>}0\text{<s>}0 \to 00010$ would be an acceptable production rule in a phrase structure grammar but not in a context-free grammar.

The notions $w' \Rightarrow w''$ (w' directly yields w'') and $w' \overset{*}{\Rightarrow} w''$ (w' yields w'') are defined the same way as for context-free grammars except that our production rules may, of course, be more general as we saw in the example above.

Definition: Let $(V, A\ \text{<s>}, P)$ be a phrase structure grammar. The language generated by this grammar is the subset L or A^* defined by $L = \{w \in A^* \mid \text{<s >} \overset{*}{\Rightarrow} w\}$

Remark: The term phrase structure grammars was introduced by Noam Chomsky.

Definition: A language L generated by a context-free grammar is called a <u>context-free language</u>.

We now want to understand a particularly important subclass of context-free languages called regular languages.

8.2 Regular Languages

Task: Understand when a language is regular and how regular languages are produced. Understand basics of automata theory.

History: The term regular language was introduced by Stephen Kleene in 1951. A more descriptive name is finite-state language as we will see that a language is regular $\Leftrightarrow$ it can be recognised by a finite state acceptor, which is a type of finite state machine.
The definition of a regular language is very abstract, though. First, describe what operations the collection of regular languages is closed under:

Let A be a finite set, and let A^* be the set of all words over the alphabet A. The regular languages over the alphabet A constitute the smallest collection C of subsets of A^* satisfying that:

1. All finite subsets of A^* belong to C.
2. C is closed under the Kleene star operation (if $M \subseteq A^*$ is inside C, **i.e.** $M \in C$, then $M^* \in C$)
3. C is closed under concatenation (if $M \subseteq A^*, N \subseteq A^*$ satisfy that $M \in C$ and $N \in C$, then $M \circ N \in C$)
4. C is closed under union (if $M \subseteq A^*$ and $N \subseteq A^*$ satisfy that $M \in C$ and $N \in C$, then $M \cup N \in C$)

Definition: Let A be a finite set, and let A^* be the set of words over the alphabet A. A subset L of A^* is called a regular language over the alphabet A if $L = L_m$ for some finite sequence $L_1, L_2, ..., L_m$ of subsets of A^* with the property that $\forall i, 1 \leq i \leq m, L_i$ satisfies one of the following:

1. L_i is a finite set
2. $L_i = L_j^*$ for some $j, 1 \leq j < i$ (the Kleene star operation applied to one of the previous $L_j's$)
3. $L_i = L_j \circ L_k$ for some j, k such that $1 \leq j, k < i$ (L_i is a concatenation of previous $L_j's$)
4. $L_i = L_j \cup L_k$ for some j, k such that $1 \leq k, j < i$ (L_i is a union of previous $L_j's$)

Example 1: Let $A = \{0, 1\}$. Let $L = \{0^m 1^n \mid m, n \in \mathbb{N} \quad m \geq 0, n \geq 0\}$
L is a regular language. Note that L consists of all strings of first 0's, then 1's or the empty string ε. $0^m 1^n$ stands for m 0's followed by n 1's, **i.e.** $0^m \circ 1^n$. Let us examine $L' = \{0^m \mid m \in \mathbb{N}, m \geq 0\}$ and $L'' = \{1^n \mid n \in \mathbb{N}, n \geq 0\}$

Q: Can we obtain them via operatons listed among 1-4?

A: Yes! Let $M = \{0\} \qquad M \subseteq A \subseteq A^*$ and $M^* = L' = \{0^m \mid m \in \mathbb{N} \quad m \geq 0\}$. Let $N = \{1\} \quad N \subseteq A \subseteq A^*$ and $N^* = L'' = \{1^n \mid n \in \mathbb{N}, n \geq 0\}$. In other words, we can do $L_1 = \{0\}, L_2 = \{1\}, L_3 = L_1^*, L_4 = L_2^*, L_5 = L_3 \circ L_4 = L$. Therefore, L is a regular language.

