Week 1

Alphabets: $\Sigma, \Gamma, \ldots$ (uppercase Greek letters). E.g. the binary alphabet $\Sigma = \{0,1\}$.

Size of an alphabet (or more generally any set): $|\Sigma|$.

Alphabet symbols: $a, b, c, \ldots$ (lowercase Latin letters from the beginning of the alphabet). For instance, if $\Sigma = \{a_1, \ldots, a_n\}$ is an alphabet, then $|\Sigma| = n$.

Strings: $u, v, w, x, y, \ldots$ (lowercase Latin letters from the end of the alphabet).

Concatenation of strings: $x \widehat{\ } y$ or simply xy.

Length of a string: $|x|$. For instance,

- $|abc| = 3$;
- If $x = a_1 \ldots a_m$ and $y = b_1 \ldots b_n$, then $|xy| = m+n$.

Empty string: ε.

String of n symbols a: a^n. As an example,

- $a^n = \underbrace{aa\ldots a}_{n \text{ symbols}}$,
- $a^2b^3 = aabbb$, and
- $|a^i b^j c^k| = i+j+k$.

Repeating a string x k times: x^k. For instance,

- $(ab)^2 = abab$, and
- $|x^k| = k|x|$.

Set of all string over an alphabet Σ: Σ^*. As an example,

- $\{a,b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

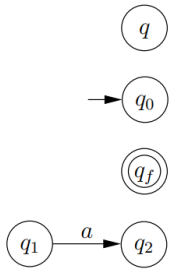
Example:

Let Σ be an alphabet. The *reversal* w^R of a string $w \in \Sigma^*$ is defined inductively with the following rules:

1. $\varepsilon^R = \varepsilon$
2. if $w = ua$, $u \in \Sigma^*$, and $a \in \Sigma$, then $w^R = a \widehat{\ } u^R$.

Week 2: Deterministic Finite Automata (DFA)

State diagram notation:



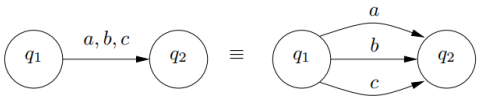
A state q

A start state q_0 (also called initial state)

An accepting state q_f

A transition from state q_1 to state q_2 on input symbol a

An abbreviation:



Definition 2.1 (Finite automata)

Definition 2.1 (Finite automata)

A *finite automaton* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *states*,
- Σ is a finite set called *alphabet*,
- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *final* or *accepting states*.

Lemma 2.1

Lemma 2.1

If a language $L \subseteq \Sigma^\star$ can be recognised with a finite automaton, then the complement language $\overline{L} = \{w \in \Sigma^\star \mid w \notin L\}$ can be recognised with a finite automaton as well.

Proof

Let $M_L = (Q, \Sigma, \delta, q_0, F)$ be any FA that recognises the language L (i.e., has $\mathcal{L}(M_L) = L$).
As the state of M_L at the end of its computation on any input string is unique and well-defined, we get an automaton for the complement language $\overline{L} = \{w \in \Sigma^\star \mid w \notin L\}$ simply by replacing all the accept states in M_L by non-accept ones and vice versa. (Note that the presentation of M_L is here assumed to be complete, i.e. all the entries in the transition table/diagram have been filled in.)
Thus, the FA $M_{\overline{L}} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognises $\overline{L}$.

Lemma 2.2: Product of FA

Lemma 2.2

If languages $A, B \subseteq \Sigma^\star$ can be recognised with finite automata, then so can the language $A \cap B = \{w \in \Sigma^\star \mid w \in A \text{ and } w \in B\}$.

Proof

Proof by construction.
Let $M_A = (Q_A, \Sigma, \delta_A, q_{A,0}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{B,0}, F_B)$ be some FA that recognise the languages A and B, respectively.
We build an automaton $M_{A\cap B} = (Q_{A\cap B}, \Sigma, \delta_{A\cap B}, q_{A\cap B,0}, F_{A\cap B})$ that "simulates" both automata at the same time and accepts a string if and only if both automata would.
$M_{A\cap B}$ is called the *(synchronous) product* automaton, and defined as:

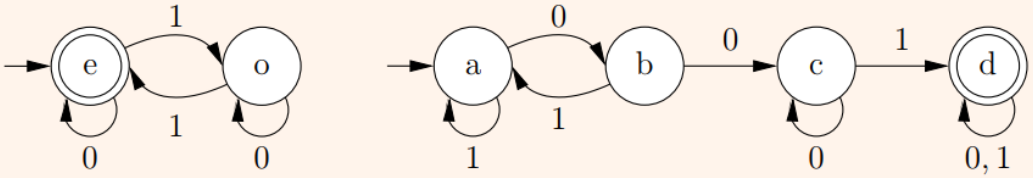
- $Q_{A\cap B} = Q_A \times Q_B$, meaning that the states are pairs that record in which states the simulated automata A and B would be.
- $q_{A\cap B,0} = (q_{A,0}, q_{B,0})$, indicating that in the beginning both simulated automata are in their start states.
- $\delta_{A\cap B}((q_a, q_b), \sigma) = (\delta_A(q_a, \sigma), \delta_B(q_b, \sigma))$, meaning that $M_{A\cap B}$ moves to a new state that corresponds to the states of A and B when they would read the same symbol σ.
- $F_{A\cap B} = F_A \times F_B$, so that $M_{A\cap B}$ accepts exactly when both of the simulated automata would.

Now one could prove, by using induction on the length of the input string, that if, after reading the input string, automaton A is in state q_a and B is in state q_b, then automaton $M_{A\cap B}$ is in state (q_a, q_b).

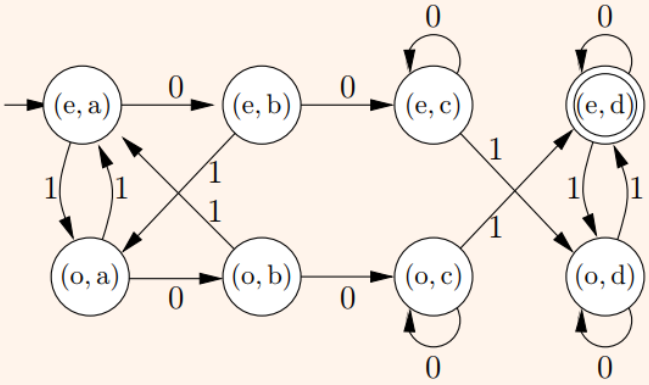
Example:

By using our earlier examples and the construction in the previous proof, we can build an automaton that accepts exactly the strings that contain an even number of symbol 1 *and* 001 as a substring.

Component automata:



The product automaton:



Week 3: Nondeterministic Final Automata (NFA)

- Minimization of DFA

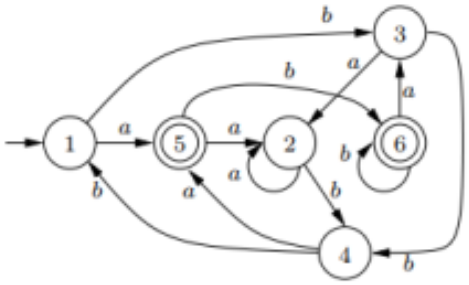
Lemma 3.1

Lemma 3.1

(i) Two k-equivalent states, q_1 and q_2, are $(k+1)$-equivalent if and only if $\delta(q_1,a) \overset{k}{\equiv} \delta(q_2,a)$ for all $a \in \Sigma$.

(ii) If for some k it holds that *all* mutually k-equivalent states are also $(k+1)$-equivalent, then they are fully equivalent as well.

H3.1 Construct the minimal automaton corresponding to the following deterministic finite automaton:



Exercise H3.1

- Step 1: Remove unreachable state. Since all states in this automaton are reachable, there's no state that can be dropped
- Step 2: The states are partitioned into non-accept states I and accept states II.

		a	b
I: →	1	5	3
	2	2	4
	3	2	4
	4	5	1
II: ←	5	2	6
←	6	3	6

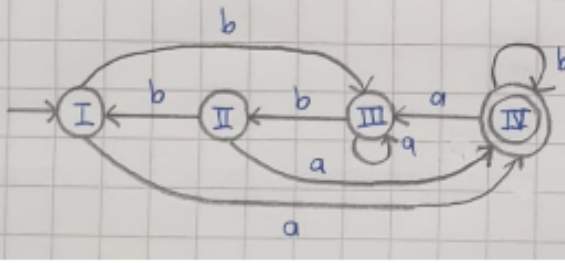
- Step 3: marking which class the transition function maps each pair

		a	b
I: →	1	5, II	3, I
	2	2, I	4, I
	3	2, I	4, I
	4	5, II	1, I
II: ←	5	2, I	6, II
←	6	3, I	6, II

=) Class I has 3 kinds of states {1}, {2, 3} and {4} =)

		a	b
I: →	1	5, IV	3, III
II:	4	5, IV	1, I
III:	2	2, III	4, II
	3	2, III	4, II
IV: ←	5	2, III	6, IV
←	6	3, III	6, IV

- Step 4: The resulting minimal finite automaton is



- Definition 3.1 (Nondeterministic Finite Automata - NFA)

Definition 3.1 (Nondeterministic finite automata)

A nondeterministic finite automaton is a tuple

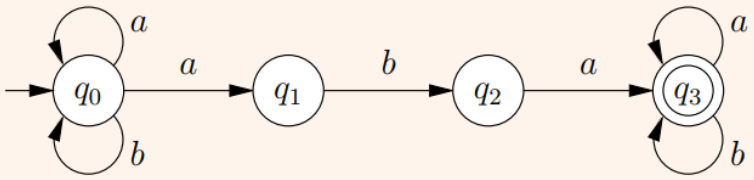
$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *states*,
- Σ is the *input alphabet*,
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the set-valued *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is the set of *accept states*.

Example:

An "aba-automaton" that detects whether the input string contains a substring aba:



The transition function is

		a	b
$\to$	q_0	$\{q_0, q_1\}$	$\{q_0\}$
	q_1	$\emptyset$	$\{q_2\}$
	q_2	$\{q_3\}$	$\emptyset$
$\leftarrow$	q_3	$\{q_3\}$	$\{q_3\}$

For example, $\delta(q_0, a) = \{q_0, q_1\}$ and $\delta(q_1, a) = \emptyset$.

Theorem 3.2 (Determinisation of NFA)
Converting NFA into DFA

Theorem 3.2 (Determinisation of NFA)

Let $A = L(M)$ be a language recognised by a nondeterministic FA M. Then there exists also a deterministic FA $\widehat{M}$ such that $A = L(\widehat{M})$.

Proof

Let $A = L(M)$ for some nondeterministic FA $M = (Q, \Sigma, \delta, q_0, F)$. The idea is to construct a deterministic FA $\widehat{M}$ that simulates the operation of M in all states possible at each step *in parallel*.
Formally, the states of $\widehat{M}$ are *sets* of states of M:

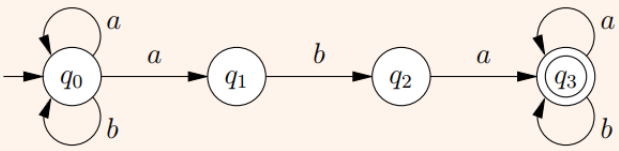
$$\widehat{M} = (\widehat{Q}, \Sigma, \hat{\delta}, \hat{q}_0, \widehat{F}),$$

where

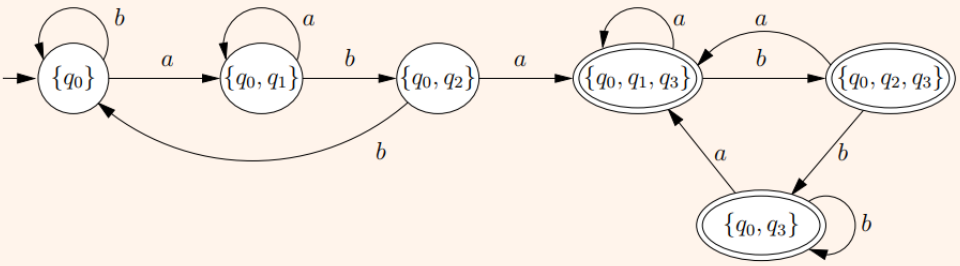
$$\begin{aligned} \widehat{Q} &= \mathcal{P}(Q) = \{S \mid S \subseteq Q\}, \\ \hat{q}_0 &= \{q_0\}, \\ \widehat{F} &= \{S \subseteq Q \mid S \cap F \neq \emptyset\}, \\ \hat{\delta}(S,a) &= \bigcup_{q \in S} \delta(q,a). \end{aligned}$$

Example:

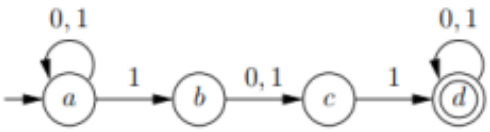
When applied to the aba-automaton



the algorithm produces the following deterministic automaton (only those states that can be reached from the new initial state are drawn):



H3.2 Consider the following nondeterministic finite automaton M recognising a language $\mathcal{L}(M)$ over the alphabet $\{0,1\}$:



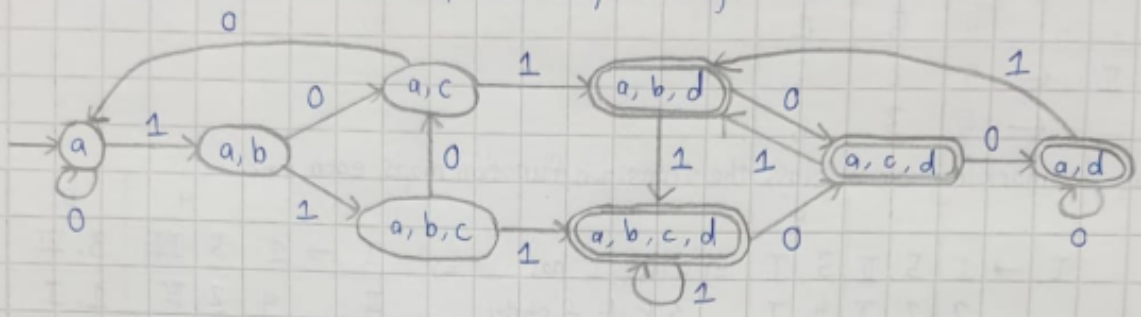
Construct the *minimal deterministic* finite automaton that recognises the *complement* of the language $\mathcal{L}(M)$.

Exercise 3.2

The transition function is:

		0	1
→	a	{a}	{a, b}
	b	{c}	{c}
	c	∅	{d}
←	d	{d}	{d}

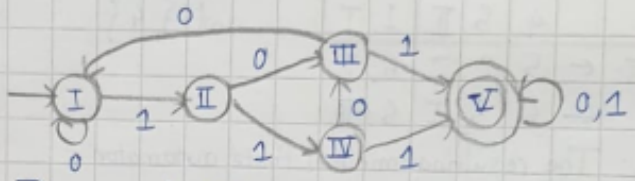
- The deterministic automaton produced by the algorithm is:



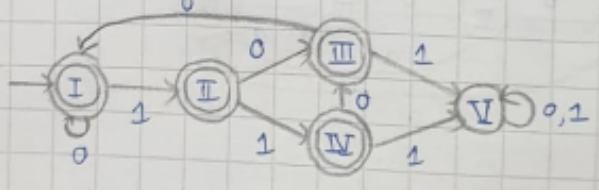
- Marking the classes of the transition function like ex H3.1

The minimal FA would be:

	0	1
→ q_1 : I	q_1 : I	q_2 : II
q_2 : II	q_3 : III	q_4 : IV
q_3 : III	q_1 : I	q_5 : V
q_4 : IV	q_3 : III	q_6 : V
← q_5 : V	q_7 : V	q_6 : V
← q_6 : V	q_7 : V	q_6 : V
← q_7 : V	q_8 : V	q_5 : V
← q_8 : V	q_8 : V	q_5 : V



⇒ The minimal DFA that recognizes the complement of the language is by making above automaton accept as non-accept and vice versa



Empty-automata: ε-eutomata

Definition 3.2 (ε-automata)

An *ε-automaton* is a tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where the transition function δ is a function

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q).$$

The other definitions are as for standard nondeterministic FA except that the "leads directly" relation is now defined so that

$$(q, w) \underset{M}{\vdash} (q', w')$$

if

- $w = aw'$ for an $a \in \Sigma$ and $q' \in \delta(q, a)$, or
- $w = w'$ and $q' \in \delta(q, \varepsilon)$.

Theorem 3.3: (Eliminating empty-transitions)

Theorem 3.3 (Eliminating ε-transitions)

Let $A = L(M)$ for an ε-automaton M. Then there exists also a standard nondeterministic FA $\widehat{M}$ such that $L(\widehat{M}) = A$.

Proof

Let $M = (Q, \Sigma, \delta, q_0, F)$ be any ε-automaton. Intuitively, the automaton $\widehat{M}$ we construct below works otherwise exactly as M except that it "jumps over" ε-transitions by taking only those "proper" transitions from each state that may follow immediately after a sequence of ε-transitions.

Formally, given a state $q \in Q$, we define its *ε-closure* $\varepsilon^*(q)$ in M by

$$\varepsilon^*(q) = \{q' \in Q \mid (q, \varepsilon) \underset{M}{\vdash}^* (q', \varepsilon)\}.$$

That is, $\varepsilon^*(q)$ consists of all the states of M that can be reached from q by taking only ε-transitions.