Example 2 Let $A = \{0, 1\}$. Let $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$. L is the language we used as an example earlier. It turns out L is NOT regular. This language consists of strings of 0's followed by an equal number of strings of 1's. For a machine to decide that the string $0^m 1^m$ is inside the language, it must store the number of 1's, as it examines the number of 0's or vice versa. The number of strings of the type $0^m 1^m$ is not finite, however, so a finite-state machine cannot recognise this language. Heuristically, regular languages correspond to problems that can be solved with finite memory, **i.e.** we only need to remember one of finitely many things. By contrast, nonregular languages correspond to problems that cannot be solved with finite memory.

Theorem: The collection of regular languages L is also closed under the following two operations:

1. Intersection, **i.e.** if L', L'' are regular languages (**i.e.** $L' \in C$ and $L'' \in C$), then their intersection $L' \cap L''$ is a regular language.
2. Complement, **i.e.** if L is a regular language (**i.e.** $L \in C$), then $A^* \backslash L$ is a regular language ($A^* \backslash L \in C$).

Remark: These two properties did not come into the definition of a regular language, but they are true and often quite useful.

8.3 Finite State Acceptors and Automata Theory

Definition: An automaton is a mathematical model of a computing device. Plural of automaton is automata.

Basic idea: Reason about computability without having to worry about the complexity of actual implementation.
It is most reasonable to consider at the beginning just finite states automata, **i.e.** machines with a finite number of internal states. The data is entered discretely, and each datum causes the machine to either remain in the same internal state or else make the transition to some other state determined solely by 2 pieces of information:

1. The current state
2. The input datum

In other words, if S is the finite set of all possible states of our finite state machine, then the transition mapping t that tells us how the internal state of the machine changes on inputting a datum will depend on the current state $s \in S$ and the input datum a, **i.e.** the machine will enter a (potentially) new state $s' = t(s, a)$.

Want to use finite state machines to recognise languages over some alphabet A. Let L be our language.

Input	Output
Word $w = a_1 ... a_n, a_i \in A \, \forall i$	Yes if $w \in L$
	No if $w \notin L$

Since our finite state machine accepts (**i.e.** returns yes to) w if $w \in L$, we call our machine a finite state acceptor. We want to give a rigorous definition of a finite state acceptor. To check $w = a_1 ... a_n$, we input each a_i starting with a_1 and trace how the internal state of the machine changes. S is our set of states of the machine (a finite set). The transition mapping t takes the pair(s, a) and returns the new state $s' = t(s, a)$ (where $s \in S$ and $a \in A$) that the machine has reached so $t : S \times A \to S$.
Some elements and subsets of S are important to understand:

1. The initial state $i \in S$ where the machine starts
2. The subset $F \subseteq S$ of finishing states

It turns out that knowing S, F, i, t, A specifies a finite state acceptor completely.

Definition: A finite state acceptor (S, A, i, t, F) consists of a finite set S of states, a finite set A that is the input alphabet, a starting state $i \in S$, a transition mapping $t : S \times A \to S$, and a set F of finishing states, where $F \subseteq S$.

Definition: Let (S, A, i, t, F) be a finite state acceptor, and let A^* denote the set of words over the input alphabet A. A word $a_1 ... a_n$ of length n over the alphabet A is said to be recognised or accepted by the finite state acceptor if $\exists s_0, s_1, ..., s_n \in S$ states s.t. $s_0 = i$ (the initial state), $s_n \in F$, and $s_i = t(s_{i-1}, a_i) \, \forall i \quad 1 \leq i \leq n$.

Definition: Let (S, A, i, t, F) be a finite state acceptor. A language L over the alphabet A is said to be recognised or accepted by the finite state acceptor if L is the set consisting of all words recognized by the finite state acceptor.

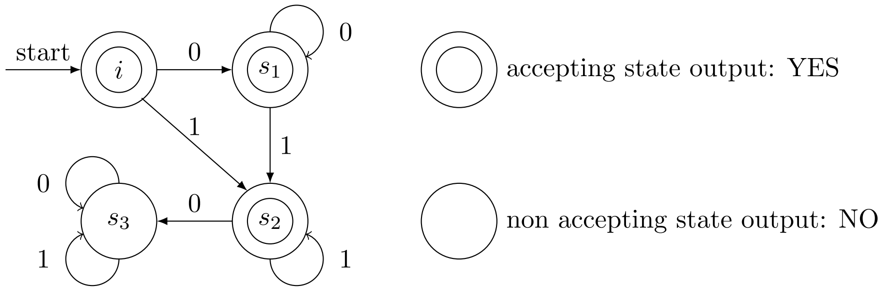
In the definition of a finite state acceptor, t is the transition mapping, which may or may not be a function (hence the careful terminology). This is because finite state acceptors come in 2 flavours:

1. Deterministic: every state has exactly one transition for each possible input, **i.e.** $\forall (s, a) \in S \times A \; \exists! \; t(s, a) \in S$. In other words, the transition mapping is a function.
2. Non-deterministic: an input can lead to one, more than one or no transition for a given state. Some $(s, a) \in S \times A$ might be assigned to more than one element of S, **i.e.** the transition mapping is not a function.

Surprisingly $\exists$ algorithm that transforms a non-deterministic (though more complex one) into a deterministic one using the power set construction. As a result, we have the following theorem:

Theorem: A language L over some alphabet A is a regular language $\Leftrightarrow L$ is recognised by a deterministic finite state acceptor with input alphabet $A \Leftrightarrow L$ is recognised by a non-deterministic finite state acceptor with input alphabet A.

Example: Build a deterministic finite state acceptor for the regular language $L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$



Accepting states in this examples: i, s_1, s_2
Non accepting states: s_3
Start state: i
Here $S = \{i, s_1, s_2, s_3\}$ $F = \{i, s_1, s_2\}$ $A = \{0, 1\}$ $t : S \times A \to S$ $t(i, 0) = s_1$ $t(i, 1) = s_2$ $t(s_1, 0) = s_1$ $t(s_1, 1) = s_2$ $t(s_2, 0) = s_3$ $t(s_2, 1) = s_2$ $t(s_3, 0) = s_3$ $t(s_3, 1) = s_3$
Let's process some strings:

String	ε (empty string)
State (i)	i
Output	YES

String	0	0	1	1	1
State i	s_1	s_1	s_2	s_2	s_2
Output	YES				

String	1	1
State i	s_2	s_2
Output	YES	

String	1
State i	s_2
Output	YES

String	0	1	0	1
State i	s_1	s_2	s_3	s_3
Output	NO			

Now that we really understand what a finite state acceptor is, we can develop a criterion for recognising regular languages called the Myhill-Nerode theorem based on an equivalence relation we can set up on words in our language over the alphabet A.

Definition: Let $x, y \in L$, a language over the alphabet A. We call x and y equivalent over L denoted by $x \equiv_L y$ if $\forall w \in A^*, xw \in L \Leftrightarrow yw \in L$.

Note: xw means the concatenation $x \circ w$, and yw is the concatenation $y \circ w$.

Idea: If $x \equiv_L y$, then x and y place our finite state acceptor into the $\underline{\text{same state}}$ s.

Notation: Let $L/\equiv$ be the set of equivalence classes determined by the equivalence relation $\equiv_L$.

The Myhill-Nerode Theorem: Let L be a language over the alphabet A. If the set $L/\equiv$ of equivalence classes in L is infinite, then L is not a regular language.

Stretch of Proof: All elements of one equivalence class in $L/\equiv$ place our automaton into the same state s. Elements of distinct equivalence classes place the automaton into distinct states, **i.e.** if $[x], [y] \in L/\equiv$ and $[x] \neq [y]$, then all elements of $[x]$ place the automaton into some state s, while all elements of $[y]$ place the automaton into some state s', with $s \neq s' \Rightarrow$ an automaton that can recognise L has $\underline{\text{as many}}$ states at the number of equivalence classes in $L/\equiv$, but $L/\equiv$ is $\underline{\text{NOT}}$ finite $\Rightarrow L$ cannot be recognised by a finite state automaton $\Rightarrow L$ is not regular by the theorem above.

qed

8.4 Regular Grammars

Task: Understand what is the form of the production rules of a grammar that generates a regular language.