The automaton $\widehat{M}$ can now be defined as follows:

$$\widehat{M} = (Q, \Sigma, \hat{\delta}, q_0, \widehat{F}),$$

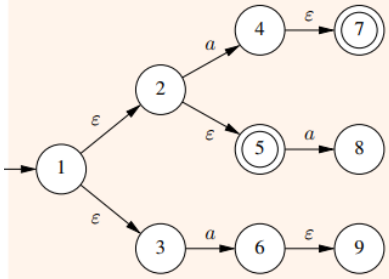
where

$$\begin{aligned} \hat{\delta}(q, a) &= \bigcup_{q' \in \varepsilon^*(q)} \delta(q', a), \\ \widehat{F} &= \{q \in Q \mid \varepsilon^*(q) \cap F \neq \emptyset\} \end{aligned}$$

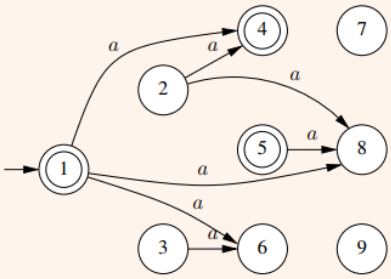
Example:

By removing ε-transitions from an ε-automaton with the above construction we get a standard nondeterministic automaton:

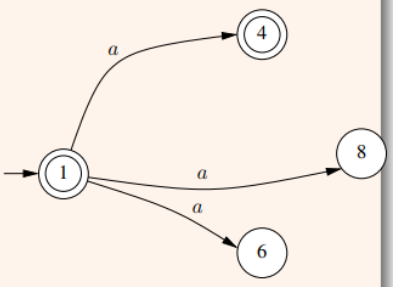
Original ε-automaton:



After removing ε-transitions:



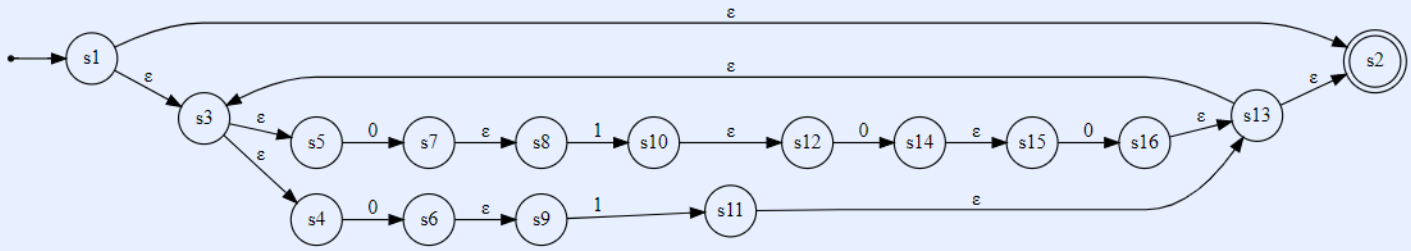
After removing unreachable states:



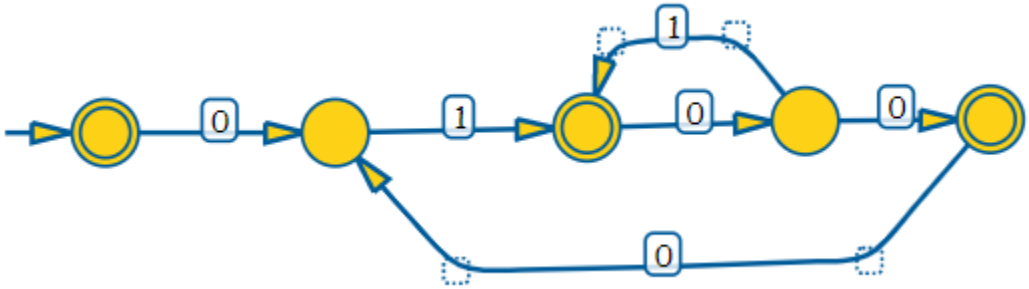
Consider the automaton after removing the ε-transitions:

1. Why is the initial state “1” now an accepting final state?
2. Why does it not matter that the accepting final state “7” is not anymore reachable from the initial state?

Consider the following automaton



Design a *deterministic* finite automaton (DFA) that recognises the same language.



Week 4: Regular Expressions (Regex)

Syntax and semantics of regular expressions

Let A and B be languages over an alphabet Σ.

- *Union:* $A \cup B = \{x \in \Sigma^* \mid x \in A \text{ or } x \in B\}$
- *Concatenation:* $AB = \{xy \in \Sigma^* \mid x \in A \text{ and } y \in B\}$
- *Powers:*

$$\begin{cases} A^0 &= \{\varepsilon\}, \\ A^k &= AA^{k-1} = \{x_1 \dots x_k \mid x_i \in A \quad \forall i = 1, \dots, k\}, \quad \text{for } k \geq 1 \end{cases}$$

- *Kleene closure (or "Kleene star"):*

$$\begin{aligned} A^* &= \bigcup_{k \geq 0} A^k \\ &= \{x_1 \dots x_k \mid k \geq 0,\ x_i \in A \quad \forall i = 1, \dots, k\} \end{aligned}$$

Definition 4.1 (Syntax of Regular Expressions)

Definition 4.1 (Syntax of regular expressions)

Regular expressions over an alphabet Σ are defined inductively by the following rules:

1. $\emptyset$ and ε are regular expressions over Σ.
2. a is a regular expression over Σ when $a \in \Sigma$.
3. If r and s are regular expressions over Σ, then also $(r \cup s)$, (rs), and r^* are regular expressions over Σ.
4. There are no other regular expressions over Σ.

Note

All the rules are purely syntactic. Thus, for instance '$\emptyset$' and '$\cup$' are here just symbols, without any meaning (yet).

The first two rules are the "base cases" while the third one is the inductive (recursive) case.

The last case is usually implicitly assumed and thus omitted.

Definition 4.2 (Semantics of regular expressions)

Definition 4.2 (Semantics of regular expressions)

A regular expression r over Σ *describes* the language $\mathcal{L}(r)$ defined inductively as follows:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$ when $a \in \Sigma$
- $\mathcal{L}((r \cup s)) = \mathcal{L}(r) \cup \mathcal{L}(s)$
- $\mathcal{L}((rs)) = \mathcal{L}(r)\mathcal{L}(s)$
- $\mathcal{L}(r^*) = (\mathcal{L}(r))^*$

Example

Some regular expressions over the alphabet $\{a, b\}$:

$$r_1 = ((ab)b), \quad r_2 = (ab)^*, \quad r_3 = (ab^*), \quad r_4 = (a(b \cup (bb)))^*$$

The languages described by the expressions are:

$$\begin{aligned}
\mathcal{L}(r_1) &= (\{a\}\{b\})\{b\} = \{ab\}\{b\} = \{abb\} \\
\mathcal{L}(r_2) &= \{ab\}^* = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^i \mid i \geq 0\} \\
\mathcal{L}(r_3) &= \{a\}(\{b\})^* = \{a, ab, abb, abbb, \ldots\} = \{ab^i \mid i \geq 0\} \\
\mathcal{L}(r_4) &= (\{a\}\{b, bb\})^* = \{ab, abb\}^* \\
&= \{\varepsilon, ab, abb, abab, ababb, \ldots\} \\
&= \{x \in \{a, b\}^* \mid \text{if } x \neq \varepsilon \text{ then it begins with an } a \text{ and} \\
&\qquad \text{each } a \text{ in } x \text{ is followed by 1 or 2 } b\text{s.}\}
\end{aligned}$$

Some simplification rules:

$$\begin{aligned}
r \cup (s \cup t) &= (r \cup s) \cup t \\
r(st) &= (rs)t \\
r \cup s &= s \cup r \\
r(s \cup t) &= rs \cup rt \\
(r \cup s)t &= rt \cup st \\
r \cup r &= r
\end{aligned}$$

$$\begin{aligned}
r \cup \emptyset &= r \\
\varepsilon r &= r \\
\emptyset r &= \emptyset \\
r^* &= \varepsilon \cup r^* r \\
r^* &= (\varepsilon \cup r)^*
\end{aligned}$$

In fact, any valid equivalence between regular expressions can be derived from these equations and the rule:

$$\text{if } \varepsilon \notin \mathcal{L}(s) \text{ and } r = rs \cup t, \text{ then } r = ts^*$$

Definition 4.3 (Regular languages)

Definition 4.3 (Regular languages)

A language is *regular* if it can be described with a regular expression.

Theorem 4.1 (Language recognised by a regex is also recognised by a DFA)

Theorem 4.1

If a language can be described with a regular expression, then it can be recognised by a finite automaton.

Proof

By using the inductive construction on the next slide, we can design, for each regular expression r, a nondeterministic automaton M_r with ε-transitions such that $\mathcal{L}(M_r) = \mathcal{L}(r)$. The resulting automaton can then be determinised if needed (cf. Lecture 3). In the construction, the intermediate component automata always have a standard form with unique and distinct initial and final states, and no transitions either

1. entering the initial state, or
2. exiting the final state.

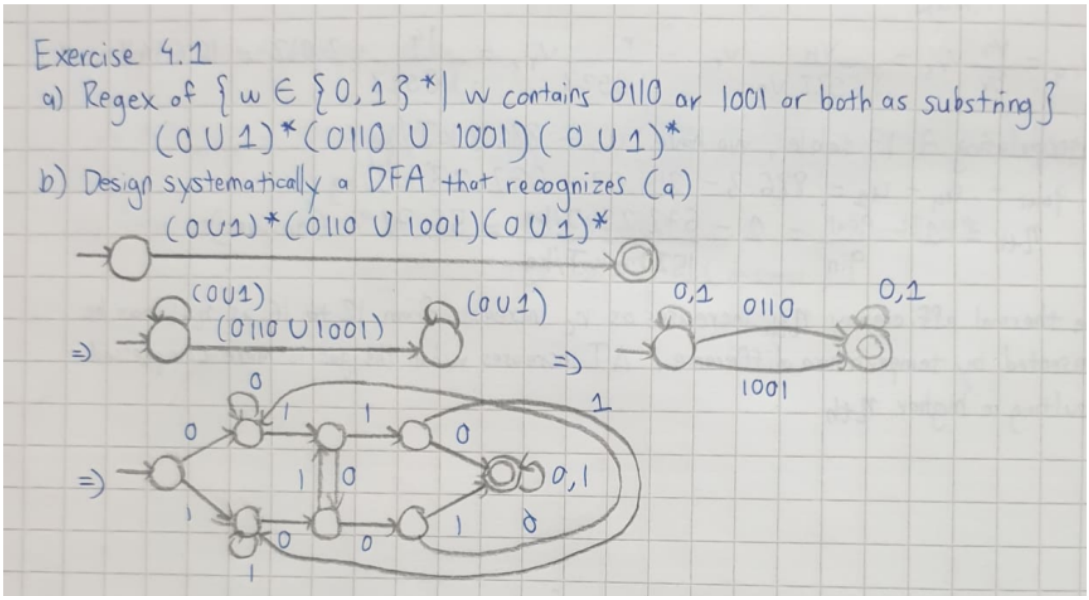
This is important when putting the component automata together. □

H4.1

(a) Give a regular expression that describes the language

$$\{w \in \{0,1\}^* \mid w \text{ contains } 0110 \text{ or } 1001 \text{ as a substring (possibly both)}\}.$$

(b) Following the guidelines presented in the lectures, design in a systematic way a deterministic finite automaton that recognises the language in part (a).



Theorem 4.2 (Language recognised by a DFA is also recognised by a regex)
=> DFA and Regex can recognize each other languages from theorem 4.1 and 4.2

Theorem 4.2

If a language can be recognised by a finite automaton, then it can be described with a regular expression.

Proof

We need one more extension of finite automata, the *generalised nondeterministic finite automata* (abbreviated GNFA), which allow transitions that are labelled with regular expressions.

Formalisation: Let RE_Σ be the set of regular expressions over Σ. A GNFA is a tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where the transition function δ is a *finite* mapping

$$\delta : Q \times \mathrm{RE}_\Sigma \to \mathcal{P}(Q)$$

(that is, $\delta(q,r) \neq \emptyset$ holds only for finitely many pairs $(q,r) \in Q \times \mathrm{RE}_\Sigma$).

Note: This definition is different from that in Sipser's book (Definition 1.64) but serves the same purpose.

The "leads directly", or "leads in one step" relation is now defined

$$(q,w) \underset{M}{\vdash} (q',w')$$

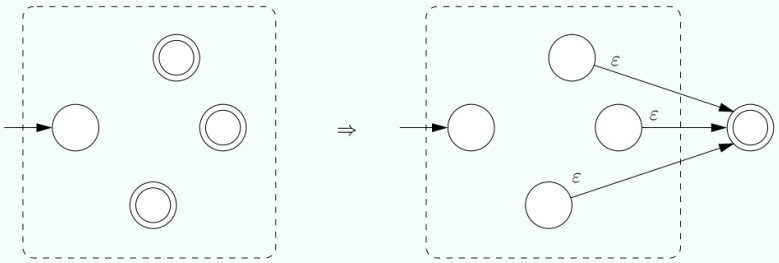
if $q' \in \delta(q,r)$ for some $r \in \mathrm{RE}_\Sigma$ such that $w = zw'$ and $z \in \mathcal{L}(r)$. Other definitions are as earlier.

Let us now prove: every language recognised by a GNFA can be described with a regular expression.

Let M be a GNFA. A regular expression that describes the language recognised by M can be constructed in three phases:

Methods of converting Regex into DFA:

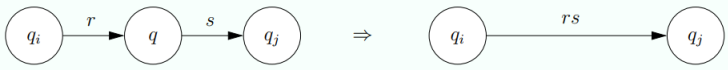
Phase I: If M has multiple final states, merge them as follows:



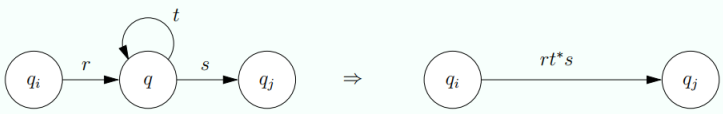
Phase II: Reduce M to an equivalent GNFA with at most two states, by removing all non-initial and non-final states one by one, using the following transformations:

Let q be any state in M that is not the initial or the final state. Consider all the "transition paths" in the state diagram of M that go through q. Let q_i and q_j be the immediate predecessor and successor states of q on some such path. Remove q from the path $q_i \to q_j$ by rule (i) below if q has no transition to itself, and by rule (ii) if it has:

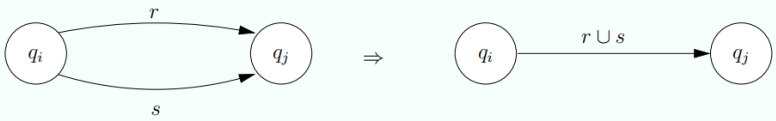
(i):



(ii):

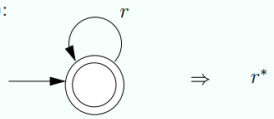


At the same time, merge parallel transitions with the rule:

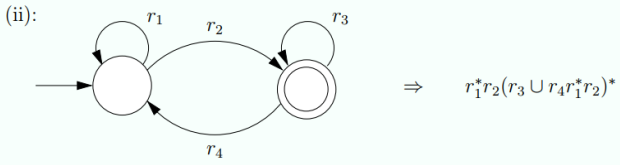


Phase III: At the end of the reduction process in Phase II, the automaton has at most two states. The corresponding regular expression is then constructed as follows:

(i):



(ii):



1. (a) Design a deterministic finite automaton that recognises the language

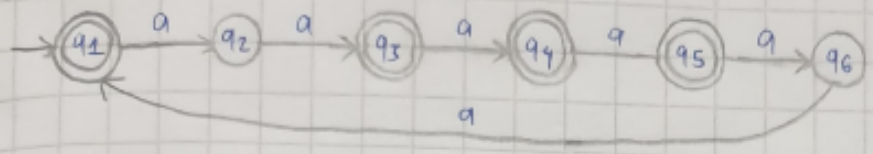
$$L = \{a^n \mid n \text{ is divisible by 2 or 3 (or both)}\}.$$

(The value 0 is considered to be divisible by any number.) 8 points

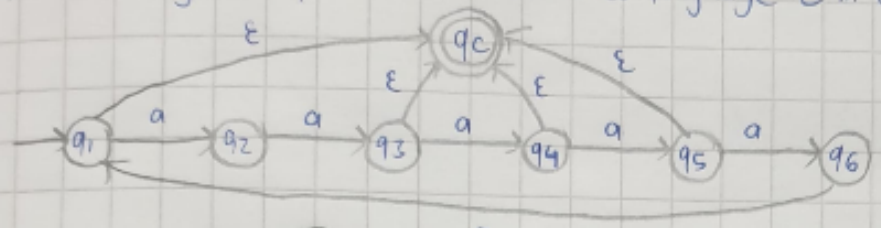
(b) Give a regular expression that describes the language L in part (a). 6 points

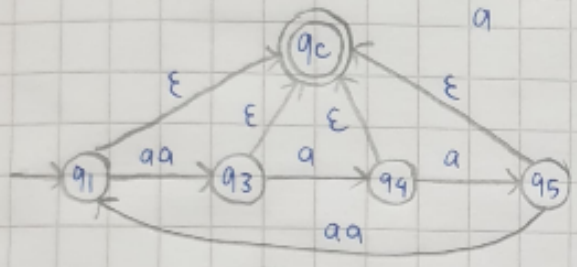
Exercise 1: Design DFA for

a) $L = \{a^n \mid n$ is divisible by 2 or 3 or both$\}$

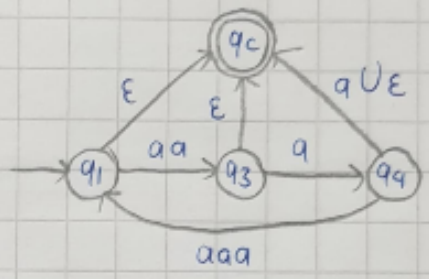


b) Give a regular expression that describes language L in (a)

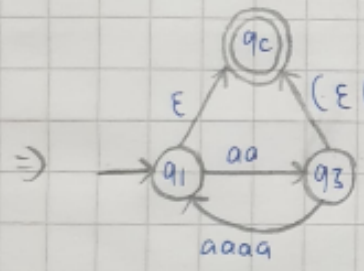




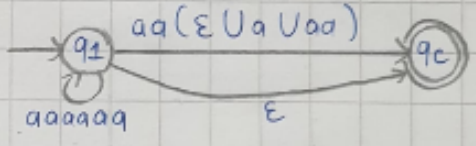
⇒



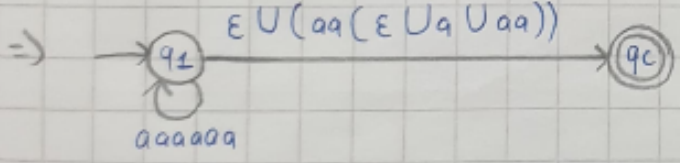
⇒



⇒



⇒



⇒ Regex of the language : $(aaaaaa)^*(\varepsilon \cup (aa(\varepsilon \cup a \cup aa)))$

or $(aaaaaa)^*(\varepsilon \cup aa \cup aaa \cup aaaa)$

Week 5: Context Free Grammars and Languages

Lemma 4.3 (Pumping lemma for regular languages)

Lemma 4.3 (Pumping lemma for regular languages)

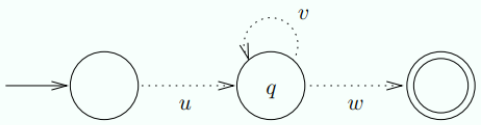
Let A be a regular language. Then there is a number $p \geq 1$ such that every $x \in A$ with $|x| \geq p$ can be divided in three parts, $x = uvw$, satisfying the conditions (i) $|uv| \leq p$, (ii) $|v| \geq 1$, and (iii) $uv^i w \in A$ for all $i = 0, 1, 2, \ldots$

Proof

Let M be a deterministic finite automaton that recognises A, i.e. has $\mathcal{L}(M) = A$. Let p be the number of states in M. Study the sequence of states visited when M is run on any input $x \in A$ with $|x| \geq p$. The initial state is visited first and then $|x|$ other states, one for each symbol in x. Thus at least $p+1$ states are visited, and since M has only p states, some state(s) are visited more than once. In fact, the revisit to some state happens already during the first p symbols of x. Let q be the first revisited state.

Let now:

- u be the prefix of x already processed by M's first visit to q,
- v the substring of x processed next before M's first revisit to q,
- w the remaining suffix of x.



Obviously $|uv| \leq p$ and $|v| \geq 1$. Consider what happens on any string of the form $uv^i w$, with $i \geq 0$: (i) M processes u similarly as when accepting $x = uvw$, and enters q, (ii) M processes v^i by executing the same loop as when accepting $x = uvw$, but now i times instead of once, and again enters q, (iii) processes w and enters the same accepting state as when accepting $x = uvw$. Thus M accepts $uv^i w$ as well, and so $uv^i w \in \mathcal{L}(M) = A$.

Example:

Consider the language of balanced parentheses (for clarity, replace '(' = a and ')' = b):

$$L = L_{\text{match}} = \{a^k b^k \mid k \geq 0\}.$$

Suppose (towards an eventual contradiction) that L were regular.
In that case, by the pumping lemma, there should be some $p \geq 1$ so that all the strings in L that have at least p symbols can be "pumped".
Choose one such string, say $x = a^p b^p$. Now $x \in L$ and $|x| = 2p > p$.
The lemma says that x can be divided in three parts, $x = uvw$, so that $|uv| \leq p$, $|v| \geq 1$ and $uv^i w \in L$ for all $i = 0, 1, 2, \ldots$. In detail:
$u = a^m$, $v = a^n$, $w = a^{p-(m+n)} b^p$ for some m, n s.t. $m + n \leq p$ and $n \geq 1$.
But when "pumping" the middle part zero times we get

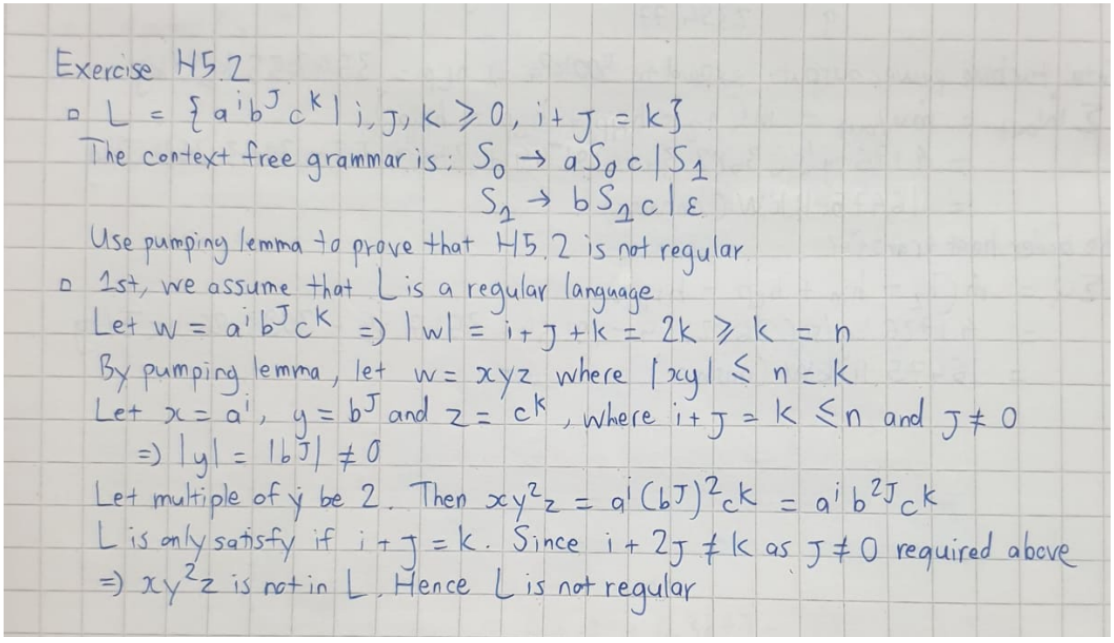
$$uv^0 w = a^m a^{p-(m+n)} b^p = \mathbf{a^{p-n} b^p} \notin \mathbf{L} \text{ as } n \geq 1.$$

From the contradiction, we conclude that the assumption must be wrong and in fact L is not regular.

H5.2 Design a context-free grammar that generates the language

$$L = \{a^i b^j c^k \mid i, j, k \geq 0,\ i + j = k\}.$$

Prove also, by using the pumping lemma, that the language L is not regular.



Example:

Consider the language of balanced parentheses (for clarity, replace '(' = a and ')' = b):

$$L = L_{\text{match}} = \{a^k b^k \mid k \geq 0\}.$$

Suppose (towards an eventual contradiction) that L were regular.
In that case, by the pumping lemma, there should be some $p \geq 1$ so that all the strings in L that have at least p symbols can be "pumped".
Choose one such string, say $x = a^p b^p$. Now $x \in L$ and $|x| = 2p > p$.
The lemma says that x can be divided in three parts, $x = uvw$, so that $|uv| \leq p$, $|v| \geq 1$ and $uv^i w \in L$ for all $i = 0, 1, 2, \ldots$. In detail:
$u = a^m$, $v = a^n$, $w = a^{p-(m+n)} b^p$ for some m, n s.t. $m + n \leq p$ and $n \geq 1$.
But when "pumping" the middle part zero times we get

$$uv^0 w = a^m a^{p-(m+n)} b^p = \mathbf{a^{p-n} b^p} \notin \mathbf{L} \text{ as } n \geq 1.$$

From the contradiction, we conclude that the assumption must be wrong and in fact L is not regular.

Note

- There are also non-regular languages that can be "pumped".
  ☞ Lemma 4.3 can **not** be used to show that a language **is** regular.
- As an example, the language

  $$L = \{c^r a^k b^k \mid r \geq 1, k \geq 0\} \cup \{a^k b^l \mid k, l \geq 0\}$$

  is *not* regular, even though the strings in it can be "pumped".
- Brainteaser: Show that (i) L satisfies the conditions of Lemma 4.3, (ii) L is not regular.

Definition 5.1 (Context-Free Grammar - CFG)

Definition 5.1

A *context-free grammar* is a tuple

$$G = (V, \Sigma, R, S),$$

where

- V is a finite set of *variable* symbols (also called *nonterminals*),
- Σ is a finite set of *terminal* symbols not in V,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *rules* (also called *productions*),
- $S \in V$ is the *start variable* of the grammar.

- A rule $(A, \omega) \in R$ is usually written as $A \to \omega$.
- We often use the notation $\Gamma = V \cup \Sigma$ for the full alphabet of a grammar.

Definition 5.2 (Context-Free Languages)

Definition 5.2

A language $L \subseteq \Sigma^*$ is *context-free* if it can be generated by some context-free grammar.

Theorem 5.1 (CFG is closed under union, concatenation and star operations)

rule	language
$A \to B \mid C$	union $\mathcal{L}(A) = \mathcal{L}(B) \cup \mathcal{L}(C)$
$A \to BC$	concatenation $\mathcal{L}(A) = \mathcal{L}(B)\mathcal{L}(C)$
$A \to AB \mid \varepsilon$ (left recursion) or $A \to BA \mid \varepsilon$ (right recursion)	Kleene star $\mathcal{L}(A) = (\mathcal{L}(B))^*$

- Using these constructions it is easy to prove the following:

Theorem 5.1

The class of context-free languages is closed under union, concatenation and star operations. In other words, if the languages $L_1, L_2 \subseteq \Sigma^\star$ are context-free, then so are $L_1 \cup L_2$, $L_1 L_2$ and $L_1^\star$.

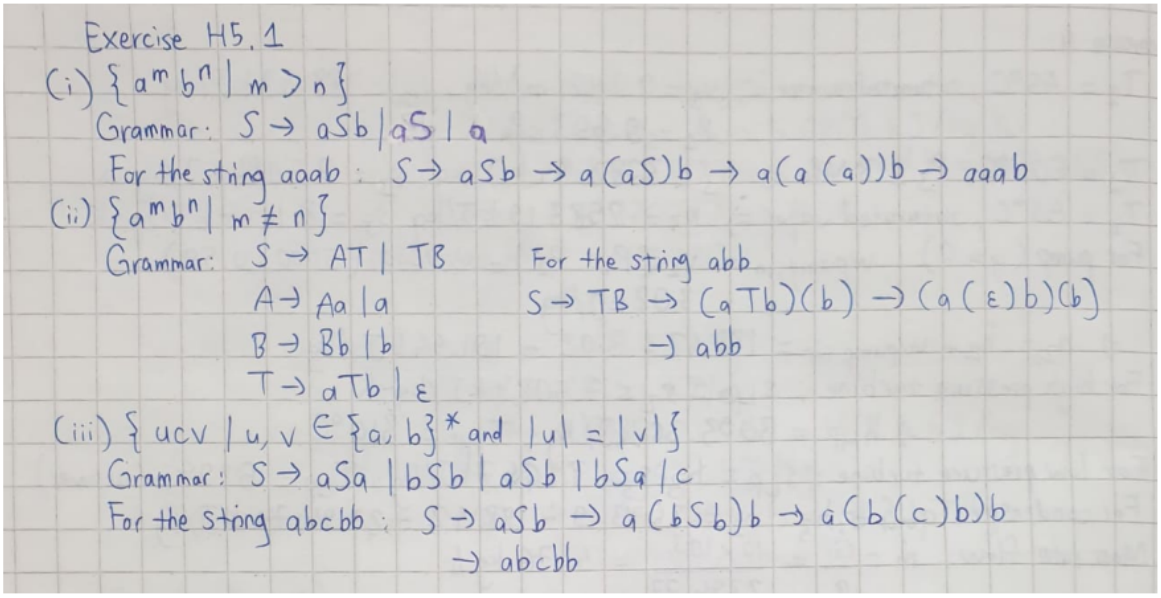
H5.1 Design context-free grammars for the following languages:

(i) $\{a^m b^n \mid m > n\}$.

(ii) $\{a^m b^n \mid m \neq n\}$. Observe that $m \neq n$ if and only if $m < n$ or $m > n$.

(iii) $\{ucv \mid u, v \in \{a, b\}^* \text{ and } |u| = |v|\}$.

Additionally, give a derivation for the string *aaab* using your first grammar, a derivation for *abb* using your second grammar, and a derivation for *abcbb* using your third grammar.



Right and left linear grammars

5.4 Right- and left-linear grammars

- Context-free grammars can generate non-regular languages, e.g. the languages L_{match} and L_{expr}.
- We next show that also every regular language can be generated by some context-free grammar. Thus, context-free languages are a proper super-class of regular languages.
- We say that a context-free grammar is
  - *right-linear* if all the rules in it are of form $A \to aB$ or $A \to \varepsilon$, and
  - *left-linear* if all the rules in it are of form $A \to Ba$ or $A \to \varepsilon$.
- It turns out that both right- and left-linear grammars generate exactly the class of regular languages. (Hence they are also known as *regular grammars*.)
- In the following, we prove this for right-linear grammars.

Theorem 5.2 (regular language can be generated by right linear CFG)

Theorem 5.2

Each regular language can be generated by some right-linear context-free grammar.

Proof

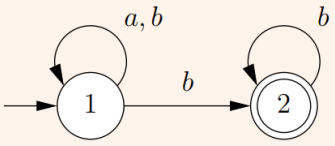
Let L be a regular language over an alphabet Σ and let $M = (Q, \Sigma, \delta, q_0, F)$ be a (deterministic or nondeterministic) finite automaton that recognises it. We construct a right-linear context-free grammar G_M such that $\mathcal{L}(G_M) = \mathcal{L}(M) = L$.

The set of terminal symbols of G_M is Σ and its variable set contains a variable A_q for each state q of M. The start variable of the grammar is A_{q_0} and its rules correspond to the transitions of M:

- For each accept state $q \in F$ of M there is a rule $A_q \to \varepsilon$.
- For each transition $q \xrightarrow{a} q'$ in M (that is, $q' \in \delta(q, a)$) there is a rule $A_q \to aA_{q'}$.

Example:

Automaton:



The corresponding grammar:

$$\begin{aligned} A_1 &\to aA_1 \mid bA_1 \mid bA_2 \\ A_2 &\to \varepsilon \mid bA_2. \end{aligned}$$

Theorem 5.3 (Language generated by right linear CFG is regular)

=> Right linear CFG is equivalent to regular languages from theorem 5.2 and theorem 5.3

Theorem 5.3

If a language can be generated by a right-linear context-free grammar, then it is regular.

Proof

Let $G = (V, \Sigma, R, S)$ be a right-linear context-free grammar. We construct a nondeterministic finite automaton $M_G = (Q, \Sigma, \delta, q_S, F)$ that recognises $\mathcal{L}(G)$ as follows:

- The states of M_G correspond to the variables of G:

$$Q = \{q_A \mid A \in V\}.$$

- The initial state of M_G is the state q_S corresponding to the start variable of G.
- The alphabet of M_G is the set Σ of terminal symbols in G.

H5.3 Design right-linear context-free grammars for the following languages:

(i) $\{w \in \{a,b\}^* \mid w \text{ does not contain } aba \text{ as a substring}\}$;

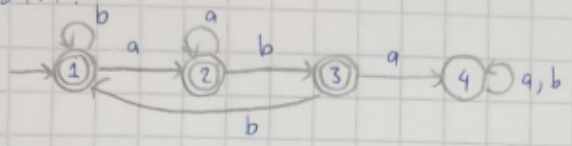
(ii) $\{w \in \{0,1\}^* \mid w \text{ contains an even number of 0's and an odd number of 1's}\}$.

Use the systematic construction presented at Lecture 5. That is, first design a finite automaton for the language in question and then translate the automaton into the corresponding right-linear context-free grammar. In addition to the final solutions, also show the intermediate phases, e.g., the automata that you designed.

Exercise 5.3

(i) $\{w \in \{a,b\}^* \mid w$ does not contain substring $aba\}$

The DFA:



The corresponding context-free grammar (right linear) is

$A_1 \to bA_1 \mid aA_2 \mid \varepsilon$

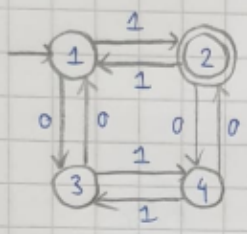
$A_2 \to aA_2 \mid bA_3 \mid \varepsilon$

$A_3 \to bA_1 \mid \varepsilon$

(There is no A_4 because it is a dead state, once entered it will not be accepted. CFG discards all failing state)

(ii) $\{w \in \{0,1\}^* \mid w$ contains even number of 0's and an odd number of 1's$\}$

The DFA:



A_1: even 0, even 1

A_2: even 0, odd 1

A_3: odd 0, even 1

A_4: odd 0, odd 1

and A_2 is accepting state

The corresponding right linear context-free grammar is

$A_1 \to 1A_2 \mid 0A_3$

$A_2 \to 1A_1 \mid 0A_4 \mid \varepsilon$

$A_3 \to 0A_1 \mid 1A_4$

$A_4 \to 0A_2 \mid 1A_3$

Week 6: The Parsing Problem, Parse Trees and Recursive-Descent Parsing

6.1 The parsing problem and parse trees

We want to solve the following problem:

Given a context-free grammar G and a string x. Is $x \in \mathcal{L}(G)$?

A program that solves this problem for a fixed grammar G is called a *parser* for G. In this case only the string x is considered as the input.

- Two canonical ("standard") derivation orders:
- A derivation is a *leftmost* derivation if in each step a rewrite rule is applied to the leftmost available variable. (To emphasise this, we may use the symbol $\underset{\text{lm}}{\Rightarrow}$ instead of $\Rightarrow$.) Derivation (i) on the previous slide is a leftmost derivation.
- *Rightmost derivations* (symbol $\underset{\text{rm}}{\Rightarrow}$) are defined similarly. Derivation (iii) on the previous slide is a rightmost derivation.

Parse Trees

- Let $G = (V, \Sigma, R, S)$ be a context-free grammar.
- A *parse tree* in G is an ordered tree τ [1] where:
  - The nodes in τ are labelled with elements from $V \cup \Sigma \cup \{\varepsilon\}$ so that (i) non-leaf nodes are labeled with elements in V and (ii) the root is labelled with the start variable S.
  - If A is the label of a non-leaf node and $X_1, \ldots, X_k$ are the labels of its (ordered) children, then $A \to X_1 \ldots X_k$ is a production in R.
- The string ("sentential form") *represented* by a parse tree is obtained by listing the labels of its leaf nodes in preorder ("from left to right").

[1] In an ordered tree the children of each node have a fixed left-to-right ordering.

Example:

Recall the grammar G_{expr}:

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow a \mid (E) \end{aligned}$$

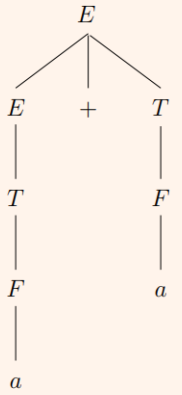
Some derivations of string $a + a$ in the grammar are:

$$\begin{aligned} \text{(i)}\quad & \underline{E} \Rightarrow \underline{E}+T \Rightarrow \underline{T}+T \Rightarrow \underline{F}+T \\ & \Rightarrow a+\underline{T} \Rightarrow a+\underline{F} \Rightarrow a+a \\ \text{(ii)}\quad & \underline{E} \Rightarrow E+\underline{T} \Rightarrow \underline{E}+F \Rightarrow \underline{T}+F \\ & \Rightarrow F+\underline{F} \Rightarrow \underline{F}+a \Rightarrow a+a \\ \text{(iii)}\quad & \underline{E} \Rightarrow E+\underline{T} \Rightarrow E+\underline{F} \Rightarrow \underline{E}+a \\ & \Rightarrow \underline{T}+a \Rightarrow \underline{F}+a \Rightarrow a+a \end{aligned}$$

The underlines denote which non-terminal variable is substituted in which step.

Example:

A parse tree for string $a + a$ in grammar G_{expr}:



A derivation for the string:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \\ &\Rightarrow a+T \Rightarrow a+F \Rightarrow a+a \end{aligned}$$

Lemma 6.1 (Unique leftmost derivation and unique rightmost derivation)

Lemma 6.1

Let $G = (V, \Sigma, R, S)$ be a context-free grammar.

- Each string γ that can be derived in G has a parse tree that represents γ.
- For each parse tree τ that represents a string $x \in L(G)$ there is a unique leftmost derivation $S \underset{\text{lm}}{\Rightarrow}^* x$ and a unique rightmost derivation $S \underset{\text{rm}}{\Rightarrow}^* x$.