Recall: that a context-free grammar is given by $(V, A, <s>, P)$ where every production rule <T>$\rightarrow w$ in P causes $\underline{\text{one and only one}}$ nonterminal to be replaced by a string in V^*.

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

(i) <A>$\rightarrow$ b<B>

(ii) <A>$\rightarrow$ b

(iii) <A>$\rightarrow \varepsilon$

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 type (i) 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 form given by production rules:

1. <s>→ 0<A>
2. <A>→ 0<A>
3. <A>→ ε
4. <s>→ ε
5. <A>→ 1<B>
6. <B>→ 1<B>
7. <s>→ 1 <B>
8. <B>→ ε

Rules (1), (2), (5), (6), (7) are of type (i), while rules (3), (4) and (8) are of type (iii).
(1) and (3) give 0. (1), (2) applied $m-1$ times and (3) give 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) give 01^n for $n \geq 2$.
(1), (2) applied $m-1$ times, (5) and (8) give $0^m 1$ for $m \geq 2$.
(1), (2) applied $m-1$ times, (5), (6) applied $n-1$ times, and (8) give $0^m 1^n$ 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, <u>but</u> we can construct the finite state acceptor from the regular grammar as follows: our regular grammar is given by $(V, A, <s>, P)$. <u>Want</u> 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 type (iii), <A>$\to \varepsilon$. Those are applied when we <u>finish</u> 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 $t(\text{<A>}, w) = f$. 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.

8.5 Regular expressions

Task: Understand another equivalent way of characterizing regular languages due to Kleene in the 1950's.

Definition: 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.

Remark: This definition is an inductive one.

NB It is important not to confuse the regular expressions $\emptyset$ and ϵ. The expression ϵ represents the language consisting of a single string, namely ϵ, the empty string, whereas $\emptyset$ represents the language that does not contain any strings. Recall that a language L is any subset of

$$A^* = \bigcup_{n=0}^{\infty} A^n = A^0 \cup A^1 \cup A^2 \cup \cdots ,$$

where $A^0 = \{\epsilon\}$, the set of words of length 0, $A^1 =$ the set of words of length 1, and $A^2 =$ the set of words of length 2.

Precedence order of operations if parentheses are not present:

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

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

$$\begin{aligned} 1^* \circ 0 &= \{w \in A^* \mid w = 1^m 0 \text{ for } m \in \mathbb{N}, m \geq 0\} = \{0, 10, 110, 1110, \dots\} \\ &= 1^*0. \end{aligned}$$

We can omit the concatenation symbol.

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

$$\begin{aligned} A^* \circ 1 \circ A^* &= \{w \in A^* \mid w \text{ contains at least one } 1\} \\ &= \{u \circ 1 \circ v \mid u, v \in A^*\} = A^* 1 A^* \end{aligned}$$

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

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

Recall that $L^* = \bigcup_{n=0}^{\infty} L^n$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and inductively $L^n = L \circ L^{n-1}$. Here $L = \{00, 01, 10, 11\}$.

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

(4) $A = \{0,1\}$ $\quad (0 \cup \epsilon) \circ (1 \cup \epsilon) = \{\epsilon, 0, 1, 01\}$.

(5) $\epsilon^* = \{\epsilon\}$.

(6) $\emptyset^* = \{\epsilon\}$. 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.

Use of regular expressions in programming:

$\rightarrow$ 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 with regular expressions. We get the syntax of a programming language this way. There exists an algorithm for recognizing regular expressions that has been implemented $\Longrightarrow$ an automatic system generates the lexical analyzer that checks the input in a compiler.

$\rightarrow$ eliminate redundancy in programming

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

Theoretical importance of regular expressions

For the study of formal languages and grammars, the importance of regular expressions comes from the following theorem:

Theorem: A language is regular $\Longleftrightarrow$ some regular expression describes it.

Sketch of proof: Recall the definition of a regular language as the language obtained in finitely many steps from finite subsets of words via union, concatenation or the Kleene star. We can construct a regular expression from the definition of the regular language in question, and vice versa starting with a regular expression, we can define a finite sequence of L_i's such that each L_i is a finite set of words or is obtained from previous L_i's via union, concatenation or the Kleene star.

qed

Finally, we can state the complete characterization of regular languages:

Theorem: The following are equivalent:

(i) L is a regular language.