Corollary 6.2

Corollary 6.2

Each string $x \in L(G)$ has a leftmost and a rightmost derivation.

That is: parse trees, leftmost derivations and rightmost derivations for words in a language are in one-to-one correspondence.
When solving the parsing problem "Is $x \in L(G)$?", one usually also produces a parse tree (or equivalently a leftmost/rightmost derivation) for x if the answer is "yes".

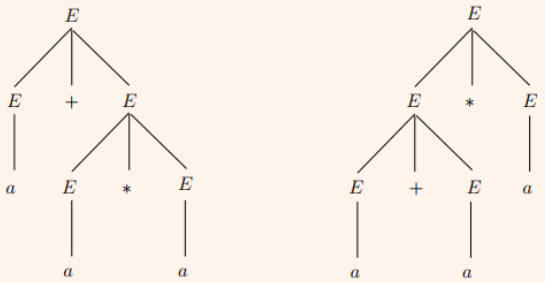
Ambiguous and unambiguous CFG (ambiguity)

Example

Let us consider the following grammar for simple arithmetic expressions:

$$G'_{\text{expr}} = \{E \to E+E,\ E \to E*E,\ E \to a,\ E \to (E)\}.$$

In this grammar, e.g. string $a+a*a$ has *two* parse trees:



- A context-free grammar G is *ambiguous* if some word $x \in L(G)$ has two different parse trees.
- Otherwise the grammar is *unambiguous*.

Inherently ambiguous grammar

- A context-free language for which all the grammars are ambiguous is called an *inherently ambiguous language*.
- As an example, the grammar G'_{expr} is ambiguous while G_{expr} is unambiguous. The language $L_{\text{expr}} = L(G'_{\text{expr}})$ is not inherently ambiguous because it also has an unambiguous grammar G_{expr} generating it.
- On the other hand, e.g. the language

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

  is inherently ambiguous. (The proof of this result is rather complicated and hence omitted here.)

Recursive descent parsing
LL(1) Grammar

Left recursion

- Left recursion is a problem for recursive-descent parsing.

Definition 6.1

A grammar $G = (V, \Sigma, P, S)$ is *left recursive* if one can derive from some variable A with one or more steps the string $A\alpha$, where $\alpha \in (V \cup \Sigma)^\star$.

Example:

The grammar G_{expr}

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to a \mid (E) \end{aligned}$$

is left recursive because $E \Rightarrow E + T$ and $T \Rightarrow T * F$.
This kind of left recursion that occurs in a single step is called *immediate left recursion*.

Example:

Also the grammar

$$\begin{aligned} S &\to ASa \mid b \\ A &\to BB \mid dA \\ B &\to b \mid \varepsilon \end{aligned}$$

is left recursive because e.g. $S \Rightarrow ASa \Rightarrow BBSa \Rightarrow BSa \Rightarrow Sa$.

Lecture 7: The CYK Parsing Algorithm and Chomsky Normal Form, Pushdown Automata, Limitations of Context-Free Languages

Chomsky Normal Form (CNF)

Definition 7.1

A context-free grammar $G = (V, \Sigma, P, S)$ is in *Chomsky normal form* if

1. none of the variables, except possibly the start variable, are nullable,
2. except for the possible production $S \to \varepsilon$, all the productions are of form
$$A \to BC \qquad \text{or} \qquad A \to a$$
where A, B, C are variables and a is a terminal symbol, and
3. the start variable S does not occur on the right-hand side of any production.

- We say that a variable A is *nullable* (in grammar G) if $A \underset{G}{\Rightarrow}^* \varepsilon$.

1. Removing ε-productions

- Let $G = (V, \Sigma, P, S)$ be a context-free grammar.
- Recall: a variable $A \in V - \Sigma$ is *nullable* if $A \underset{G}{\Rightarrow}^* \varepsilon$.

Example:

In the grammar

$$\begin{aligned} S &\to A \mid B \\ A &\to aBa \mid \varepsilon \\ B &\to bAb \mid \varepsilon \end{aligned}$$

all the variables S, A, B are nullable.

Lemma 7.1 (All nullable variables can be eliminated in CFG, except the start variable)

Lemma 7.1

For any context-free grammar G, we can construct an equivalent grammar G' (that is, G' generates the same language as G) in which none of the variables, except possibly the start variable, are nullable.

Proof

Let $G = (V, \Sigma, P, S)$. First, determine all the nullable variables in G:

1. Start by setting

$$\text{NULL} := \{A \in V - \Sigma \mid A \to \varepsilon \text{ is a production in } G\}$$

2. and then expand the NULL set as long as possible with the operation

$$\begin{aligned} \text{NULL} \quad := \quad & \text{NULL} \cup \\ & \{A \in V - \Sigma \mid A \to B_1 \dots B_k \text{ is a production in } G \text{ and} \\ & B_i \in \text{NULL for all } i = 1, \dots, k\}. \end{aligned}$$

After this, replace each production $A \to X_1 \dots X_k$ in G with the set of all productions of the form

$$A \to \alpha_1 \dots \alpha_k, \quad \text{where}$$

$$\alpha_i = \begin{cases} X_i & \text{if } X_i \notin \text{NULL}, \\ X_i \text{ or } \varepsilon & \text{if } X_i \in \text{NULL}. \end{cases}$$

Finally, remove all productions of form $A \to \varepsilon$. If the production $S \to \varepsilon$ is removed, add a new start variable S' and the productions $S' \to S$ and $S' \to \varepsilon$.

Lemma 7.2 (All unit productions can be eliminated in CFG)

2. Removing unit productions

- A production of form $A \to B$, where A and B are variables, is called a *unit production*.

Lemma 7.2

For any context-free grammar G, we can construct an equivalent context-free grammar G' that does not contain any unit productions.

Proof

Let6 $G = (V, \Sigma, P, S)$. First compute the "unit successors" set $F(A)$ for each variable A as follows:

1. First, for each variable A, set

$$F(A) := \{B \mid A \to B \text{ is a production in } G \text{ and } B \text{ is a variable}\}$$

2. and then expand the F-sets as long as possible with the operation

$$F(A) := F(A) \cup \bigcup \{F(B) \mid A \to B \text{ is a production in } G\}.$$

After this:

1. remove all the unit productions from G, and
2. for every variable A, add all possible productions of form $A \to \omega$, where $B \to \omega$ is a non-unit production in G for some $B \in F(A)$.

Theorem 7.3 (All CFG can be converted in to CNF)

Theorem 7.3

For any context-free grammar G, we can construct an equivalent grammar G' that is in Chomsky normal form.

Proof

Let $G = (V, \Sigma, P, S)$ be a CFG.

1. If the start variable S occurs on the right-hand side of any production, introduce a new start variable S' and add the production $S' \to S$.
2. Remove all the ε- and unit productions by using the constructions of Lemmas 3.1 and 3.2.

After this, each production is of one of the following forms:

- $A \to a$, where a is a terminal,
- $A \to X_1 \ldots X_k$, where $k \geq 2$, or
- $S' \to \varepsilon$.

To conclude the transformation, do the following:

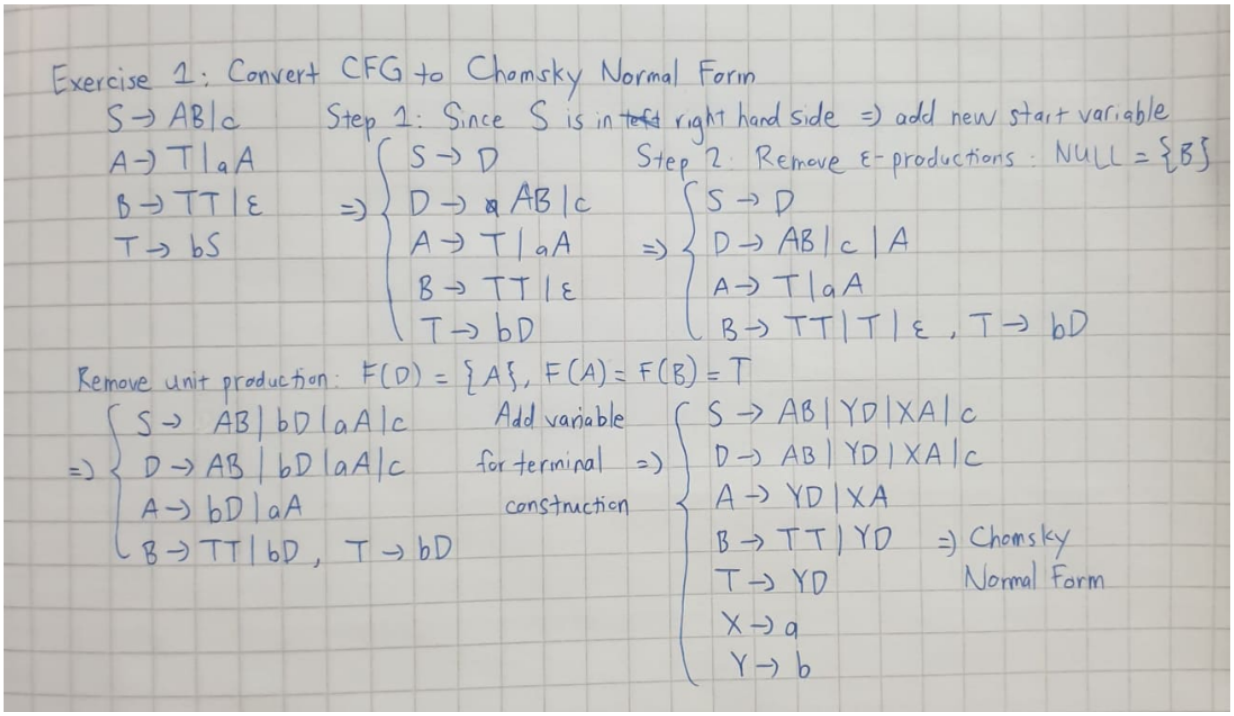
1. For each terminal a, add a new variable C_a and the production $C_a \to a$.
2. In each production of form $A \to X_1 \dots X_k$, $k \geq 2$, first replace each terminal a with the corresponding variable C_a, and then the whole production with the production set

$$\begin{aligned} A &\to X_1 A_1 \\ A_1 &\to X_2 A_2 \\ &\vdots \\ A_{k-2} &\to X_{k-1} X_k, \end{aligned}$$

where $A_1, \dots, A_{k-2}$ are again new variables.

H7.1 Convert the following grammar into Chomsky normal form:

$$\begin{aligned} S &\to AB \mid c \\ A &\to T \mid aA \\ B &\to TT \mid \varepsilon \\ T &\to bS \end{aligned}$$



The CYK Parsing Algorithm

H7.2 Determine, by using the CYK algorithm, whether strings *aba*, *abba* and *bbaa* are generated by the grammar

$$\begin{aligned} S &\rightarrow AB \mid BA \\ A &\rightarrow BA \mid a \\ B &\rightarrow AB \mid b \end{aligned}$$

In the positive cases, give also the respective parse trees.

Exercise 2:

- For string aba, the CYK table is

$N_{i,k}$	1: a	2: b	3: a	→ i
k↓ 1	A	B	A	
2	S, B	S, A		
3	S, A			

$S \rightarrow AB \mid BA$
$A \rightarrow BA \mid a$
$B \rightarrow AB \mid b$

$N_{1,2} = N_{1,1} \times N_{2,1} = \{AB\}$
$N_{2,2} = N_{2,1} \times N_{3,1} = \{BA\}$
$N_{1,3} = (N_{1,1} \times N_{2,2}) \cup (N_{1,2} \times N_{3,1})$
$= \{AS, AA, SA, BA\}$

Start variable is in set $N_{1,3}$ => aba belongs to the language
$S \in N_{1,3}$ as $S \rightarrow BA$ and $B \in N_{1,2}$, $A \in N_{3,1}$
=> $S \rightarrow (AB)A \rightarrow aba$

- For the string abba, the CYK table is

$N_{i,k}$	1: a	2: b	3: b	4: a
k↓ 1	A	B	B	A
2	S, B	ϕ	S, A	
3	ϕ	S, A		
4	S, A			

$N_{1,2} = N_{1,1} \times N_{2,1} = \{AB\}$
$N_{2,2} = N_{2,1} \times N_{3,1} = \{BB\}$
$N_{3,2} = N_{3,1} \times N_{4,1} = \{BA\}$
$N_{1,3} = \{SB, BB\}$
$N_{2,3} = \{BS, BA\}$
$N_{1,4} = \{AS, AA, SS, SA, BS, BA\}$

Start variable is in set $N_{1,4}$ => abba belongs to the language
$S \rightarrow BA \rightarrow (AB)(BA) \rightarrow abba$

- For the string bbaa, the CYK table is

$N_{i,k}$	1: b	2: b	3: a	4: a
k↓ 1	B	B	A	A
2	ϕ	S, A	ϕ	
3	S, A	ϕ		
4	ϕ			

Start variable is not in $N_{1,4}$
=> bbaa doesn't belong to the language

Example:

Consider the grammar G in Chomsky normal form:

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a \end{aligned}$$

Executing the CYK algorithm on target string $x = baaba$:

		$i \to$				
	$N_{i,k}$	$1:b$	$2:a$	$3:a$	$4:b$	$5:a$
	1	B	A,C	A,C	B	A,C
	2					
$k\downarrow$	3					
	4					
	5					

Similarly for all $N_{i,1}$, where $1 \leq i \leq 5$

Example:

Consider the grammar G in Chomsky normal form:

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a \end{aligned}$$

Executing the CYK algorithm on target string $x = baaba$:

		$i \to$				
	$N_{i,k}$	$1:b$	$2:a$	$3:a$	$4:b$	$5:a$
	1	B	A,C	A,C	B	A,C
	2	S,A	B			
$k\downarrow$	3					
	4					
	5					

$N_{2,2} = \{B\}$ as $B \to CC$ is the only production whose right-hand side is in the set $N_{2,1} \times N_{3,1} = \{AA, AC, CA, CC\}$.

Example:

Consider the grammar G in Chomsky normal form:

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a \end{aligned}$$

Executing the CYK algorithm on target string $x = baaba$:

		$i \to$				
	$N_{i,k}$	$1:b$	$2:a$	$3:a$	$4:b$	$5:a$
	1	B	A,C	A,C	B	A,C
	2	S,A	B	S,C	S,A	
$k\downarrow$	3	$\emptyset$	B			
	4					
	5					

$N_{2,3} = \{B\}$ as $B \to CC$ is the only production with right-hand side in the set $(N_{2,1} \times N_{3,2}) \cup (N_{2,2} \times N_{4,1}) = \{AS, AC, CS, CC, BB\}$.

Example:

Consider the grammar G in Chomsky normal form:

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a \end{aligned}$$

Executing the CYK algorithm on target string $x = baaba$:

		$i \to$				
	$N_{i,k}$	$1:b$	$2:a$	$3:a$	$4:b$	$5:a$
	1	B	A,C	A,C	B	A,C
	2	S,A	B	S,C	S,A	
$k\downarrow$	3	$\emptyset$	B	B		
	4	$\emptyset$				
	5					

$N_{1,4} = \emptyset$ as there are no productions with right-hand sides in the set $(N_{1,1} \times N_{2,3}) \cup (N_{1,2} \times N_{3,2}) \cup (N_{1,3} \times N_{4,1}) = \{BB, SS, SC, AS, AC\}$.

Example:

Consider the grammar G in Chomsky normal form:

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a \end{aligned}$$

Executing the CYK algorithm on target string $x = baaba$:

		$i \to$				
	$N_{i,k}$	$1:b$	$2:a$	$3:a$	$4:b$	$5:a$
	1	B	A,C	A,C	B	A,C
	2	S,A	B	S,C	S,A	
$k\downarrow$	3	$\emptyset$	B	B		
	4	$\emptyset$	S,A,C			
	5	S,A,C				

As the start variable S belongs to the set $N_{1,5}$, we deduce that x belongs to the language $\mathcal{L}(G)$.

Definition 7.2 (Push-down Automata - PDA)

Definition 7.2

A *pushdown automaton* is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F),$$

where

- Q is the finite set of *states*,
- Σ is the finite *input alphabet*,
- Γ is the finite *stack alphabet*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$ is the (set-valued) *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.

- The interpretation for a value

$$\delta(q,\sigma,\gamma) = \{(q_1,\gamma_1),\ldots,(q_k,\gamma_k)\}$$

of the transition function is that, when in the state q and reading input symbol σ and stack symbol γ, the automaton can move to any one of the states $q_1,\ldots,q_k$ and replace the topmost symbol on the stack with the corresponding symbol $\gamma_1,\ldots,\gamma_k$.
- Note that in pushdown automata, nondeterminism is allowed already in the basic definition!
- If $\sigma = \varepsilon$, the automaton can take the transition without reading the next input symbol.
- If $\gamma = \varepsilon$, the automaton does not read (and remove) the topmost symbol on the stack, but the new symbol is added on the stack on top of the previous one (the "push" operation).
- If the stack symbol read is $\gamma \neq \varepsilon$ and the stack symbol to be written is $\gamma_i = \varepsilon$, the topmost symbol of the stack is removed (the "pop" operation).

- A *configuration* of the automaton is a triple $(q,w,\alpha) \in Q \times \Sigma^* \times \Gamma^*$.
- In particular, the *start configuration* of the automaton on input x is the triple (q_0,x,ε).
- Intuition: in configuration (q,w,α) the automaton is in state q, the input string that has not yet been processed is w, and the stack contains (from top to bottom) the symbols in α.
- A configuration (q,w,α) *leads in one step* to a configuration (q',w',α'), denoted as

$$(q,w,\alpha) \underset{M}{\vdash} (q',w',\alpha'),$$

if one can write $w = \sigma w'$, $\alpha = \gamma\beta$, $\alpha' = \gamma'\beta$ ($|\sigma|,|\gamma|,|\gamma'| \leq 1$), so that

$$(q',\gamma') \in \delta(q,\sigma,\gamma).$$

- A configuration (q,w,α) *leads* to a configuration (q',w',α'), denoted as

$$(q,w,\alpha) \underset{M}{\vdash^*} (q',w',\alpha'),$$

if there is a sequence (q_0,w_0,α_0), (q_1,w_1,α_1), …, (q_n,w_n,α_n), $n \geq 0$, of configurations such that

$$(q,w,\alpha) = (q_0,w_0,\alpha_0) \underset{M}{\vdash} (q_1,w_1,\alpha_1) \underset{M}{\vdash} \cdots \underset{M}{\vdash} (q_n,w_n,\alpha_n) = (q',w',\alpha').$$

- A pushdown automaton M *accepts* a string $x \in \Sigma^*$ if

$$(q_0,x,\varepsilon) \underset{M}{\vdash^*} (q_f,\varepsilon,\alpha) \quad \text{for some } q_f \in F \text{ and } \alpha \in \Gamma^*,$$

meaning that it *can be* in an accept state when the whole input has been processed; otherwise, M *rejects* x.
- The language *recognised* by the automaton M is

$$\mathcal{L}(M) = \{x \in \Sigma^* \mid (q_0,x,\varepsilon) \underset{M}{\vdash^*} (q_f,\varepsilon,\alpha) \text{ for some } q_f \in F \text{ and } \alpha \in \Gamma^*\}.$$

H7.3 Design pushdown automata recognising the following languages.