(ii) L is recognized by a (deterministic or non-deterministic) finite state acceptor.

(iii) L is produced by a regular grammar.

(iv) L is given by a regular expression.

Remark: It is possible to prove directly that (iv) $\Longleftrightarrow$ (ii), but the construction is rather complicated. Instead, we sketched above the proof that (i) $\Longleftrightarrow$ (iv), and we had previously stated that (i) $\Longleftrightarrow$ (ii) $\Longleftrightarrow$ (iii), so we now have that (i) $\Longleftrightarrow$ (ii) $\Longleftrightarrow$ (iii) $\Longleftrightarrow$ (iv).

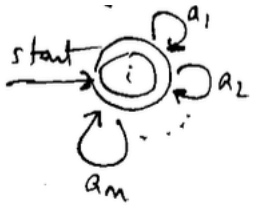
Example: Let $L = \{0^m 1^n \bigm| m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$ be the regular language we considered before. We now give a regular expression for L: $L = 0^* \circ 1^*$. Recall we previously show this language is regular from the definition of a regular language, so solving this problem is a direct illustration of the implication (i) $\Longleftrightarrow$ (iv).

8.6 The Pumping Lemma

Task: Understand another criterion for figuring out when a language is regular.

Let a finite set A be the alphabet, and let L be a language over A. Then $L \subset A^*$. We make the following two crucial observations:

1. If L is finite, then clearly there exists a finite state acceptor that recognizes $L \Rightarrow L$ is regular.
2. If $L = A^*$, then L is likewise regular. Here is why: Let $A = \{a_1, \ldots, a_n\}$. The acceptor



with just one state i recognizes A^*.

Question: If L is infinite, but $L \subsetneqq A^*$, how can we tell whether L is regular?

Answer: The Myhill-Nerode Theorem would have us look at equivalence classes of words, but that analysis can be complicated at times. The Pumping Lemma provides another way of checking whether L is regular.

The Pumping Lemma: If L is a regular language, then there is a number p (the pumping length) where if w is any word in L of length at least p, then $w = xuy$ for words x, y, and u satisfying:

1. $u \neq \epsilon$ (i.e., $|u| > 0$, the length of u is positive);
2. $|xu| \leq p$;
3. $xu^n y \in L \; \forall \, n \geq 0$.

Remark: p can be taken to equal the number of states of a deterministic finite state acceptor that recognizes L (we know such a finite state acceptor exists because L is regular).

Sketch of proof: The name of the lemma comes from the fact that if L is regular, then all of its words can be pumped through a finite state acceptor that recognizes L. We assume this acceptor is deterministic and has p states. We will show the Pumping Lemma is a consequence of the Pigeonhole Principle we studied in the unit on functions. If a word w has length l, then the finite state acceptor must process l pieces of information ($w = a_1 a_2 \cdots a_l$, where $a_k \in A \; \forall \, k, \; 1 \leq k \leq l$) $\Longrightarrow$ it passes through $l+1$ states starting with the initial state. In the hypotheses of the lemma, we assume $|w| = l \geq p$, but $p = \#$(states of the acceptor) $\Longrightarrow$ the acceptor passes through $l + 1 \geq p + 1$ states to process w and therefore at least one state is repeated among the first $p+1$. Let s_1, s_2, ..., s_{l+1} be the sequence of states. $|w| = l \geq p \implies s_i = s_j$ with $i < j \leq p+1$. Now we set x to be the part of w that makes the acceptor pass through states s_1, s_2, ..., s_i, i.e., $x = a_1 a_2 \cdots a_{i-1}$ (the first $i-1$ letters in w). We set u to be the part of w that makes the acceptor pass through states s_i, s_{i+1}, s_{i+2}, ..., s_j. In other words, $u = a_i a_{i+1} \cdots a_{j-1}$. Since $i < j$, $|u| \geq 1 \implies u \neq \epsilon$. Finally, set y to be the part of w (the tail end) that makes the acceptor pass through states s_j, s_{j+1}, ..., s_{l+1}, i.e., $y = a_j a_{j+1} \cdots a_l$. Since $j \leq p+1$, $j-1 \leq p$, so $|xu| = |a_1 a_2 \cdots a_{j-1}| = j-1 \leq p$ as needed. Furthermore, $s_i = s_j$, so at the beginning of u and at its end the acceptor is in the same state $s_i = s_j \implies xu^n y$ is accepted for every $n \geq 0 \implies xu^n y \in L$ as needed. We have obtained conditions (1)-(3).