(a) The language:

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

(*Hint:* Have a look at the example automaton in Section 7.4 of the lecture slides.)

(b) The language generated by the grammar

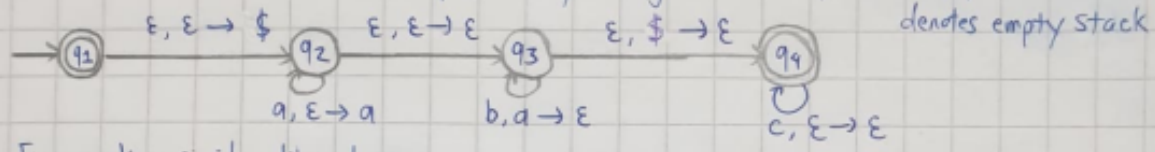
$$S \to (S) \mid S, S \mid a$$

Exercise 3 : Design PDA recognizing the language

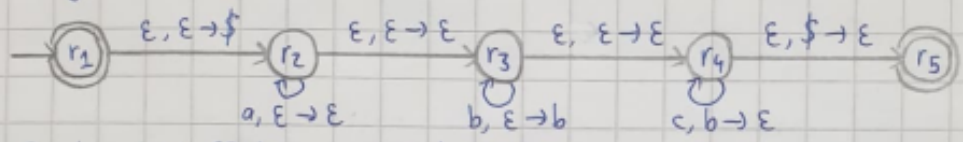
a) The language

$\{a^i b^j c^k \mid i = j$ or $j = k$ or both $\}$

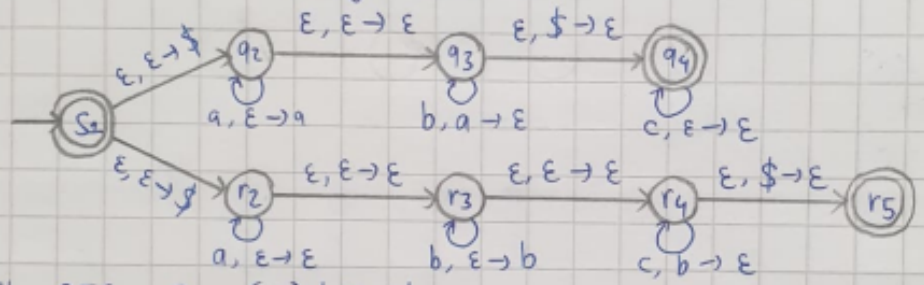
- For $i = j$: as PDA reads a , it adds a to the stack. Then when it reads b, it pops a from the stack until stack is empty. Finally reading c without push nor pop , where $ denotes empty stack



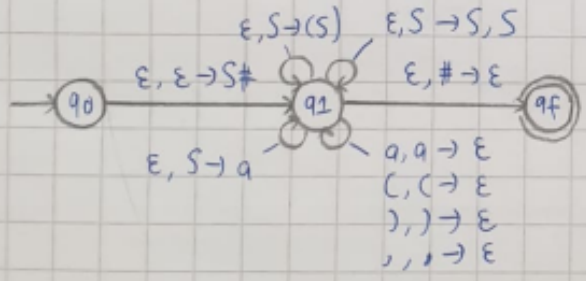
- For $j = k$: similar like above



=) Combine both PDA, we got final result



b) The CFG : $S \to (S) \mid S, S \mid a$



Example:

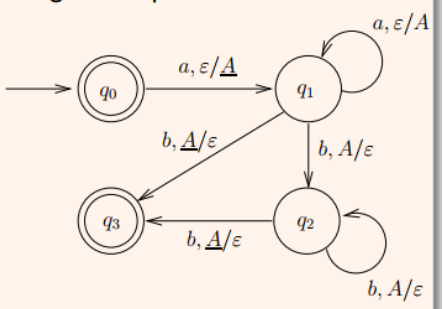
A pushdown automaton for the language $\{a^k b^k \mid k \geq 0\}$:

$$M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{A, \underline{A}\}, \delta, q_0, \{q_0, q_3\}),$$

where

$$\begin{aligned}
\delta(q_0, a, \varepsilon) &= \{(q_1, \underline{A})\}, \\
\delta(q_1, a, \varepsilon) &= \{(q_1, A)\}, \\
\delta(q_1, b, A) &= \{(q_2, \varepsilon)\}, \\
\delta(q_1, b, \underline{A}) &= \{(q_3, \varepsilon)\}, \\
\delta(q_2, b, A) &= \{(q_2, \varepsilon)\}, \\
\delta(q_2, b, \underline{A}) &= \{(q_3, \varepsilon)\}, \\
\delta(q, \sigma, \gamma) &= \emptyset \quad \text{for other } (q, \sigma, \gamma).
\end{aligned}$$

Diagram representation:



A computation of the automaton on input $aabb$:

$$\begin{aligned}
(q_0, aabb, \varepsilon) &\vdash (q_1, abb, \underline{A}) \vdash (q_1, bb, A\underline{A}) \\
&\vdash (q_2, b, \underline{A}) \vdash (q_3, \varepsilon, \varepsilon).
\end{aligned}$$

Since $q_3 \in F = \{q_0, q_3\}$, we have that $aabb \in \mathcal{L}(M)$.

Theorem 7.4 (Context free lang ⇔ recognized by nondeterministic pushdown automata - NPDA)

Theorem 7.4

A language is context-free if and only if it can be recognised by a (nondeterministic) pushdown automaton.

- The proof of the direction "if L can be recognised by a (nondeterministic) pushdown automaton, then L is context-free" is omitted here but can be found in Sipser's book (Lemma 2.27).
- Also the detailed proof of the direction "if L is context-free, then L can be recognised by a (nondeterministic) pushdown automaton" is omitted. The main idea, however, is to have the pushdown automaton M_G corresponding to a CFG G try out leftmost derivations $S \underset{\text{lm}}{\Rightarrow}^* x$ of G on its stack as follows:
  - If the topmost symbol on the stack is a variable A, the machine chooses some rule $A \to \omega$ and pushes the symbols corresponding to ω on its stack.
  - If the topmost symbol on the stack is a terminal symbol a, the machine tries to match it with the current input symbol.

- A pushdown automaton M is *deterministic*, if for each configuration (q, w, α) there is at most one configuration (q', w', α') such that

$$(q, w, \alpha) \underset{M}{\vdash} (q', w', \alpha').$$

- Unlike in the case of finite automata, *nondeterministic pushdown automata are properly more powerful than deterministic ones*.
- As an example, the language $\{ww^R \mid w \in \{a, b\}^\star\}$ can be recognised by a nondeterministic pushdown automaton but not by any deterministic one. (Proof omitted.)
- A context-free language is *deterministic* if it can be recognised by some deterministic pushdown automaton.
- Deterministic context-free languages can be parsed in time $O(n)$; in general, parsing context-free languages with currently known algorithms takes almost time $O(n^3)$.

Pumping lemma for CFG

7.6 A pumping lemma for context-free languages

- There is a "pumping lemma" for context-free languages as well.
- However, now the string must be "pumped" synchronously in two parts.

Lemma 7.5 (The "uvwxy-lemma")

Let L be a context-free language. Then for some $n \geq 1$, every string $z \in L$ with $|z| \geq n$ can be divided in five parts $z = uvwxy$ in such a way that

1. $|vx| \geq 1$,
2. $|vwx| \leq n$, and
3. $uv^i wx^i y \in L$ for all $i = 0, 1, 2, \ldots$.

Example:

Consider the language

$$L = \{a^k b^k c^k \mid k \geq 0\}.$$

Suppose that L would be context-free and consider the string $z = a^n b^n c^n \in L$, where n is the length parameter in Lemma 7.5.

By the Lemma, we should now be able to divide z into five parts

$$z = uvwxy, \qquad |vx| \geq 1, \qquad |vwx| \leq n.$$

Due to the second condition above, the substring vx contains at least one symbol: a, b, or c. But due to the third condition, the substring vx cannot contain every symbol a, b and c. Therefore, the string $uv^0wx^0y = uwy$ cannot contain equal numbers of all symbols a, b and c, and thus does not belong to language L. Therefore, our assumption is wrong and L is not context-free.

Week 8: Turing Machines

Definition 8.1 (Turing machines - TM)

Definition 8.1

A *Turing machine*[a] is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

- Q is the finite set of *states*,
- Σ is the finite *input alphabet*,
- $\Gamma \supseteq \Sigma$ is the finite *tape alphabet* (we assume $\triangleright, \triangleleft \notin \Gamma$),
- $\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times (\Gamma \cup \{\triangleright, \triangleleft\}) \to Q \times (\Gamma \cup \{\triangleright, \triangleleft\}) \times \{L, R\}$ is the *transition function*,
- $q_0 \in Q$ is the *start state* ($q_0 \neq q_{\text{acc}}$ and $q_0 \neq q_{\text{rej}}$),
- $q_{\text{acc}} \in Q$ is the *accept state*, and
- $q_{\text{rej}} \in Q$ is the *reject state* ($q_{\text{rej}} \neq q_{\text{acc}}$).

The interpretation for a value

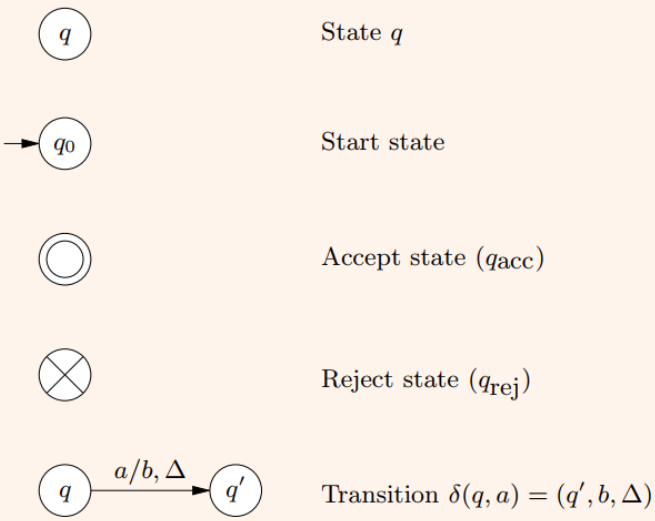
$$\delta(q,a) = (q',b,\Delta)$$

of the transition function is that when in state q and reading symbol a on the tape, the machine:

- moves to state q',
- writes symbol b at the same position on the tape, and
- moves the tape head one position in direction Δ ($L \sim$ "left", $R \sim$ "right").

- The transition function is not defined for the states q_{acc} and q_{rej}. When in either of these states, the machine *halts* immediately.
- For all transitions $\delta(q,a) = (q',b,\Delta)$ it is required that:
  - if $a = \triangleright$, then $b = \triangleright$ and $\Delta = R$
    That is, the start-of-tape symbol is never overwritten and the machine cannot move left beyond that symbol (i.e., off the tape).
  - $b = \triangleright$ is allowed only if $a = \triangleright$
    In other words, new start-of-tape symbols cannot be written.
  - $b = \triangleleft$ is allowed only if $a = \triangleleft$ and $\Delta = L$
    The machine does not explicitly write new end-of-tape symbols; they are introduced automatically when the machine moves past (and overwrites) the current end-of-tape symbol.

The notations used in the state diagram representation:



Example:

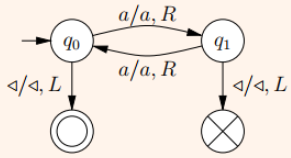
The (regular) language $\{a^{2k} \mid k \geq 0\}$ can be recognised with the Turing machine

$$M = (\{q_0, q_1, q_{\text{acc}}, q_{\text{rej}}\}, \{a\}, \{a\}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

$$\begin{aligned}
\delta(q_0, a) &= (q_1, a, R), \\
\delta(q_1, a) &= (q_0, a, R), \\
\delta(q_0, \triangleleft) &= (q_{\text{acc}}, \triangleleft, L), \\
\delta(q_1, \triangleleft) &= (q_{\text{rej}}, \triangleleft, L).
\end{aligned}$$

State diagram representation:



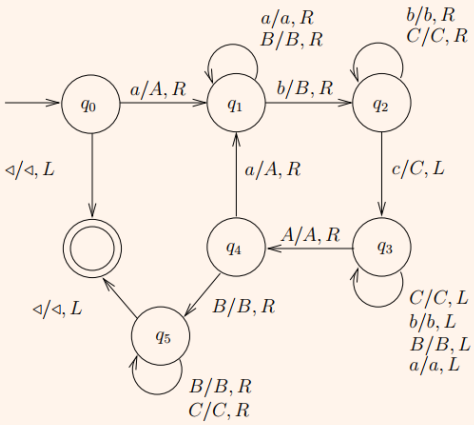
The computation of machine M on input aaa:

$$\begin{aligned}
(q_0, \underline{a}aa) &\underset{M}{\vdash} (q_1, a\underline{a}a) \underset{M}{\vdash} (q_0, aa\underline{a}) \\
&\underset{M}{\vdash} (q_1, aaa\underline{\varepsilon}) \underset{M}{\vdash} (q_{\text{rej}}, aa\underline{a}).
\end{aligned}$$

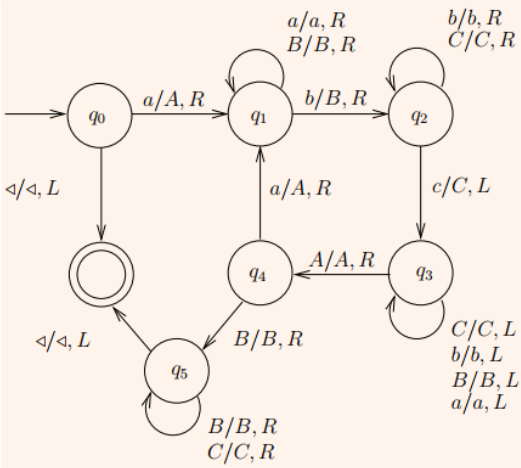
The machine halts in state q_{rej} and thus $aaa \notin \mathcal{L}(M)$.

Example:

A Turing machine recognising the (non-context-free) language $\{a^k b^k c^k \mid k \geq 0\}$:

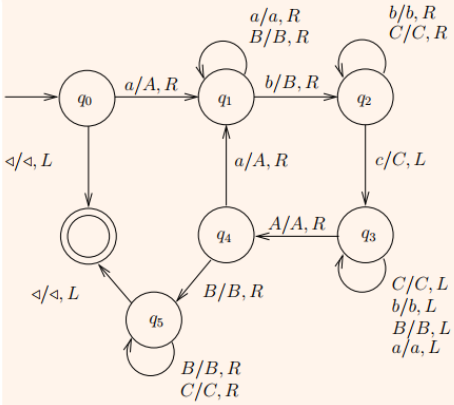


For the sake of clarity, the reject state is not drawn here, but again one interpretes that all the "missing transitions" implicitly lead to the reject state.



The computation of the machine on input $aabcbc$:

$$(q_0, \underline{a}abcbc) \vdash (q_1, A\underline{a}bcbc) \vdash (q_1, Aa\underline{b}cbc) \vdash (q_2, AaB\underline{c}bc) \vdash (q_3, Aa\underline{B}Cbc) \vdash (q_3, A\underline{a}BCbc) \vdash (q_3, \underline{A}aBCbc) \vdash (q_4, A\underline{a}BCbc) \vdash (q_1, AA\underline{B}Cbc) \vdash (q_1, AAB\underline{C}bc) \vdash (q_{\text{rej}}, AABC\underline{b}c).$$



The computation of the machine on input $aabbcc$:

$$(q_0, \underline{a}abbcc) \vdash (q_1, A\underline{a}bbcc) \vdash (q_1, Aa\underline{b}bcc) \vdash (q_2, AaB\underline{b}cc) \vdash (q_2, AaBb\underline{c}c) \vdash (q_3, AaB\underline{b}Cc) \vdash (q_3, Aa\underline{B}bCc) \vdash (q_3, A\underline{a}BbCc) \vdash (q_3, \underline{A}aBbCc) \vdash (q_4, A\underline{a}BbCc) \vdash (q_1, AA\underline{B}bCc) \vdash (q_1, AAB\underline{b}Cc) \vdash (q_2, AABB\underline{C}c) \vdash (q_2, AABBC\underline{c}) \vdash (q_3, AABB\underline{C}C) \vdash (q_3, AAB\underline{B}CC) \vdash (q_3, AA\underline{B}BCC) \vdash (q_3, A\underline{A}BBCC) \vdash (q_4, AA\underline{B}BCC) \vdash (q_5, AAB\underline{B}CC) \vdash (q_5, AABB\underline{C}C) \vdash (q_5, AABBC\underline{C}) \vdash (q_5, AABBCC\underline{\varepsilon}) \vdash (q_{\text{acc}}, AABBC\underline{C}).$$

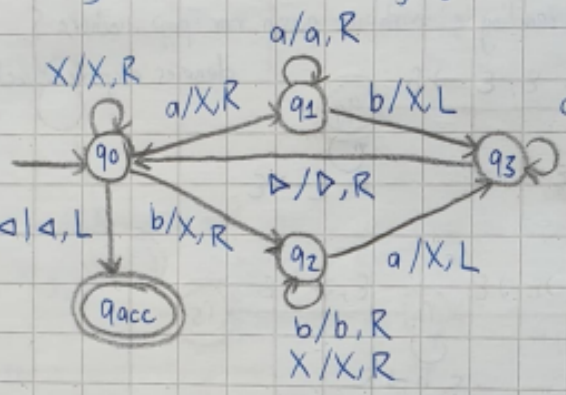
H8.1 Design a deterministic single-tape Turing machine that decides (i.e. recognises and halts on every input) the language

$$\{w \in \{a,b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s}\}.$$

Show the accepting computation sequence ("run") of your machine on input *aabb* and the rejecting sequence on input *baa*.