qed

Applications of the Pumping Lemma

As a statement, the Pumping Lemma is the implication $P \to Q$ with P being the sentence "L is a regular language" and Q being the decomposition of every w, $|w| \geq p$ as $w = xuy$. We use the contrapositive $\neg Q \to \neg P$ (tautologically equivalent to $P \to Q$) as our criterion for detecting non-regular languages.

Examples:

1. $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 0\}$ is not regular. Let $w = 0^m 1^m$. We cannot decompose w as $w = xuy$ because whatever we let u be, we get a contradiction to $xu^n y \in L \; \forall\, n \geq 0$. If $u \in 0^*$ (string of 0's), $x \in 0^*$ and $y = 0^s 1^m$ (string of s 0's with $s \geq 0$ and m 1's). If $n \geq 2$, $xu^n y \notin L$ because $xu^n y$ has more 0's than 1's.
   If $u \in 1^*$, we get a contradiction the same way (more 1's than 0's in this case).
   If $u \in 0^* 1^*$, $xu^2 y \notin L$ for any x, y words!

2. $L = \{0^m \mid m \text{ is prime}\}$ is not regular.
   Since $w = 0^m$, x, u, y can consist only of 0's, so then $x = 0^i$, $u = 0^j$, $y = 0^k$. If $xu^n y \in L \; \forall\, n \geq 0$, then $i + nj + k$ is prime $\forall\, n \geq 0$, which is impossible.
   Set $n = i + 2j + k + 2$, then

$$\begin{aligned} i + nj + k &= i + (i + 2j + k + 2)j + k = i + ij + 2j^2 + jk + 2j + k \\ &= i(j+1) + 2j(j+1) + k(j+1) = (j+1)(i + 2j + k), \end{aligned}$$

   where $|u| > 0$, so $j \geq 1$. Therefore, $n = (j+1)(i+2j+k)$ is not prime!

Practice at home: `weitz.de/pump` (on Edi Weitz's website)

The pumping game, an online game to help you understand the Pumping Lemma.

8.7 Applications of Formal Languages and Grammars as well as Automata Theory

1. Compiler architecture uses context-free grammars
2. Parsers - recognise if commands comply with the syntax of a language
3. Pattern matching and data mining - guess the language from a given set of words (applied in CS, genetics, etc.)
4. Natural language processing - example in David Wilkins' notes pp.40-44
5. Checking proofs by computers/automatic theorem proving - simpler example of this kind in David Wilkins' notes pp.45-57 that pertains to propositional logic
6. The theory of regular expressions enables
   (a) grep/awk/sed in Unix
   (b) More efficient coding (avoiding unnecessary detours in your code)
7. Biology - John Conway's game of life is a cellular automaton
8. Modelling of AI characters in games uses the finite state automaton idea. Our character can choose among different behaviours based on stimuli - like a finite state automaton reacting to input
9. Strategy and tactics in games - teach the opposition to recognise certain patterns, then suddenly change them to gain an advantage and score - used in football, fencing, etc.
10. Learning a sport/a numerical instrument/a new field or subject - split the information into blocks and learn how to combine them into meaningful patterns - uses notions from context-sensitive grammars.
11. Finite state automata and probability $\rightsquigarrow$ Markov chains - chaos theory, financial mathematics.

etc...

9 Graph Theory

Task: Introduce terminology related to graphs; understand different types of graphs; learn how to put together arguments involving graphs.
An <u>undirected graph</u> consists of:

1. A finite set of points V called <u>vertices</u>
2. A finite set E of <u>edges</u> joining two distinct vertices of the graph.