We have to scan the input from left to right.
Convert first 'a' and first 'b' in the scanning to 'X', then in the second turn convert second 'a' and second 'b' to 'X' and so on. We have to repeat the process until we convert all a's and b's to 'X'. Characters scanned in between 'a' and 'b' will not be changed.
=> a and b are converted to X in pairs. The input is rejected when it reads end of input symbol at state q1 or q2

Exercise 1:
Design deterministic single-tape Turing Machine that recognizes the language
$\{w \in \{a,b\}^* \mid w$ contains equal a and b $\}$
The Turing Machine for this language would be:



▷: start of tape
◁: end of input
q_3: rewinding back to start of tape at q_0

- Accepting computation sequence on input aabb (← start rewinding)
  $(q_0, \underline{a}abb) \vdash (q_1, X\underline{a}bb) \vdash (q_1, Xa\underline{b}b) \vdash (q_3, X\underline{a}Xb)$
  $\vdash (q_3, \underline{X}aXb) \vdash (q_3, \underline{\triangleright}XaXb) \vdash (q_0, \underline{X}aXb)$ (end rewinding)
  $\vdash (q_0, X\underline{a}Xb) \vdash (q_1, XX\underline{X}b) \vdash (q_1, XX\underline{X}b) \vdash (q_3, XX\underline{X}X)$
  $\vdash$ rewinding $\vdash (q_0, \underline{X}XXX)$ ← just move left for scanning X
  $\vdash (q_0, XXX\underline{\triangleleft}) \vdash q_{acc}$
- Rejecting computation sequence on input bba
  $(q_0, \underline{b}aa) \vdash (q_2, X\underline{a}a) \vdash (q_3, \underline{X}Xa) \vdash (q_3, \triangleright XXa)$
  $\vdash (q_0, \underline{X}Xa) \vdash (q_0, X\underline{X}a) \vdash (q_0, XX\underline{a}) \vdash (q_1, XXX\triangleleft)$
  $\vdash q_{rej}$

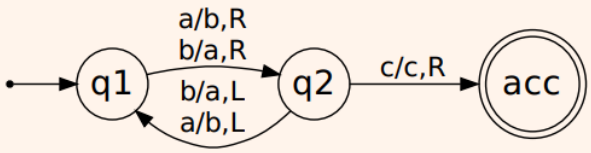
TM does not necessarily halts on strings that doesnt belong to the language it recognized, such as in an infinite loop

Note

The definition of "language recognised by a machine" *does not require that the machine halts* on strings that do not belong to the language.

Example:

A Turing machine that enters an infinite loop on some inputs:



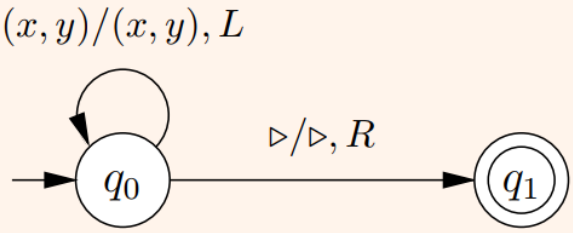
The computation on input abc:

$(q_1, \underline{a}bc) \vdash (q_2, b\underline{b}c) \vdash (q_1, \underline{b}ac) \vdash (q_2, a\underline{a}c) \vdash$
$(q_1, \underline{a}bc) \vdash (q_2, b\underline{b}c) \vdash (q_1, \underline{b}ac) \vdash (q_2, a\underline{a}c) \vdash \ldots$

Rewinding the tape

Example:

A 2-track Turing machine `rewind` that rewinds the tape head to the beginning of the tape:



(The notation $(x,y)/(x,y),L$ is here a shorthand meant to cover all the transitions that can be obtained by replacing the variables x and y with some tape alphabet symbols.)

Multi-track Turing Machine

Lemma 8.1 (Language recognized by multitrack TM can also be recognized by standard TM)

Lemma 8.1

If a language L can be recognised with a k-track Turing machine, then it can be recognised with a standard Turing machine as well.

Proof

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a k-track Turing machine recognising the language L. An equivalent standard Turing machine $\widehat{M}$ can be constructed as follows:

$$\widehat{M} = (\widehat{Q}, \Sigma, \hat{\Gamma}, \hat{\delta}, \hat{q}_0, q_{\text{acc}}, q_{\text{rej}}),$$

where $\widehat{Q} = Q \cup \{\hat{q}_0, \hat{q}_1, \hat{q}_2\}$, $\hat{\Gamma} = \Sigma \cup \Gamma^k$ and for all $q \in Q$ we have

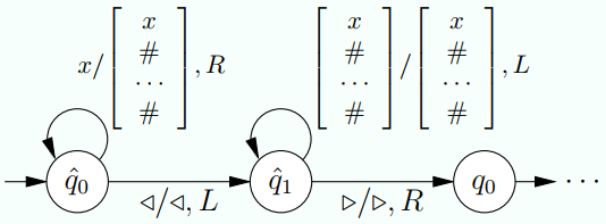
$$\hat{\delta}(q, \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}) = (q', \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix}, \Delta),$$

$$\text{when} \quad \delta(q, (a_1, \ldots, a_k)) = (q', (b_1, \ldots, b_k), \Delta).$$

In the beginning of its computation, the machine $\widehat{M}$ must "lift" the input string to the (simulating) first track, meaning that it replaces the input $a_1 a_2 \ldots a_n$ with

$$\begin{bmatrix} a_1 \\ \# \\ \vdots \\ \# \end{bmatrix} \begin{bmatrix} a_2 \\ \# \\ \vdots \\ \# \end{bmatrix} \cdots \begin{bmatrix} a_n \\ \# \\ \vdots \\ \# \end{bmatrix}.$$

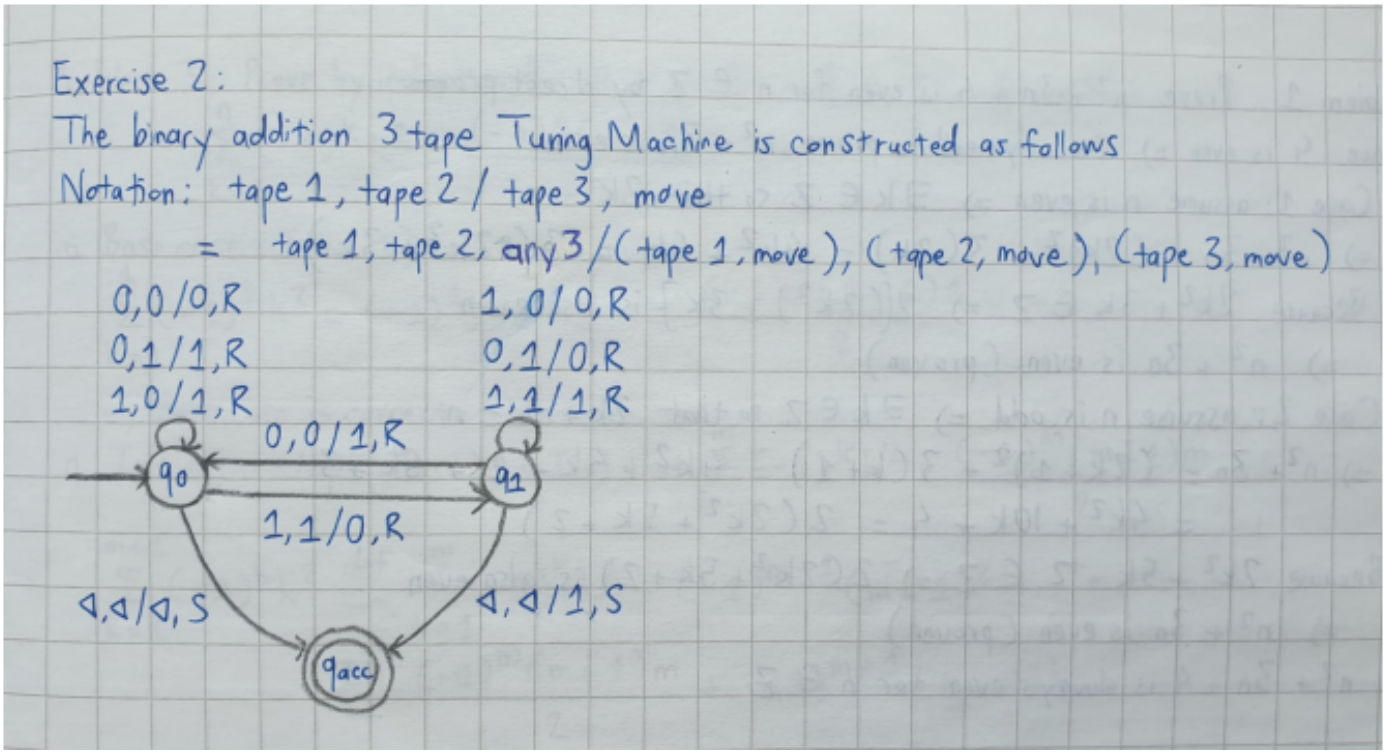
For this, the transition function of $\widehat{M}$ includes, in addition to the transitions copied from M, a small "preprocessor" sub-machine



H8.2 Design a three-tape Turing machine ADD that functions as follows. The machine gets as input on the tapes 1 and 2 two binary numbers written in reverse, i.e. with their least significant bits first. It then computes on the tape 3 the sum of the two given numbers in the same notation. For simplicity, you may assume that the input numbers are of the same length, i.e. that the possibly shorter one is padded with leading zeros. Thus, for instance, the calculation $7 + 11 = 18$ is represented as:

```
1110
1101
01001
```

You may assume that a tape head of the machine can also stay stationary in a transition (the move direction "S").

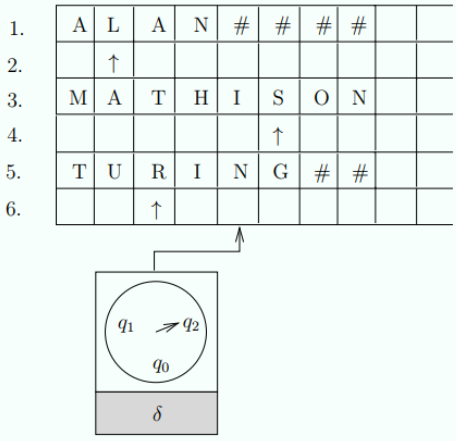


Lemma 8.2 (Language recognized by multi-tape TM can also be recognized by standard TM)

Lemma 8.2

If a language L can be recognised with a k-tape Turing machine, then it can be recognised with a standard Turing machine as well.

Proof



Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

be a k-tape Turing machine recognising the language L. We can simulate it with a $2k$-track Turing machine $\widehat{M}$ as follows. The odd tracks $1, 3, 5, \ldots, 2k-1$ of $\widehat{M}$ correspond to the tapes $1, 2, \ldots, k$ of M and for each odd track, the following even track contains exactly one $\uparrow$ symbol that indicates the tape head position on the tape of the odd track.

- In the beginning, the input string is placed on the first track as usual, and in its first move $\widehat{M}$ writes the $\uparrow$ symbols to the first positions of the even tracks.
- After this, $\widehat{M}$ operates by "sweeping" across the tape forwards and backwards.
- On each forward sweep from the beginning to the end, $\widehat{M}$ collects information about which symbols are at the positions indicated by the $\uparrow$ symbols, i.e., at the tape head positions of the simulated machine M.
- Based on this information, $\widehat{M}$ then performs a backward sweep to the beginning and makes the changes on its multitrack tape (writes tape symbols, moves tape head markers $\uparrow$) that correspond to the changes made by a single transition of the simulated machine M.

The multitrack machine $\widehat{M}$ can then be simulated with a standard Turing machine, as presented in Lemma 8.1.

Nondeterministic Turing Machine

- Formally, a *nondeterministic Turing machine* is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where the other components are as in the standard model but the transition function is of form:

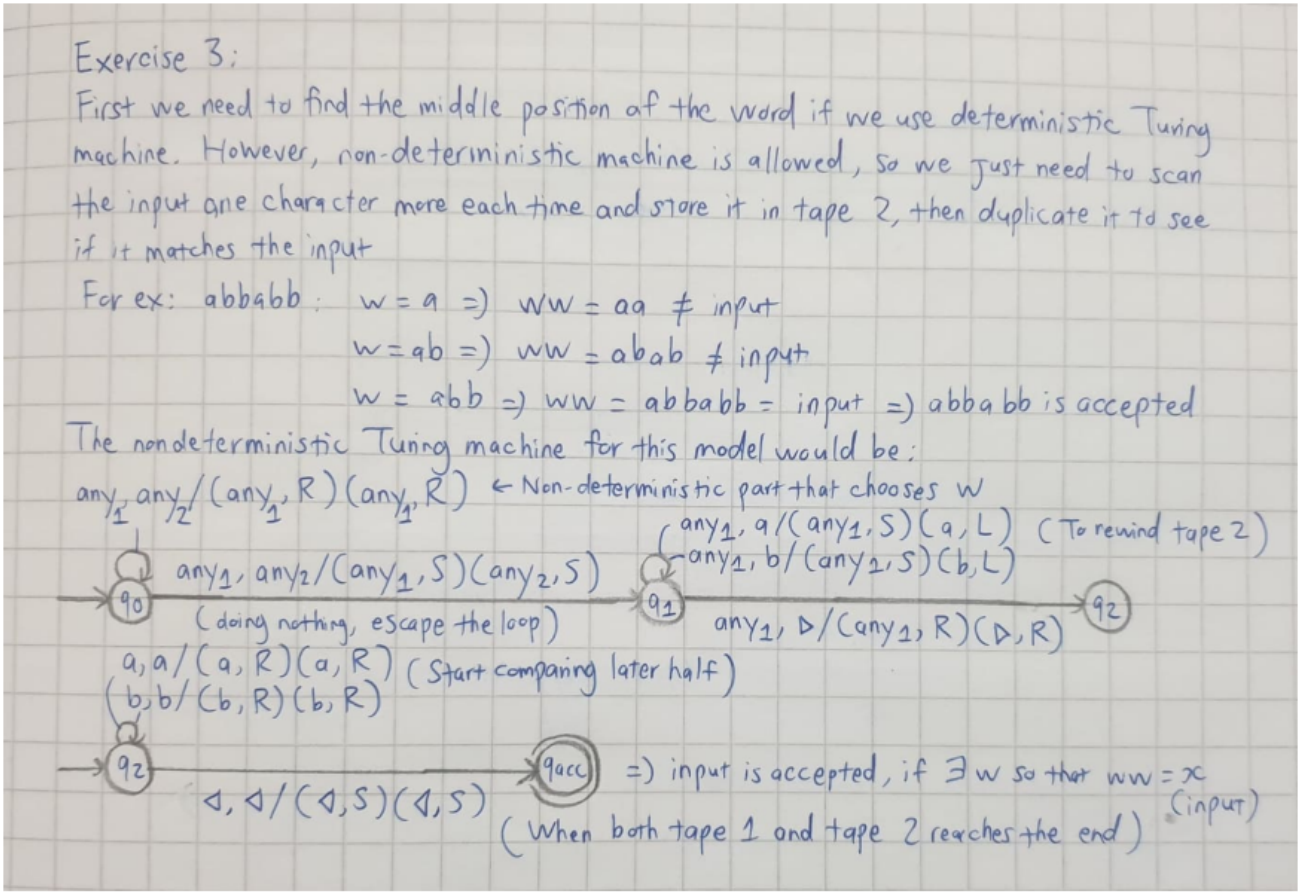
$$\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times (\Gamma \cup \{\triangleright, \triangleleft\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\triangleright, \triangleleft\}) \times \{L, R\}).$$

- The interpretation of a value

$$\delta(q, a) = \{(q_1, b_1, \Delta_1), \ldots, (q_k, b_k, \Delta_k)\}$$

of the transition function is that, when in state q and reading tape symbol a, the machine can act according to *some* triple (q_i, b_i, Δ_i) in the list.

H8.3 Design a two-tape nondeterministic Turing machine that recognises the language $\{ww \mid w \in \{a, b\}^*\}$. You may assume that a tape head of the machine can also stay stationary in a transition (the move direction "S").



Theorem 8.3 (Language recognized by NTM can be recognized by a standard TM)

Theorem 8.3

If a language L can be recognised with a nondeterministic Turing machine, then it can be recognised with a standard deterministic Turing machine as well.

Proof (idea)

- Let
$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$
be a nondeterministic Turing machine recognising the language L.
- One can simulate M with a 3-tape deterministic machine $\widehat{M}$ that systematically explores all the computations of M until it finds a computation that ends in an accepting configuration — if such a computation exists.
- The 3-tape machine $\widehat{M}$ can then be transformed into a standard deterministic machine as presented in Lemmas 8.1 and 8.2.

Week 9: Decidability and Undecidability

Definition 9.1

9.1 Turing-recognisable and Turing-decidable languages

- The *Church-Turing thesis:* Any (strong enough) computing model $\equiv$ Turing machines.
- *Computability theory:* The study of what can be, and especially what cannot be computed with Turing machines ($\equiv$ computer programs).
- *Important distinction:* Machines (programs) that always halt and those that don't.

Definition 9.1

A Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

is *total* if it halts on all input strings. A language A is

- *Turing-recognisable* (historically also called *recursively enumerable*) if it can be recognised with some Turing machine.
- *Turing-decidable* (historically also called *recursive*) if it can be recognised by some *total* Turing machine.

- The decision problem Π is
  - *decidable* if its language A_Π is Turing-decidable, and
  - *semi-decidable* if A_Π is Turing-recognisable.
  - A problem that is not decidable is called *undecidable*.
    (Note: An undecidable problem may still be semi-decidable.)
- In other words, a decision problem is (i) decidable if it has an algorithm that solves it correctly and always terminates, and (ii) semi-decidable if it has an algorithm that solves it correctly but may not terminate on some "no" instances.
- In the following, we will use this terminology also for languages, viz. decidable $\equiv$ Turing-decidable, semi-decidable $\equiv$ Turing-recognisable.

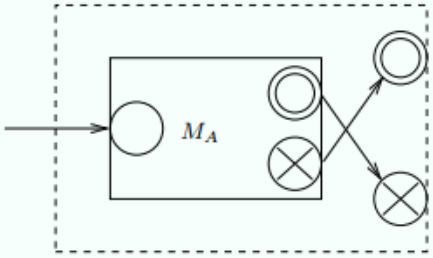
Lemma 9.1 (Union, intersection, complement of decidable langs are also decidable langs)

Lemma 9.1

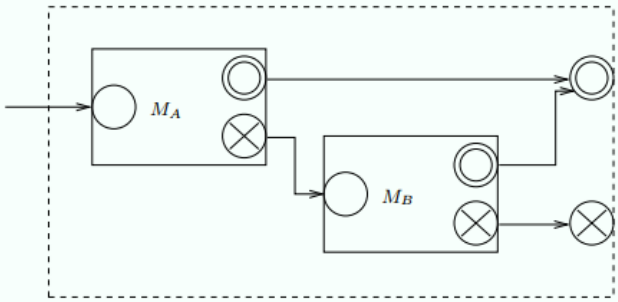
Let $A, B \subseteq \Sigma^*$ be decidable languages. Then $\bar{A} = \Sigma^* - A$, $A \cup B$, and $A \cap B$ are decidable languages as well.

Proof

(i) Let M_A be a total Turing machine with $\mathcal{L}(M_A) = A$. We obtain a total Turing machine $M_{\bar{A}}$ recognising $\bar{A}$ simply by swapping the accept and reject states of M_A:



(ii) Let M_A and M_B be total Turing machines with $\mathcal{L}(M_A) = A$ and $\mathcal{L}(M_B) = B$. We obtain a total Turing machine M recognising the language $A \cup B$ by sequential composition of M_A and M_B: if M_A accepts the input, then M also accepts; if M_A rejects the input, then execute M_B on the input string.



(iii) $A \cap B = \overline{\bar{A} \cup \bar{B}}$.

Lemma 9.2 (Union and intersection of semi-decidable langs are also semi-decidable)

Lemma 9.2

Let $A, B \subseteq \Sigma^*$ be semi-decidable languages. Then $A \cup B$ and $A \cap B$ are semi-decidable languages, too.

Proof

$A \cap B$ as in Lemma 9.1 (ii), and $A \cup B$ with a construction similar to that in Lemma 9.3 (left as an exercise).

Lemma 9.3 (Lang A is decidable if A and complement of A are both semi-decidable)

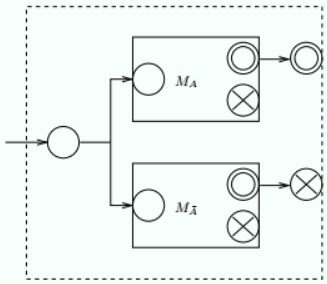
Lemma 9.3

A language $A \subseteq \Sigma^*$ is decidable if and only if both languages A and $\bar{A}$ are semi-decidable.

Proof

By Lemma 9.1(i), if A is decidable, then $\bar{A}$ is also decidable and hence both A and $\bar{A}$ are semi-decidable as well.

We next show that if A and $\bar{A}$ are both semi-decidable, then A is decidable.



Let M_A and $M_{\bar{A}}$ be Turing machines recognising the languages A and $\bar{A}$, respectively. Then for every $x \in \Sigma^*$, *either* M_A *or* $M_{\bar{A}}$ *halts* and accepts x.
We build a 2-tape Turing machine M by combining M_A and $M_{\bar{A}}$ "in parallel": on tape 1, M simulates machine M_A, and on tape 2 it simulates machine $M_{\bar{A}}$.

If the simulation on tape 1 halts in an accepting configuration, then M accepts the input. If the tape 2 simulation accepts, then M rejects the input.

Corollary 9.4 (A is semi-decidable but not decidable => Complement of A is not decidable)

Corollary 9.4

Let $A \subseteq \Sigma^*$ be a semi-decidable language that is not decidable. Then the language $\bar{A}$ is not semi-decidable.

Definition 9.2 (Countable and uncountable infinite sets)

Definition 9.2

- A set X is *countably infinite* if there is a bijection $f : \mathbb{N} \to X$.
- A set is *countable* if it is finite or countably infinite.
- A set that is not countable is called *uncountable*.

Lemma 9.5 (all strings in finite alphabet is countably infinite)

Lemma 9.5

Let Σ be a finite alphabet. The set Σ^* of all strings over Σ is countably infinite.

Proof

We construct a bijection $f : \mathbb{N} \to \Sigma^*$ as follows. Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$. We fix some arbitrary "alphabetical order" for the symbols in Σ, for instance, $a_1 < a_2 < \cdots < a_n$.
The strings in Σ^* can now be enumerated in shortlex order with respect to the chosen alphabetical order:

- We first enumerate strings of length 0 (i.e., ε), then those of length 1 (i.e., $a_1, a_2, \ldots, a_n$), then those of length 2, and so on.
- Inside each length group, the strings are enumerated in lexicographic order with respect to the chosen alphabetical order.

Theorem 9.6 (All languages over any alphabet is uncountable)

Theorem 9.6

The family of all languages over any alphabet Σ is uncountable.

Proof (Cantor's diagonal argument)

Denote the family $\mathcal{P}(\Sigma^*)$ of all languages over Σ by $\mathcal{A}$. Suppose that these languages could be enumerated in some order, say

$$\mathcal{A} = \{A_0, A_1, A_2, \ldots\}.$$

Let the set of all strings over Σ, enumerated in shortlex order, be $\Sigma^* = \{x_0, x_1, x_2, \ldots\}$. Using these orders, define a language $\tilde{A}$ as

$$\tilde{A} = \{x_i \in \Sigma^* \mid x_i \notin A_i\}.$$

Since $\tilde{A}$ belongs to the family $\mathcal{A}$, and we assumed that the languages in $\mathcal{A}$ can be enumerated, it must be the case that $\tilde{A} = A_k$ for some $k \in \mathbb{N}$. But now, according to the definition of $\tilde{A}$, we obtain the contradiction

$$x_k \in \tilde{A} \Leftrightarrow x_k \notin A_k = \tilde{A}.$$

Thus the assumption that $\mathcal{A}$ could be enumerated is wrong and $\mathcal{A}$ is uncountable.

$\tilde{A}$

$\searrow$	A_0	A_1	A_2	A_3 $\cdots$
x_0	$\not\emptyset$ (1)	0	0	1 $\cdots$
x_1	0	$\not 1$ (0)	0	0 $\cdots$
x_2	1	1	$\not 1$ (0)	1 $\cdots$
x_3	0	0	0	$\not\emptyset$ (1) $\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$ $\ddots$

Graphically, the idea of the proof can be illustrated as follows:

- We form the "incidence matrix" of the languages $A_0, A_1, A_2, \ldots$ and the strings $x_0, x_1, x_2, \ldots$.
- The cell at row i, column j has value: 1 if $x_i \in A_j$, 0 if $x_i \notin A_j$.
- Now the language $\tilde{A}$ differs from every language A_k on the "diagonal" of the matrix.

Universal Turing Machine (UTM)

What is a universal Turing machine?

- In "interpreted" computer programming languages such as Python or Java, programs are not compiled into machine code but executed (simulated) by a systems program called an *interpreter* (cf. Python interpreter, Java virtual machine).
- Similarly a "universal Turing machine" can execute (simulate) any other Turing machine, given its description as an input string.
- In modern terminology, a universal Turing machine is thus an interpreter for the "Turing machines" programming language, written in the same language.

- Let us define a *universal language* U [1] (over the binary alphabet $\{0,1\}$) as

$$U = \{c_M w \mid \text{Turing machine } M \text{ accepts string } w\}.$$

- The corresponding decision problem is:

  Given a Turing machine M and a string w.
  Does M accept the string w?

- If A is a semi-decidable language and M is a Turing machine recognising A, then

$$A = \{w \in \{0,1\}^* \mid c_M w \in U\}.$$

- The language U itself is semi-decidable, too. The machines recognising U are called *universal Turing machines*.

Theorem 9.8: Universal language is semi-decidable

Theorem 9.8

The language U is semi-decidable.

Proof

It is easiest to describe a Turing machine M_U recognising U in the 3-tape machine model. This can then be transformed into a standard 1-tape machine as explained in Lecture 8.

Theorem 9.9 Language U is not decidable

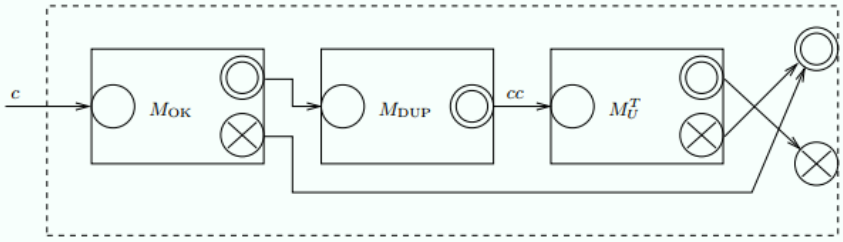
Theorem 9.9

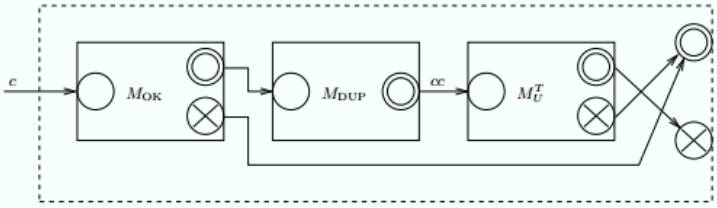
The language U is not decidable.

Proof

Suppose that U would be decidable. Then there is a total Turing machine M_U^T recognising U. We can now build a total Turing machine M_D that recognises the diagonal language D of Lemma 9.7 as follows.

Let M_{OK} be a total Turing machine that checks whether a string is a valid code for a Turing machine. Similarly, let M_{DUP} be a total Turing machine that transforms a string c into the string cc. We build the machine M_D by combining the machines M_U^T, M_{OK} and M_{DUP}:





Now clearly M_D is total whenever the machine M_U^T is, and M_D recognises D:

$$\begin{aligned} c \in \mathcal{L}(M_D) &\Leftrightarrow c \notin \mathcal{L}(M_{\mathsf{OK}}) \vee cc \notin \mathcal{L}(M_U^T) \\ &\Leftrightarrow c \notin \mathcal{L}(M_c) \\ &\Leftrightarrow c \in D. \end{aligned}$$

But by Theorem 9.7, the language D is not decidable. We have thus obtained a contradiction, and our assumption that U is decidable must be wrong.

Corollary 9.10 Complement of universal language is not semidecidable

Corollary 9.10

The language

$$\tilde{U} = \{c_M w \mid w \notin \mathcal{L}(M)\}$$

is not semi-decidable.

Proof

The language $\tilde{U}$ is effectively the same as the complement $\bar{U}$ of the universal language U. More precisely, $\bar{U} = \tilde{U} \cup \text{ERR}$, where ERR is the (obviously) decidable language

$$\text{ERR} = \{x \in \{0,1\}^* \mid x \text{ does not start with a valid code for a Turing machine}\}.$$

If the language $\tilde{U}$ were semi-decidable, then so would the language $\bar{U}$. But as the language U is semi-decidable, it would follow (by Lemma 9.3) that U is decidable. This contradicts Theorem 9.7 and therefore we must conclude that $\tilde{U}$ is not semi-decidable.

Lemma 9.7 (Diagonal lang is not semi-decidable)

Lemma 9.7

The "diagonal language"

$$D = \{c \in \{0,1\}^* \mid c \notin \mathcal{L}(M_c)\}$$

is not semi-decidable.

Proof (Cantor-style diagonal argument)

Suppose that D is semi-decidable; then $D = \mathcal{L}(M)$ for some Turing machine M. Let d be the binary encoding of M, i.e. $D = \mathcal{L}(M_d)$. Now

$$d \in D \quad \Leftrightarrow \quad d \notin \mathcal{L}(M_d) = D.$$

From the contradiction, we deduce that the assumption must be wrong and thus D is not semi-decidable.

Week 10: Undecidability problems

The Halting Problem (HP)

Theorem 10.1 (The halting language is semi-decidable but not decidable)

Theorem 10.1

The language

$$H = \{c_M w \mid \text{Turing machine } M \text{ halts on input } w\}$$

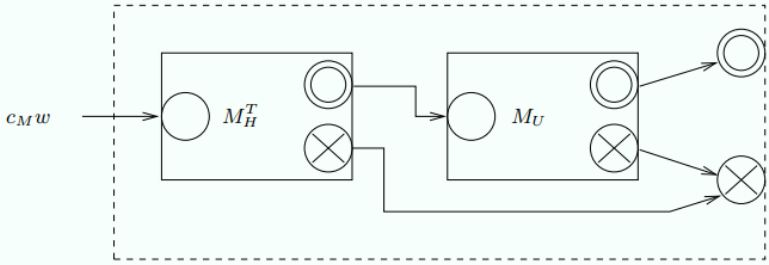
is semi-decidable but not decidable.

Proof

Let us first verify that H is semi-decidable. It is easy to modify the universal Turing machine M_U presented in the proof of Theorem 9.8 to a Turing machine that, on input $c_M w$ simulates the computation of machine M on input w and accepts if and only if the simulated computation halts (in either reject or accept state).

We next show that H is not decidable. Suppose that it were and that $H = \mathcal{L}(M_H^T)$ for some total Turing machine M_H^T. Suppose that M_H^T is such that when it halts, it leaves its original input on the tape (possibly extended with blank symbols). Let M_U be the universal Turing machine designed in the proof of Theorem 9.8.

We could now design a *total* Turing machine recognising U by combining the machines M_H^T and M_U is follows:




```
def U(m,w):
  wcopy = w
  if not M_H(m,w):
    reject
  return M_U(m,wcopy)
```

But according to Theorem 9.9 such a total Turing machine recognising U cannot exist. This contradiction means that our assumption must be wrong and H is not decidable.

Corollary 10.2 Complement of Halting language is not semi decidable. This is deduced from Theorem 10.1 and Corollary 9.4

Corollary 10.2

The language

$$\tilde{H} = \{c_M w \mid M \text{ does } \textit{not} \text{ halt on the input } w\}$$

is not semi-decidable.

10.2 The non-emptiness problem

- Consider the following *non-emptiness problem* for Turing machines:

  Given a Turing machine M.
  Does M accept any string?

- This problem corresponds to the formal language:

$$\text{NE} = \{c \in \{0,1\}^* \mid \mathcal{L}(M_c) \neq \emptyset\}.$$

Theorem 10.3 Non-eptiness (NE) language is semidecidable but not decidable

Theorem 10.3

The language NE is semi-decidable but not decidable.

Lemma 10.4 NE language is semidecidable

Lemma 10.4

The language NE is semi-decidable.

Lemma 10.5 NE language is not decidable

Lemma 10.5

The language NE is not decidable.

Theorem 10.6

Theorem 10.6 (Undecidability of FO logic; Church/Turing 1936)

There is no algorithm that, given a formula ϕ in first-order logic, decides whether the formula is valid (i.e. true in all possible interpretations).

Theorem 10.7

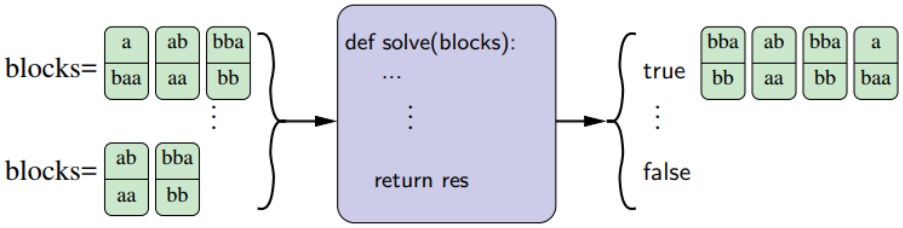
Theorem 10.7 ("Hilbert's tenth problem"; Matijasevitsh/Davis/Robinson/Putnam 1953–70)

There is no algorithm that, given a multivariate polynomial $P(x_1,\ldots,x_n)$ with integer coefficients, decides whether the polynomial has integer-valued zero points (i.e. tuples $(m_1,\ldots,m_n) \in \mathbf{Z}^n$ for which $P(m_1,\ldots,m_n) = 0$). The problem is undecidable already when $n = 15$ or $\deg(P) = 4$.

Theorem 10.8

10.4 Post's correspondence problem

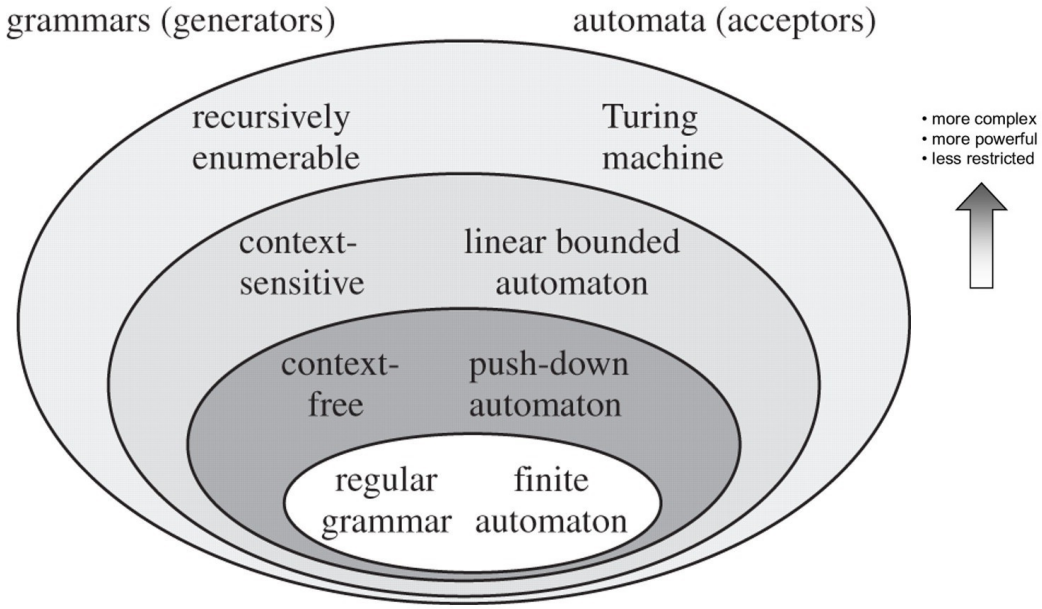
Given a finite set of domino block types (we can have arbitrarily many blocks of each type), can we have a finite sequence of blocks so that the upper and lower rows contain the same string?



Theorem 10.8

Post's correspondence problem is undecidable.

The Chomsky Hierachy



Synonyms
- Decidable / recursive language/ Turing decidable language
- Semi-decidable/ recursively enumerable/ Turing recognizable language
- Undecidable/ non recursively enumerable language. Undecidable lang can be semidecidable

Undecidability in the Chomsky hierarchy

The decidability and undecidability of some problems related to grammars, when given grammars G and G' of type i in Chomsky hierarchy and a string w. The abbreviations mean $D \sim$ "decidable", $U \sim$ "undecidable", $T \sim$ "always true".

	Type i:			
Problem: is...	3	2	1	0
$w \in \mathcal{L}(G)$?	D	D	D	U
$\mathcal{L}(G) = \emptyset$?	D	D	U	U
$\mathcal{L}(G) = \Sigma^*$?	D	U	U	U
$\mathcal{L}(G) = \mathcal{L}(G')$?	D	U	U	U
$\mathcal{L}(G) \subseteq \mathcal{L}(G')$?	D	U	U	U
$\mathcal{L}(G) \cap \mathcal{L}(G') = \emptyset$?	D	U	U	U
$\mathcal{L}(G)$ regular?	T	U	U	U
$\mathcal{L}(G) \cap \mathcal{L}(G')$ of type i?	T	U	T	T
$\overline{\mathcal{L}(G)}$ of type i?	T	U	T	U

Computable functions

10.6 Computable functions

- We define the *partial function*

$$f_M : \Sigma^* \to \Gamma^*$$

computed by a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ as:

$$f_M(x) = \begin{cases} u, \text{ if } (q_0, \underline{x}) \vdash_M^* (q, u\underline{a}v) \text{ where } q \in \{q_{\text{acc}}, q_{\text{rej}}\}, av \in \Gamma^*; \\ \text{undefined, if } M \text{ does not halt on input } x. \end{cases}$$

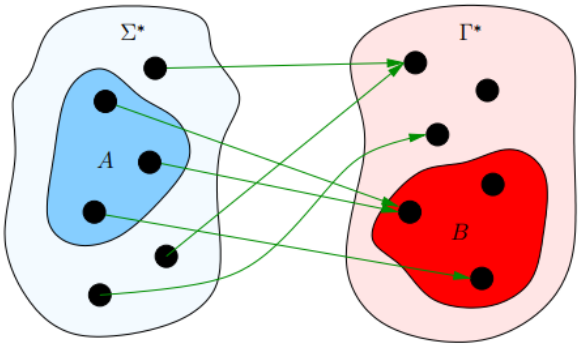
A partial function $f : \Sigma^* \to A$ is:

- *partially computable* (historically: *partially recursive*) if it can be computed by some Turing machine, and
- *computable* (historically: *recursive*) if it can be computed by some *total* Turing machine.
- *Note:* We could equivalently define that a partially computable function f is computable if its value $f(x)$ is defined for all x.

Reduction proof

10.7 Reductions between languages

- A language $A \subseteq \Sigma^\star$ can be *(computably) reduced* to a language $B \subseteq \Gamma^\star$, denoted as
$$A \leq_m B,$$
if there is a computable function $f : \Sigma^\star \to \Gamma^\star$ such that
$$x \in A \quad \Leftrightarrow \quad f(x) \in B, \quad \text{for all } x \in \Sigma^\star.$$
Such a function is called a *(computable many-one) reduction* from A to B.
- Graphically:



- Reductions can be used to translate solution methods (∼recognising/deciding automata) from one problem to another.

Lemma 10.8 Reduction proof

Lemma 10.8

If $A \leq_m B$ and B is a decidable language, then A is decidable as well.

Proof

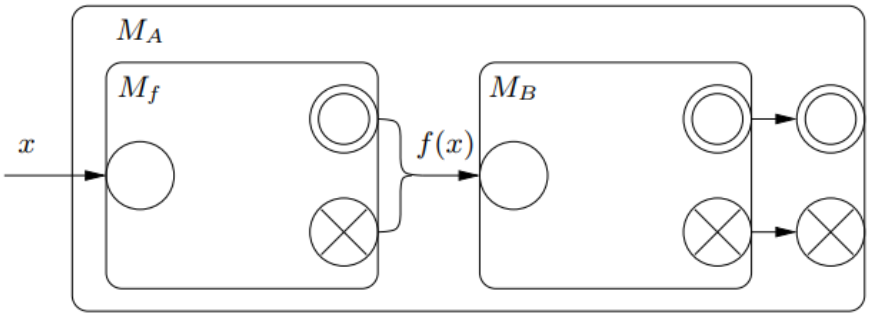
Let M_B be a total Turing machine recognising language B, and M_f a total Turing machine that computes the reduction f from language A to language B.

We can combine machines M_f and M_B into a total Turing machine M_A recognising A as follows: On input string w,

- first compute the value $f(w)$ using M_f, and
- then run machine M_B on input string $f(w)$.

The combined machine M_A is clearly total and accepts input string w if and only if $f(w) \in B$, i.e. $w \in A$.

- Graphically, the total Turing machine M_A recognising language A can be illustrated as below, where:
  - M_f is the total Turing machine computing reduction f, and
  - M_B is the total Turing machine recognising B.



Example:

Consider the decision problem DFA-EQ:

> *Given a pair of deterministic finite automata (M_1, M_2) over an alphabet Σ. Do M_1 and M_2 recognise the same language?*

This problem is decidable, since we can minimise the automata and check whether the results are the same (up to a renaming of the states).

Consider then the decision problem REX-COMP:

> *Given two regular expressions r_1 and r_2 over an alphabet Σ. Is r_1 equivalent to the complement of r_2, i.e., is $\mathcal{L}(r_1) = \Sigma^* \setminus \mathcal{L}(r_2)$?*

Also this is decidable, since we can reduce it to problem DFA-EQ:

- Given a pair (r_1, r_2) of regular expressions.
- The reduction produces a pair (M_1, M_2) of DFA, where:
  - M_1 is a DFA for the language $\mathcal{L}(r_1)$, and
  - M_2 is the "state-complement" of a DFA for the language $\mathcal{L}(r_2)$.

- By using reductions, we can also prove that some languages are *not* decidable:

Corollaro 10.9

Corollary 10.9

If $A \leq_m B$ and A is not decidable, then B is not decidable.

Proof

Assume that $A \leq_m B$ and that A is not decidable.

Now if B were decidable, then (by Lemma 10.8) also A should be decidable, which would be a contradiction.

- Showing that a language B is undecidable:
  - Choose a previously-known undecidable language A.
  - Design a reduction from language A to language B.
  - Conclude by Corollary 10.9 that B is undecidable as well.

Example: Proving undecidability of the halting problem by reduction

- We design a reduction mapping f from the undecidable universal language U to the "halting language" H.
- Given an arbitrary input $c_M w$ to problem U, the reduction f produces a string $f(c_M w) = c_{M'} w'$ with the property that:

$$c_M w \in U \Leftrightarrow c_{M'} w' \in H.$$

  In other words, the reduction will satisfy:

  M accepts input w if and only if M' halts on input w'

- ☞ If we can solve the problem "Does M' halt on input w'", we can also solve the problem "Does M accept input w".
- ☞ As the language U is undecidable and we can reduce it to language H, language H must also be undecidable (Cor 10.9)

Example: Proving undecidability of the non-emptiness problem by reduction

- We design a reduction mapping f from the undecidable universal language U to the "non-emptiness" language NE.
- Given an arbitrary input $c_M w$ to problem U, the reduction f produces a string $f(c_M w) = c_{M'}$, with the property that:

$$c_M w \in U \Leftrightarrow c_{M'} \in \text{NE}.$$

  In other words, the reduction will satisfy:

  M accepts input string w if and only if M' accepts some input string

- ☞ If we can solve the problem "Does M' accept *some* input string", we can also solve the problem "Does M accept input string w".
- ☞ As the language U is undecidable and we can reduce it to language NE, language NE must also be undecidable (Cor 10.9)

11.1 Rice's theorem

- Rice's Theorem states that *all* decision problems concerning the languages recognised by Turing machines[1] are undecidable.
- Let us denote the family of all semi-decidable (i.e. recursively enumerable) languages by **RE**.
- A *semantic property*[2] $\mathbf{S}$ of Turing machines is any family of semi-decidable languages, i.e. $\mathbf{S} \subseteq \mathbf{RE}$.
- A machine M *has property* $\mathbf{S}$ if $\mathcal{L}(M) \in \mathbf{S}$.
- Examples of semantic properties:
  - **NE** $= \{L \subseteq \{0,1\}^* \mid L \neq \emptyset\}$
  - **ALLSTRINGS** $= \{L \subseteq \{0,1\}^* \mid L = \{0,1\}^*\} = \{\{0,1\}^*\}$
  - **EVEN** $= \{L \subseteq \{0,1\}^* \mid |x| \text{ is even for all } x \in L\}$
  - **ONLY**$_w = \{L \subseteq \{0,1\}^* \mid x \in L \Leftrightarrow x = w\} = \{\{w\}\}$
  - **EMPTYSET** $= \{L \subseteq \{0,1\}^* \mid L = \emptyset\} = \{\emptyset\}$
- A semantic property is *trivial* if
  - $\mathbf{S} = \emptyset$ (no machine has this property) or
  - $\mathbf{S} =$ **RE** (all machines have this property)
- A property $\mathbf{S}$ is *decidable* if the language

$$\text{codes}(\mathbf{S}) = \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \in \mathbf{S}\}$$

  is decidable.
- In other words: A semantic property is decidable if one can algorithmically decide whether a given Turing machine has the property.[3]

Theorem 11.1

Theorem 11.1 (Rice 1953)

All non-trivial semantic properties of Turing machines are undecidable.

Example:

- Let us consider the *non-emptiness problem* for Turing machines from Lecture 10:

  Given a Turing machine M.
  Does the machine accept any strings?

- The corresponding semantic property is $\mathbf{NE} = \{L \in \mathbf{RE} \mid L \neq \emptyset\}$.
- The property is non-trivial because:
  - $\mathbf{NE} \neq \emptyset$ (witness any semi-decidable language $L \neq \emptyset$)
  - $\mathbf{NE} \subsetneq \mathbf{RE}$ (since $\emptyset \in \mathbf{RE} \setminus \mathbf{NE}$)
- Thus by Rice's theorem, the language

$$\begin{aligned}\text{codes}(\mathbf{NE}) &= \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \in \mathbf{NE}\} \\ &= \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \neq \emptyset\}\end{aligned}$$

  is undecidable. (Note that this is precisely the result in Lemma 10.5.)

Proof

- A simple generalisation of the proof of Lemma 10.5.
- Let $\mathbf{S}$ be any non-trivial semantic property.
- We can assume that $\emptyset \notin \mathbf{S}$; in other words, machines that recognise the empty language do not have the property.[a]
- As $\mathbf{S}$ is non-trivial, there is a Turing machine $M_{\mathbf{S}}$ that has the property $\mathbf{S}$, i.e. one for which $\mathcal{L}(M_{\mathbf{S}}) \neq \emptyset$ and $\mathcal{L}(M_{\mathbf{S}}) \in \mathbf{S}$ hold.

[a] If $\emptyset \in \mathbf{S}$, we can first show that the property $\bar{\mathbf{S}} = \mathbf{RE} \setminus \mathbf{S}$ is undecidable and then conclude that also $\mathbf{S}$ is undecidable; this is because $\text{codes}(\bar{\mathbf{S}}) = \{0,1\}^* \setminus \text{codes}(\mathbf{S})$.

H11.1 Prove, by using Rice's theorem, that the decision problem

> Given an arbitrary Turing machine M, is the language recognised by M finite?

i.e., the language

$$\{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \text{ is finite}\}$$

is undecidable. Define the semantic property you apply precisely, and explain why it is not a trivial one.

Semantic property S is a collection of languages that has such property S. Let's define the semantic property as the set of languages that are finite, called FINITE.
The corresponding semantic property is FINITE = { L ∈ RE | L is finite }

- Finite languages are those having a finite number of accepted strings. As we know, all finite languages are regular. If you have a finite set of strings that your languages matches, you can simply use alternation (string1|string2|...) to construct a regular expression to match them, or construct a finite automaton in a straightforward manner.
For example, a finite language can be constructed as follows:
L = { x | x ∈ (cat | dog | bird) }
- Note: It is not true that all regular languages are finite. Even something as simple as 'a*' is a regular expression that matches an infinite set of strings: '' (the empty string), 'a', 'aa', 'aaa', ...

Therefore, the property FINITE is non-trivial because:
FINITE ≠ ∅ (witness a semi-decidable language L ≠ ∅ proved by the example above)
FINITE ⊂ RE (since ∅ ∈ RE/NE)
Thus by Rice's theorem, the language
codes(FINITE) = { c_M | M is a Turing Machine and ℓ(M) ∈ RE }
= { c_M | M is a Turing Machine and ℓ(M) ≠ ∅ }
is undecidable (proven)

- We now prove that codes$(\mathbf{S})$ is undecidable by reducing the undecidable language U to it.
- Let (M,w) be any instance of the Turing machine acceptance problem, encoded as the string $c_M w$.
- From input $c_M w$ construct (the code for) a Turing machine M^w that on any input string x works as follows:
  - First run machine M on string w, and then:
    - if M accepts w, run $M_{\mathbf{S}}$ on x
    - if M rejects w (or doesn't halt), reject x (or don't halt)
- Now M^w recognises the language

$$\mathcal{L}(M^w) = \begin{cases} \mathcal{L}(M_{\mathbf{S}}) & \text{if } w \in \mathcal{L}(M) \\ \emptyset & \text{if } w \notin \mathcal{L}(M) \end{cases}$$

- Thus M accepts w if and only if M^w has the property $\mathbf{S}$. That is, $c_M w \in U$ if and only if $c_{M^w} \in$ codes$(\mathbf{S})$.
- Therefore, codes$(\mathbf{S})$ is an undecidable language.

General grammar

Definition 11.1 Unrestricted grammar

In automaton, Unrestricted Grammar or Phrase Structure Grammar is the most general in the Chomsky Hierarchy of classification. This is type0 grammar, generally used to generate Recursively Enumerable languages. It is called unrestricted because no other restriction is made on this except each of their left hand sides being non-empty. The left hand sides of the rules can contain terminal and non-terminal, but the condition is at least one of them must be non-terminal. A Turning Machine can simulate Unrestricted Grammar and Unrestricted Grammar can simulate Turning Machine configurations. It can always be found for the language recognized or generated by any Turning Machine.

Definition 11.1

An *unrestricted grammar* is a quadruple

$$G = (V, \Sigma, R, S),$$

where

- V is a finite set of *variables*;
- Σ is a finite set, disjoint from V, of *terminals*;
- $R \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ is a finite set of *rules* (also called productions), where $(V \cup \Sigma)^+ = (V \cup \Sigma)^* \setminus \{\varepsilon\}$;
- $S \in V$ is the *start variable*.

A rule $(\omega, \omega') \in R$ is usually written as $\omega \to \omega'$.

Example:

An unrestricted grammar for the non-context-free language $\{a^k b^k c^k \mid k \geq 0\}$:

$$
\begin{array}{rcl@{\qquad}rcl}
S & \to & LT \mid \varepsilon & LA & \to & a \\
T & \to & ABCT \mid ABC & aA & \to & aa \\
BA & \to & AB & aB & \to & ab \\
CB & \to & BC & bB & \to & bb \\
CA & \to & AC & bC & \to & bc \\
 & & & cC & \to & cc
\end{array}
$$

A derivation of string $aabbcc$ in the grammar:

$$
\begin{array}{rcl@{\quad}rcl@{\quad}rcl}
\underline{S} & \Rightarrow & \underline{LT} \Rightarrow LABC\underline{T} & \Rightarrow & LAB\underline{CA}BC & \Rightarrow & LA\underline{BA}CBC \\
 & \Rightarrow & LAAB\underline{CB}C & \Rightarrow & \underline{LA}ABBCC & \Rightarrow & \underline{aA}BBCC \\
 & \Rightarrow & a\underline{aB}BCC & \Rightarrow & aa\underline{bB}CC & \Rightarrow & aabb\underline{bC}C \\
 & \Rightarrow & aabbc\underline{cC} & \Rightarrow & aabbcc & &
\end{array}
$$

Theorem 11.2

Theorem 11.2

If a language L can be generated with an unrestricted grammar, then it can be recognised with a Turing machine.

Theorem 11.3

Theorem 11.3

If a language L can be recognised with a Turing machine, then it can be generated with an unrestricted grammar.

Unrestricted grammar ⇔ Turing recognizable

Theorem 11.4 Context-sensitive grammar

Theorem 11.4

A language L is context-sensitive if and only if it can be recognised with a non-deterministic Turing machine that does not use more tape space than was already allocated for the input.

- The machines in Theorem 11.4 are called *linear bounded automata*.
- It is an open problem whether the non-determinism in Theorem 11.4 is necessary or not. (The “LBA ?= DLBA” problem.)

- An unrestricted grammar is *context-sensitive* if all its rules are of form $\omega \to \omega'$, where $|\omega'| \geq |\omega|$, or $S \to \varepsilon$, where S is the start variable.
- In addition, it is required that if the grammar contains the rule $S \to \varepsilon$, then the start variable S does not occur on the right-hand side of any rule.
- A language L is *context-sensitive* if it can be generated with some context-sensitive grammar.
- *A normal form for context-sensitive grammars:* Each context-sensitive language can be generated with a grammar whose rules are of form $S \to \varepsilon$ and $\alpha A \beta \to \alpha \omega \beta$, where A is a variable and $\omega \neq \varepsilon$.
- A rule $\alpha A \beta \to \alpha \omega \beta$ can be interpreted as the application of a rule $A \to \omega$ "in the context" $\alpha \_ \beta$.

The **Context Sensitive Grammar** is formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of **terminal** and **non-terminal** grammar. It is less general than Unrestricted Grammar and more general than Context Free Grammar.

The language generated by the context-sensitive grammar is called **Context Sensitive Language**. **Context Sensitive Language** has the following properties -

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- The Kleene plus of a context-sensitive language is context-sensitive.
- Every context-sensitive language is recursive.
- Complement of a context-sensitive language is context-sensitive.

Example of Context-Sensitive Grammar

Suppose **P** is set of rules and a context-sensitive grammar **G** is -

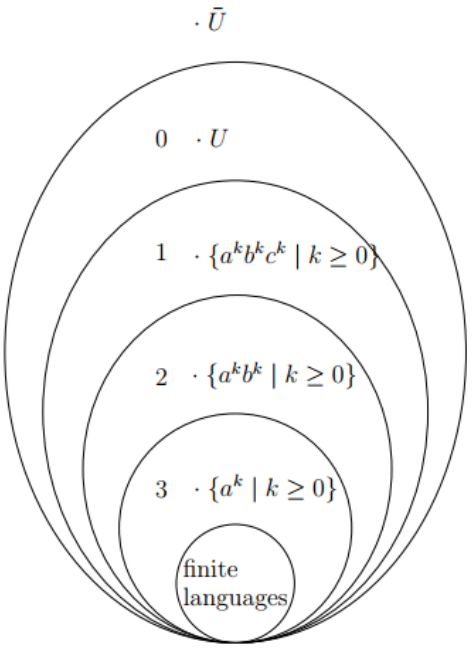
```
G = {{S, A,B,C,a,b,c},{a,b,c},P,S}
```

```
S -> aSBC
S -> aBC
CB -> BC
aB -> ab
bB -> bb
bC -> bc
cC -> cc
```

Then, the language generated by the grammar G is -

$\{a^n b^n c^n | n >= 1\}$

11.4 Recap: The Chomsky hierarchy



A classification of grammars, languages generated by grammars and recogniser automata classes:
Type-0: unrestricted grammars / semi-decidable languages / Turing machines
Type-1: context-sensitive grammars / context-sensitive languages / linear bounded automata
Type-2: context-free grammars / context-free languages / pushdown automata
Type-3: right and left linear grammars / regular languages / finite automata

https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

```
S → X a
X → a
X → aX
X → abc
X → ε
```

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in N$ (Non-terminal)

and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

```
AB → AbBc
A → bcA
B → b
```

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

```
S → ACaB
Bc → acB
CB → DB
aD → Db
```